### MPI: A Message-Passing Interface Standard

Version 4.0

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

November 15, 2020

#-... all changes
 #-nnn by issue nnn
 #-PRnnn by pull request
 Non problematic fomatting-changes and typo-corrections are not listed
 Embiggening: only new \_c versions are marked, all other is not listed
 New or changed content
 #-error
 #-error!
 Errors in RC 40 - also marked with #-error or #-error!
 B.m.n: i. - refers to Change-Log section B.m.n, Item i
 Updates after the rc-40 vote

1 2 3 4 5 6	This document describes the 2020 Draft Specification of the Message-Passing Interface (MPI) standard, intended for comment. It is not an official version of the standard. The MPI standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for
7	C and Fortran are defined.
8	Historically, the evolution of the standards is from MPI-1.0 (May 5, 1994) to MPI-1.1
9 10	(June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and
11	published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality, to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2
12	and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008),
13	combining the previous documents. Version MPI-2.2 (September 4, 2009) added additional
14	clarifications and seven new routines. Version MPI-3.0 (September 21, 2012) is an extension
15	of MPI-2.2. Version MPI-3.1 (June 4, 2015) adds clarifications and minor extensions to
16	MPI-3.0.
17	
18	Comments. Please send comments on MPI to the MPI Forum as follows:
19	
20	1. Subscribe to https://lists.mpi-forum.org/mailman/listinfo/mpi-comments
21	2. Send your comment to: mpi-comments@lists.mpi-forum.org, together with the version
22	of the MPI standard and the page and line numbers on which you are commenting.
23	Your comment will be forwarded to MPI Forum committee members for consideration.
24 25	Messages sent from an unsubscribed e-mail address will not be considered.
26	hieldsages sent from all discusserised e final address will not be considered.
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, 2020 University of Tennessee,
46	Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted,
47	provided the University of Tennessee copyright notice and the title of this document appear,
48	and notice is given that copying is by permission of the University of Tennessee.

### Unofficial Draft for Comment Only

2020 Draft Specification, November, 2020. This document contains a draft of the MPI specification as of the date of publication. It has not been adopted as an official MPI specification, and is provided for comment only. This document includes a number of new features that will be present in the final MPI-4.0 document. The largest changes are the addition of persistent collectives, partitioned communication, application info assertions, and improvements to the definitions of error handling. In addition, there are a number of smaller improvements and corrections.

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.

### Unofficial Draft for Comment Only

 $^{31}$ 

#-other

#-done

Done

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

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

<sup>14</sup> Version 1.0: May, 1994. The Message-Passing Interface Forum (MPIF), with participation from over 40 organizations, has been meeting since January 1993 to discuss and define a set of library interface standards for message passing. MPIF is not sanctioned or supported by any official standards organization.

<sup>18</sup> The goal of the Message-Passing Interface, simply stated, is to develop a widely used <sup>19</sup> standard for writing message-passing programs. As such the interface should establish a <sup>20</sup> practical, portable, efficient, and flexible standard for message-passing.

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

## Contents

Li	st of	Figures	ix
Li	st of	Tables	cxi
A	cknov	wledgments xx	iii
1	Intr	roduction to MPI	1
	1.1	Overview and Goals	1
	1.2	Background of MPI-1.0	2
	1.3	Background of MPI-1.1, MPI-1.2, and MPI-2.0	2
	1.4	Background of MPI-1.3 and MPI-2.1	3
	1.5	Background of MPI-2.2	4
	1.6	Background of MPI-3.0	4
	1.7	Background of MPI-3.1	4
	1.8	Background of 2020 Draft Specification	5
	1.9	Who Should Use This Standard?	5
	1.10	What Platforms Are Targets for Implementation?	5
	1.11	What Is Included in the Standard?	5
	1.12	What Is Not Included in the Standard?	6
	1.13	Organization of This Document	6
2	MPI	Terms and Conventions	9
	2.1	Document Notation	9
	2.2	Naming Conventions	9
	2.3	Procedure Specification	10
	2.4	Semantic Terms	11
		2.4.1 MPI Operations	11
		2.4.2 MPI Procedures	13
		2.4.3 MPI Datatypes	15
	2.5	Data Types	16
		2.5.1 Opaque Objects	16
		2.5.2 Array Arguments	18
		2.5.3 State	18
		2.5.4 Named Constants	18
		2.5.5 Choice	19
		2.5.6 Absolute Addresses and Relative Address Displacements	20
		2.5.7 File Offsets	20
		2.5.8 Counts	20

	2.6	Langua	age Binding
		2.6.1	Deprecated and Removed Interfaces
		2.6.2	Fortran Binding Issues
		2.6.3	C Binding Issues
		2.6.4	Functions and Macros
	2.7	Proces	ses
	2.8	Error 1	Handling
	2.9	Implen	nentation Issues
		2.9.1	Independence of Basic Runtime Routines
		2.9.2	Interaction with Signals
	2.10	Examp	bles
3	Poir	nt-to-P	oint Communication 29
	3.1	Introdu	uction
	3.2	Blocki	ng Send and Receive Operations
		3.2.1	Blocking Send
		3.2.2	Message Data
		3.2.3	Message Envelope
		3.2.4	Blocking Receive
		3.2.5	Return Status
		3.2.6	Passing MPI_STATUS_IGNORE for Status
		3.2.7	Send-Receive
	3.3	Data 7	Cype Matching and Data Conversion    42
		3.3.1	Type Matching Rules
			Type MPI_CHARACTER
		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 Mode57
	3.7		ocking Communication
			Communication Request Objects
		3.7.2	Communication Initiation
		3.7.3	Communication Completion
4		3.7.4	Semantics of Nonblocking Communications
		3.7.5	Multiple Completions    72
		3.7.6	Non-Destructive Test of status
	3.8	Probe	and Cancel
		3.8.1	Probe
		3.8.2	Matching Probe 83
		3.8.3	Matched Receives
		3.8.4	Cancel
	3.9		ent Communication Requests
	3.10	Null P	rocesses

4	Par	titione	ed Point-to-Point Communication	99
	4.1		luction	
	4.2	Seman	ntics of Partitioned Point-to-Point Communication	. 100
		4.2.1	Communication Initialization and Starting with Partitioning	. 102
		4.2.2	Communication Completion under Partitioning	. 106
		4.2.3	Semantics of Communications in Partitioned Mode	. 107
	4.3	Partit	ioned Communication Examples	. 108
		4.3.1	Partition Communication with Threads/Tasks Using OpenMP 4.0	)
			and greater	. 108
		4.3.2	Send-only Partitioning Example with Tasks and OpenMP version 4.0	
			and greater	. 109
		4.3.3	Send and Receive Partitioning Example with OpenMP version 4.0	)
			and greater	. 111
5	Dat	atypes		115
	5.1	Derive	ed Datatypes	. 115
		5.1.1	Type Constructors with Explicit Addresses	
		5.1.2	Datatype Constructors	. 117
		5.1.3	Subarray Datatype Constructor	. 129
		5.1.4	Distributed Array Datatype Constructor	. 131
		5.1.5	Address and Size Functions	. 137
		5.1.6	Lower-Bound and Upper-Bound Markers	. 140
		5.1.7	Extent and Bounds of Datatypes	. 142
		5.1.8	True Extent of Datatypes	
		5.1.9	Commit and Free	. 146
		5.1.10	Duplicating a Datatype	. 148
		5.1.11	Use of General Datatypes in Communication	. 148
		5.1.12	Correct Use of Addresses	. 152
		5.1.13	Decoding a Datatype	. 153
		5.1.14	Examples	. 162
	5.2	Pack a	and Unpack	. 170
	5.3	Canon	nical MPI_PACK and MPI_UNPACK	. 178
			Communication	183
4			luction and Overview	
	6.2		unicator Argument	
		6.2.1	Specifics for Intra-Communicator Collective Operations	
		6.2.2		
		6.2.3	Specifics for Inter-Communicator Collective Operations	
	6.3		$\mathbf{r} \mathbf{Synchronization}  \dots  \dots  \dots  \dots  \dots  \dots  \dots  \dots  \dots  $	
	6.4		$\operatorname{cast}$	
		6.4.1	Example using MPI_BCAST	
	6.5		r	
		6.5.1	Examples using MPI_GATHER, MPI_GATHERV	
	6.6	Scatte	<b>r</b>	
		6.6.1	Examples using MPI_SCATTER, MPI_SCATTERV	
	6.7		r-to-all	
		6.7.1	Example using MPI_ALLGATHER	. 212

6.8	All-to-	All Scatter/Gather	213
6.9	Global	Reduction Operations	219
	6.9.1	Reduce	220
	6.9.2	Predefined Reduction Operations	222
	6.9.3	Signed Characters and Reductions	225
	6.9.4	MINLOC and MAXLOC	225
	6.9.5	User-Defined Reduction Operations	229
		Example of User-Defined Reduce	233
	6.9.6	All-Reduce	234
	6.9.7	Process-Local Reduction	236
6.10		e-Scatter	238
	6.10.1	MPI_REDUCE_SCATTER_BLOCK	238
		MPI_REDUCE_SCATTER	239
6.11	Scan .		241
	6.11.1	Inclusive Scan	241
	6.11.2	Exclusive Scan	243
	6.11.3	Example using MPI_SCAN	244
6.12	Nonble	ocking Collective Operations	246
		Nonblocking Barrier Synchronization	248
	6.12.2	Nonblocking Broadcast	248
		Example using MPI_IBCAST	249
	6.12.3	Nonblocking Gather	250
	6.12.4	Nonblocking Scatter	253
	6.12.5	Nonblocking Gather-to-all	256
	6.12.6	Nonblocking All-to-All Scatter/Gather	258
	6.12.7	Nonblocking Reduce	264
	6.12.8	Nonblocking All-Reduce	265
	6.12.9	Nonblocking Reduce-Scatter with Equal Blocks	266
		Nonblocking Reduce-Scatter	268
	6.12.11	Nonblocking Inclusive Scan	269
	6.12.12	Nonblocking Exclusive Scan	270
6.13		ent Collective Operations	271
		Persistent Barrier Synchronization	272
	6.13.2	Persistent Broadcast	273
		Persistent Gather	274
		Persistent Scatter	277
		Persistent Gather-to-all	280
		Persistent All-to-All Scatter/Gather	283
		Persistent Reduce	289
		Persistent All-Reduce	290
		Persistent Reduce-Scatter with Equal Blocks	291
		Persistent Reduce-Scatter	293
		Persistent Inclusive Scan	294
		Persistent Exclusive Scan	295
6.14	Correc	tness	296

7	Gro	ups, C	Contexts, Communicators, and Caching	<b>305</b>
	7.1	Introd	uction	. 305
		7.1.1	Features Needed to Support Libraries	. 305
		7.1.2	MPI's Support for Libraries	. 306
	7.2	Basic	Concepts	. 308
		7.2.1	Groups	. 308
		7.2.2	Contexts	. 308
		7.2.3	Intra-Communicators	. 309
		7.2.4	Predefined Intra-Communicators	. 309
	7.3	Group	Management	. 310
		7.3.1	Group Accessors	. 310
		7.3.2	Group Constructors	. 312
		7.3.3	Group Destructors	. 318
	7.4	Comm	nunicator Management	. 319
		7.4.1	Communicator Accessors	. 319
		7.4.2	Communicator Constructors	321
		7.4.3	Communicator Destructors	. 338
		7.4.4	Communicator Info	. 339
	7.5	Motiva	ating Examples	. 341
		7.5.1	Current Practice $\#1$	. 341
		7.5.2	Current Practice $#2$	. 342
		7.5.3	(Approximate) Current Practice #3	
		7.5.4	Example #4	. 344
		7.5.5	Library Example #1	. 345
		7.5.6	Library Example $#2 \dots \dots$	. 346
	7.6	Inter-O	Communication	. 348
		7.6.1	Inter-Communicator Accessors	
		7.6.2	Inter-Communicator Operations	
		7.6.3	Inter-Communication Examples	
			Example 1: Three-Group "Pipeline"	
			Example 2: Three-Group "Ring"	
	7.7		ng	
		7.7.1	Functionality	
	A	7.7.2	Communicators	
		7.7.3	Windows	
		7.7.4	Datatypes	
		7.7.5	Error Class for Invalid Keyval	
		7.7.6	Attributes Example	
	7.8		ng Objects	
	7.9		lizing the Loosely Synchronous Model	
		7.9.1	Basic Statements	
		7.9.2	Models of Execution	
			Static Communicator Allocation	
			Dynamic Communicator Allocation	
			The General Case	. 381

8	Pro	ocess Topologies 3	83
	8.1	Introduction	383
	8.2	Virtual Topologies	384
	8.3	Embedding in MPI	384
	8.4	Overview of the Functions	385
	8.5	Topology Constructors	386
		8.5.1 Cartesian Constructor	386
		8.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE	387
		8.5.3 Graph Constructor	388
		8.5.4 Distributed Graph Constructor	390
		8.5.5 Topology Inquiry Functions	397
		8.5.6 Cartesian Shift Coordinates	105
		8.5.7 Partitioning of Cartesian Structures	107
		8.5.8 Low-Level Topology Functions	108
	8.6		410
		8.6.1 Neighborhood Gather	410
			415
	8.7	Nonblocking Neighborhood Communication	422
		8.7.1 Nonblocking Neighborhood Gather	422
		8.7.2 Nonblocking Neighborhood Alltoall	125
	8.8	Persistent Neighborhood Communication	129
			130
			133
	8.9	An Application Example	137
9			43
	9.1	1	143
			443
			145
		5	145
			145
			146
			146
		*	146
	9.2		147
	9.3	0	450
			453
			155
		9.3.3 Error Handlers for Files	457
		9.3.3Error Handlers for Files49.3.4Error Handlers for Sessions4	458
	0.4	9.3.3Error Handlers for Files49.3.4Error Handlers for Sessions49.3.5Freeing Errorhandlers and Retrieving Error Strings4	458 460
	9.4	9.3.3Error Handlers for Files49.3.4Error Handlers for Sessions49.3.5Freeing Errorhandlers and Retrieving Error Strings4Error Codes and Classes4	458 460 461
	9.5	9.3.3       Error Handlers for Files       4         9.3.4       Error Handlers for Sessions       4         9.3.5       Freeing Errorhandlers and Retrieving Error Strings       4         Error Codes and Classes       4         Error Classes, Error Codes, and Error Handlers       4	458 460 461 462
	-	9.3.3Error Handlers for Files49.3.4Error Handlers for Sessions49.3.5Freeing Errorhandlers and Retrieving Error Strings4Error Codes and Classes4Error Classes, Error Codes, and Error Handlers4	458 460 461

#### х

11 Pro	cess In	itialization, Creation, and Management	<b>479</b>
11.1	Introdu	uction	479
11.2	The W	Vorld Model	480
	11.2.1	Starting MPI Processes	480
	11.2.2	Finalizing MPI	486
		Determining Whether MPI Has Been Initialized When Using the World	
		Model	
	11.2.4	Allowing User Functions at MPI Finalization	
11.3		ssions Model	
11.0		Session Creation and Destruction Methods	
		Processes Sets	
		Runtime Query Functions	
		Sessions Model Examples	
11 /	11.J.4 Comm	on Elements of Both Process Models	498 504
11.4		MDI Trusstianalitas that is Alarman Assilable	504
	11.4.1	MPI Functionality that is Always Available	504
11 5	11.4.2	Aborting MPI Processes	505
11.5	Portab	le MPI Process Startup	506
11.6	MPI ar	nd Threads	508
		General	
		Clarifications	
11.7		ynamic Process Model	
		Starting Processes	
		The Runtime Environment	
11.8	Proces	s Manager Interface	513
	11.8.1	Processes in MPI	513
	11.8.2	Starting Processes and Establishing Communication	513
		Starting Multiple Executables and Establishing Communication	
	11.8.4	Reserved Keys	521
	11.8.5	Spawn Example	522
11.9	Establi	ishing Communication	523
	11.9.1	Names, Addresses, Ports, and All That	524
	11.9.2	Server Routines	525
		Client Routines	
		Name Publishing	529
		Reserved Key Values	531
		Client/Server Examples	532
11.1(		Functionality	534
11.10		Universe Size	534
		Singleton MPI Initialization	535
		MPI_APPNUM	536
		Releasing Connections	
		Another Way to Establish MPI Communication	
	11.10.0	Another way to Establish with Communication	000
<b>12 One</b>	-Sided	Communications	<b>541</b>
		uction	541
	Initiali		542
		Window Creation	543
		Window That Allocates Memory	
		······································	510

	12.2.3 Window That Allocates Shared Memory
	12.2.4 Window of Dynamically Attached Memory
	12.2.5 Window Destruction
	12.2.6 Window Attributes
	12.2.7 Window Info
12.3	Communication Calls
	12.3.1 Put
	12.3.2 Get
	12.3.3 Examples for Communication Calls
	12.3.4 Accumulate Functions
	Accumulate Function
	Get Accumulate Function
	Fetch and Op Function
	Compare and Swap Function
	12.3.5 Request-based RMA Communication Operations
12.4	Memory Model
	Synchronization Calls
	12.5.1 Fence
	12.5.2 General Active Target Synchronization
	12.5.3 Lock
	12.5.4 Flush and Sync
	12.5.5 Assertions
	12.5.6 Miscellaneous Clarifications
12.6	Error Handling
	12.6.1 Error Handlers
	12.6.2 Error Classes
12.7	Semantics and Correctness
	12.7.1 Atomicity
	12.7.2 Ordering
	12.7.3 Progress
	12.7.4 Registers and Compiler Optimizations
12.8	Examples
	ernal Interfaces 62
	Introduction
13.2	Generalized Requests
	13.2.1 Examples
13.3	Associating Information with Status
141/0	
14 I/O	633 1. /
14.1	Introduction
14.0	$14.1.1 \text{ Definitions} \qquad $
14.2	File Manipulation       63:         14.2.1       Opening a File         62:
	14.2.1 Opening a File
	14.2.2 Closing a File
	14.2.3 Deleting a File
	14.2.4 Resizing a File
	14.2.5 Preallocating Space for a File

14.2.6	Querying the Size of a File	639
14.2.7	Querying File Parameters	639
14.2.8	File Info	641
	Reserved File Hints	642
14.3 File V	liews	644
14.4 Data .	Access	647
14.4.1	Data Access Routines	647
	Positioning	648
	Synchronism	648
	Coordination	649
	Data Access Conventions	649
14.4.2	Data Access with Explicit Offsets	650
14.4.3	Data Access with Individual File Pointers	657
14.4.4	Data Access with Shared File Pointers	667
	Noncollective Operations	668
	Collective Operations	671
	Seek	674
14.4.5	Split Collective Data Access Routines	675
14.5 File Ir	nteroperability	684
14.5.1	Datatypes for File Interoperability	686
	External Data Representation: "external32"	688
14.5.3	User-Defined Data Representations	691
	Extent Callback	692
	Datarep Conversion Functions	693
14.5.4	Matching Data Representations	695
	stency and Semantics	696
14.6.1	File Consistency	696
14.6.2	Random Access vs. Sequential Files	699
14.6.3	Progress	699
	Collective File Operations	700
14.6.5	Nonblocking Collective File Operations	700
14.6.6	Type Matching	700
14.6.7	Miscellaneous Clarifications	701
14.6.8	MPI_Offset Type	701
14.6.9	Logical vs. Physical File Layout	701
14.6.1	0 File Size	701
14.6.1	1 Examples	702
	Asynchronous I/O	704
14.7 I/O E	rror Handling	706
14.8 I/O E	rror Classes	707
14.9 Exam		708
14.9.1	Double Buffering with Split Collective I/O	708
14.9.2	Subarray Filetype Constructor	709

<b>15 Tool</b>	Supp	ort	713
15.1	Introd	uction	713
15.2	Profilii	ng Interface	713
	15.2.1	Requirements	713
	15.2.2	Discussion	714
	15.2.3	Logic of the Design	714
	15.2.4	Miscellaneous Control of Profiling	715
	15.2.5	Profiler Implementation Example	716
	15.2.6	MPI Library Implementation Example	716
		Systems with Weak Symbols	716
		Systems without Weak Symbols	717
	15.2.7	Complications	717
		Multiple Counting	717
		Linker Oddities	718
		Fortran Support Methods	718
	15.2.8	Multiple Levels of Interception	718
15.3 $'$	The M	PI Tool Information Interface	719
	15.3.1	Verbosity Levels	720
		Binding MPI Tool Information Interface Variables to MPI Objects .	720
	15.3.3	Convention for Returning Strings	721
	15.3.4	Initialization and Finalization	722
		Datatype System	723
	15.3.6	Control Variables	726
		Control Variable Query Functions	726
		Example: Printing All Control Variables	729
		Handle Allocation and Deallocation	730
		Control Variable Access Functions	731
		Example: Reading the Value of a Control Variable	732
	15.3.7	Performance Variables	733
		Performance Variable Classes	733
		Performance Variable Query Functions	735
		Performance Experiment Sessions	738
		Handle Allocation and Deallocation	739
		Starting and Stopping of Performance Variables	740
		Performance Variable Access Functions	741
		Example: Tool to Detect Receives with Long Unexpected Message	
		Queues	743
	15.3.8	Events	745
		Event Sources	746
		Callback Safety Requirements	748
		Event Type Query Functions	750
		Handle Allocation and Deallocation	753
		Handling Dropped Events	757
		Reading Event Data	759
		Reading Event Meta Data	760
	15.3.9	Variable Categorization	761
		Category Query Functions	762
		Category Member Query Functions	764

	15.3.10 Return Codes for the MPI Tool Information Interface	766
	15.3.11 Profiling Interface	767
16 Der	precated Interfaces	769
	Deprecated since MPI-2.0	769
	Deprecated since MPI-2.2	772
	Deprecated since MPI-3.2	772
	Deprecated since MPI-4.0	772
17 Dor	noved Interfaces	775
	Removed MPI-1 Bindings	775
11.1	17.1.1 Overview	775
	17.1.2 Removed MPI-1 Functions	775
	17.1.2 Removed MPI-1 Datatypes	775
	17.1.3 Removed MPI-1 Constants	775
	17.1.5 Removed MPL1 Collback Prototypes	776
17.9	17.1.5 Removed MPI-1 Callback Prototypes	776
11.2	C++ bindings	110
	kward Incompatibilities	777
18.1	Backward Incompatible since MPI-3.2	777
19 Lan	guage Bindings	779
19.1	Support for Fortran       19.1.1 Overview	779
	19.1.1 Overview	779
	19.1.2 Fortran Support Through the mpi_f08 Module	780
	19.1.3 Fortran Support Through the mpi Module	783
	19.1.4 Fortran Support Through the mpif.h Include File	785
	19.1.5 Interface Specifications, Procedure Names, and the Profiling Interface	786
	19.1.6 MPJ for Different Fortran Standard Versions	791
	19.1.7 Requirements on Fortran Compilers	795
	19.1.8 Additional Support for Fortran Register-Memory-Synchronization	796
	19.1.9 Additional Support for Fortran Numeric Intrinsic Types	797
	Parameterized Datatypes with Specified Precision and Exponent Range	<mark>e</mark> 798
	Support for Size-specific MPI Datatypes	802
	Communication With Size-specific Types	804
	19.1.10 Problems With Fortran Bindings for MPI	805
	19.1.11 Problems Due to Strong Typing	806
	19.1.12 Problems Due to Data Copying and Sequence Association with Sub-	
	script Triplets	807
	19.1.13 Problems Due to Data Copying and Sequence Association with Vector	
	Subscripts	810
	19.1.14 Special Constants	810
	19.1.15 Fortran Derived Types	811
	19.1.16 Optimization Problems, an Overview	812
	19.1.17 Problems with Code Movement and Register Optimization	813
	Nonblocking Operations	813
	Persistent Operations	815
	One-sided Communication	815

		MPI_BOTTOM and Combining Independent Variables in Datatyp	es.	815
		Solutions		816
		The Fortran ASYNCHRONOUS Attribute		816
		Calling MPI_F_SYNC_REG		818
		A User Defined Routine Instead of MPI_F_SYNC_REG		819
		Module Variables and COMMON Blocks		819
		The (Poorly Performing) Fortran VOLATILE Attribute		820
		The Fortran TARGET Attribute		
	19.1.18	8 Temporary Data Movement and Temporary Memory Modification		
		9 Permanent Data Movement		821
		0 Comparison with C		822
19.2		rt for Large Count and Large Byte Displacement		
19.3	Langu	age Interoperability		828
10.0		Introduction		
		Assumptions		
	10.3.2	Initialization		829
		Transfer of Handles		
	19.0.4		•••	829 831
		Status		
	19.3.0	MPI Opaque Objects		834
		Datatypes		
		Callback Functions		
		Error Handlers		836
	100-	Reduce Operations		
		Attributes		
	19.3.8	Extra-State		841
		Constants		
	19.3.10	0 Interlanguage Communication		842
A T		D'a l'e a Canada		945
	Defe	Bindings Summary ad Values and Handles		845
A.1	Denne	Defined Constants		845
		Defined Constants		
		Types		
	A.1.3	Prototype Definitions		
		C Bindings		
		Fortran 2008 Bindings with the mpi_f08 Module		862
		Fortran Bindings with mpif.h or the mpi Module		865
	A.1.4	Deprecated Prototype Definitions		867
				867
	A.1.6	Info Values		868
A.2		ary of the Semantics of all OpRelated Routines		869
A.3	C Bine	dings		870
	A.3.1	Point-to-Point Communication C Bindings		870
	A.3.2	Partitioned Communication C Bindings		873
	A.3.3	Datatypes C Bindings		873
	A.3.4	Collective Communication C Bindings		877
	A.3.5	Groups, Contexts, Communicators, and Caching C Bindings .		885
	A.3.6	Process Topologies C Bindings		888
	A.3.7	MPI Environmental Management C Bindings		892

		A.3.8 The Info Object C Bindings	893
		A.3.9 Process Creation and Management C Bindings	894
		A.3.10 One-Sided Communications C Bindings	895
		A.3.11 External Interfaces C Bindings	898
		A.3.12 I/O C Bindings	898
		A.3.13 Language Bindings C Bindings	
		A.3.14 Tools / Profiling Interface C Bindings	
		A.3.15 Tools / MPI Tool Information Interface C Bindings	
		A.3.16 Deprecated C Bindings	
	A.4	Fortran 2008 Bindings with the mpi_f08 Module	
		A.4.1 Point-to-Point Communication Fortran 2008 Bindings	
		A.4.2 Partitioned Communication Fortran 2008 Bindings	
		A.4.3 Datatypes Fortran 2008 Bindings	
		A.4.4 Collective Communication Fortran 2008 Bindings	
		A.4.5 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindin	
		A.4.6 Process Topologies Fortran 2008 Bindings	
		A.4.7 MPI Environmental Management Fortran 2008 Bindings	
		A.4.8 The Info Object Fortran 2008 Bindings	
		A.4.9 Process Creation and Management Fortran 2008 Bindings	969
		A.4.10 One-Sided Communications Fortran 2008 Bindings	
		A.4.11 External Interfaces Fortran 2008 Bindings	980
		A.4.12 I/O Fortran 2008 Bindings	980
		A.4.13 Language Bindings Fortran 2008 Bindings	993
		A.4.14 Tools / Profiling Interface Fortran 2008 Bindings	994
		A.4.15 Deprecated Fortran 2008 Bindings	994
	A.5	Fortran Bindings with mpif.h or the mpi Module	995
		A.5.1 Point-to-Point Communication Fortran Bindings	995
		A.5.2 Partitioned Communication Fortran Bindings	998
		A.5.3 Datatypes Fortran Bindings	
		A.5.4 Collective Communication Fortran Bindings	
		A.5.5 Groups, Contexts, Communicators, and Caching Fortran Bindings $\ .$	
		A.5.6 Process Topologies Fortran Bindings	
		A.5.7 MPI Environmental Management Fortran Bindings	
		A.5.8 The Info Object Fortran Bindings	
		A.5.9 Process Creation and Management Fortran Bindings	
		A.5.10 One-Sided Communications Fortran Bindings	
		A.5.11 External Interfaces Fortran Bindings	
		A.5.12 I/O Fortran Bindings	
		A.5.13 Language Bindings Fortran Bindings	
		A.5.14 Tools / Profiling Interface Fortran Bindings	
		A.5.15 Deprecated Fortran Bindings	1029
В	Cha	inge-Log	1031
	B.1		1031
		B.1.1 Fixes to Errata in Previous Versions of MPI	
		B.1.2 Changes in MPI-4.0	1031
	B.2	Changes from Version 3.1 to Version 3.2	1032
		B.2.1 Fixes to Errata in Previous Versions of MPI	1032

	B.2.2 Changes in MPI-3.2	 1032
B.3	Changes from Version 3.0 to Version 3.1	 1034
	B.3.1 Fixes to Errata in Previous Versions of MPI	 1034
	B.3.2 Changes in MPI-3.1	 1036
B.4	Changes from Version 2.2 to Version 3.0	 1037
	B.4.1 Fixes to Errata in Previous Versions of MPI	 1037
	B.4.2 Changes in MPI-3.0	 1038
B.5	Changes from Version 2.1 to Version 2.2	 1042
B.6	Changes from Version 2.0 to Version 2.1	 1045
Bibliog	graphy d Index	1051 $1057$
Genera	u muex	1097
Examp	les Index	1063
MPI C	Constant and Predefined Handle Index	1066
MPI D	eclarations Index	1072
MPI C	Callback Function Prototype Index	1073
MPI F	unction Index	1074

# List of Figures

2	.1	State Transition Diagram for Blocking Operations	12
2	.2	State Transition Diagram for Nonblocking Operations	12
2	.3	State Transition Diagram for Persistent Operations	13
6	.1	Collective communications, an overview	185
6	.2	Inter-communicator allgather	188
6	.3	Inter-communicator reduce-scatter	189
6	.4	Gather example	196
6	.5	Gatherv example with strides	197
6	.6	Gatherv example, 2-dimensional	198
	.7	Gatherv example, 2-dimensional, subarrays with different sizes	199
	.8	Gatherv example, 2-dimensional, subarrays with different sizes and strides .	201
6	.9	Scatter example	206
6	.10	Scatterv example with strides	207
		Scatterv example with different strides and counts	208
		Race conditions with point-to-point and collective communications	299
6	.13	Overlapping Communicators Example	303
	.1	Inter-communicator creation using MPI_COMM_CREATE	327
	.2	Inter-communicator construction with MPI_COMM_SPLIT	331
	.3	Recursive communicator creation with MPI_COMM_SPLIT_TYPE	337
	.4	Three-group pipeline	356
7	.5	Three-group ring	357
8	.1	Neighborhood gather communication example.	412
8	.2	Cartesian neighborhood allgather example for 3 and 1 processes in a dimension	.413
8	.3	Cartesian neighborhood alltoall example for 3 and 1 processes in a dimension.	.417
8	.4	Set-up of process structure for two-dimensional parallel Poisson solver	439
8	.5	Communication routine with local data copying and sparse neighborhood all-to-all.	440
8	.6	Communication routine with sparse neighborhood all-to-all-w and without	110
Ŭ		local data copying.	441
8	.7	Two-dimensional parallel Poisson solver with persistent sparse neighborhood	
Ŭ		all-to-all-w and without local data copying.	442
1	1.1	Session handle to communicator	492
1	1.2	Process set examples	495

12.1	Schematic description of the public/private window operations in the	
	MPI_WIN_SEPARATE memory model for two overlapping windows	582
12.2	Active target communication	585
12.3	Active target communication, with weak synchronization	586
12.4	Passive target communication	587
12.5	Active target communication with several processes	591
12.6	Symmetric communication	610
12.7	Deadlock situation	610
12.8	No deadlock	611
	Etypes and filetypes	632
14.2	Partitioning a file among parallel processes	632
14.3	Displacements	645
14.4	Example array file layout	710
14.5	Example local array filetype for process 1	710
19.1	Status conversion routines	833

# List of Tables

2.1	Deprecated and Removed constructs	23
$3.1 \\ 3.2 \\ 3.3 \\ 3.4$	Predefined MPI datatypes corresponding to Fortran datatypes Predefined MPI datatypes corresponding to C datatypes Predefined MPI datatypes corresponding to both C and Fortran datatypes . Predefined MPI datatypes corresponding to C++ datatypes	31 32 33 33
5.1	combiner values returned from MPI_TYPE_GET_ENVELOPE	155
7.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode)	351
$9.1 \\ 9.2$	Error classes (Part 1)	463 464
11.1	List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization	504
	C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR	$556 \\ 600$
$14.2 \\ 14.3 \\ 14.4$	Data access routines	647 689 690 690 707
$15.2 \\ 15.3 \\ 15.4 \\ 15.4 \\ 15.5 \\ 15.6 \\$	<ul> <li>MPI tool information interface verbosity levels</li></ul>	721 721 724 728 748 749 768
17.2	<ul> <li>Removed MPI-1 functions and their replacements</li></ul>	775 776 776

1	17.4 Removed MPI-1 callback prototypes and their replacements	776
2 3 4	19.1 Specific Fortran procedure names and related calling conventions19.2 Occurrence of Fortran optimization problems	787 813
5		010
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		
-10		

## Acknowledgments

This document is the product of a number of distinct efforts in four distinct phases: one for each of MPI-1, MPI-2, MPI-3, and MPI-4. This section describes these in historical order, starting with MPI-1. Some efforts, particularly parts of MPI-2, had distinct groups of individuals associated with them, and these efforts are detailed separately.

This document represents the work of many people who have served on the MPI Forum. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has led to the MPI standard. The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Message-Passing Interface (MPI), many people helped with this effort. Those who served as primary coordinators in MPI-1.0 and MPI-1.1 are: • Jack Dongarra, David Walker, Conveners and Meeting Chairs • Ewing Lusk, Bob Knighten, Minutes • Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communication  $^{24}$ • Al Geist, Marc Snir, Steve Otto, Collective Communication • Steve Otto, Editor • Rolf Hempel, Process Topologies • Ewing Lusk, Language Binding • William Gropp, Environmental Management • James Cownie, Profiling • Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators • Steven Huss-Lederman, Initial Implementation Subset The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 process not mentioned above. 

xxiii

 $\overline{7}$ 

1	Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz
2	Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster
$\frac{3}{4}$	Jim Feeney Daniel Frye	Vince Fernando Ian Glendinning	Sam Fineberg Adam Greenberg	Jon Flower Robert Harrison
4 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			1 NT	
14	-		-	atory made the draft avail-
15				distributing the document. RPA and NSF under grant
16		-		nology Center Cooperative
17 18				opean Community through
19	Esprit project P6643 (Pl			
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 26	• Ewing Lusk, Conve	ener and Meeting Cl	nair	
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-Sid	led Communications	3	
33	• Bill Gropp and An	thony Skjellum, Ext	ended Collective O	perations
34 35	• Steve Huss-Lederm	an, External Interfa	ices	
36	• Bill Nitzberg, I/O			
37 38	• Andrew Lumsdaine	e, Bill Saphir, and J	eff Squyres, Langua	ge Bindings
39 40	• Anthony Skjellum	and Arkady Kaneys	ky. Real-Time	
40				
42	The following list in	cludes some of the a	active participants w	who attended MPI-2 Forum
43	meetings and are not me	entioned above.		
44				
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				29
				30
The MPI Forum a	also acknowledges and ap	preciates the valuable	e input from people via	31

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

The following institutions supported the MPI-2 effort through time and travel support for the people listed above.

Argonne National Laboratory	35
Bolt, Beranek, and Newman	36
California Institute of Technology	37
Center for Computing Sciences	38
Convex Computer Corporation	39
Cray Research	40
Digital Equipment Corporation	41
Dolphin Interconnect Solutions, Inc.	42
Edinburgh Parallel Computing Centre	43
General Electric Company	44
German National Research Center for Information Technology	45
Hewlett-Packard	46
Hitachi	47
Hughes Aircraft Company	48

32

33

34

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 Adminstration
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
	8 9
• Bill Gropp, Steering Committee	10
• Rainer Keller, Merge of MPI-1.3	11
• Andrew Lumsdaine, Steering Committee	12 13
• Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)	14 15
• Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)	16 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 36
• Rajeev Thakur, I/O	37
• Jeffrey M. Squyres, Language Bindings and MPI-2.1 Secretary	38 39
• Rolf Rabenseifner, Deprecated Functions and Annex Change-Log	40
• Alexander Supalov and Denis Nagorny, Annex Language Bindings	41 42
The following list includes some of the active participants who attended MPI-2 Forum	43
meetings and in the e-mail discussions of the errata items and are not mentioned above.	44 45
	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	I		
17	The MPI Forum also ack	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	Lawrence Livermore Nat		
33 34	Los Alamos National Lab	poratory	
	Mathworks		
35 36	Mellanox Technologies		
37	Microsoft		
38	Myricom	NEC Emer Ltd	
39	NEC Laboratories Europ	, -	
40	Oak Ridge National Labo		
41	The Ohio State Universit		
42	Pacific Northwest Nation	al Laboratory	
43	QLogic Corporation Sandia National Laborat	aniag	
44	SiCortex	ones	
45	Silicon Graphics Incorpor	rated	
46	Sun Microsystems, Inc.	laucu	
47	University of Alabama at	Birmingham	
48	University of Houston	Diffinition	
	Oniversity 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
• William Gropp, Front matter, Introduction, and Bibliography; MPI-2.2 chair.	14
• Richard Graham, Point-to-Point Communication and Datatypes	15 16
Adam Moody, Collective Communication	17
• Torsten Hoefler, Collective Communication and Process Topologies	18 19
<ul> <li>Richard Treumann, Groups, Contexts, and Communicators</li> </ul>	20 21
	21
• Jesper Larsson Träff, Process Topologies, Info-Object and One-Sided Communications	23 24
• 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
• Jeffrey M. Squyres, Language Bindings and MPI-2.2 Secretary	30 31
	32
• Rolf Rabenseifner, Deprecated Functions, Annex Change-Log, and Annex Language Bindings	33
	34
• Alexander Supalov, Annex Language Bindings	35 36
The following list includes some of the active participants who attended MPI-2 Forum	37
meetings and in the e-mail discussions of the errata items and are not mentioned above.	38
	39
	40
	41 42
	42
	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
20	Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu
21	Ron Vandevaart	Adminav Visiniu	Weikuan Tu
23	The MPI Forum also acknow	vledges and appreciates the va	luable input from people via
24	e-mail and in person.		
25	The following institutions su	pported the MPI-2.2 effort three	ough time and travel support
26	for the people listed above.		
27	Argonne National Laborato	rv	
28	Auburn University		
29	Bull		
30	Cisco Systems, Inc.		
31	Cray Inc.		
32	Forschungszentrum Jülich		
33	Fujitsu		
34	The HDF Group		
35	Hewlett-Packard	v	
36	International Business Mach	nines	
37	Indiana University		
38	Institut National de Rechero	che en Informatique et Autom	atique (Inria)
39		nce & Engineering Corporation	
40	Intel Corporation		
41	Lawrence Berkeley National	Laboratory	
42	Lawrence Livermore Nation		
43	Los Alamos National Labor	Ū.	
44	Mathworks	-	
45	Mellanox Technologies		
46	Microsoft		
47	Myricom		
48	NEC Corporation		

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

Unofficial Draft for Comment Only

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

3			
4	Tatsuya Abe	Tomoya Adachi	Sadaf Alam
5	Reinhold Bader	Pavan Balaji	Purushotham V. Bangalore
6	Brian Barrett	<b>Richard Barrett</b>	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			
37	The MPI Forum also acknowled	dges and appreciates the	e valuable input from people via

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

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

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

Argonne National Laboratory

<sup>45</sup> Bull

38

44

3

- <sup>46</sup> Cisco Systems, Inc.
- <sup>47</sup> Cray 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
Intel Corporation	11
Lawrence Berkeley National Laboratory	12
Lawrence Livermore National Laboratory	13
Los Alamos National Laboratory	14
Mellanox Technologies, Inc.	15
Microsoft Corporation	16
NEC Corporation	17
National Oceanic and Atmospheric Administration, Global Systems Division	18
NVIDIA Corporation	19
Oak Ridge National Laboratory	20
The Ohio State University	21
Oracle America	22
Platform Computing	23
RIKEN AICS	24
RunTime Computing Solutions, LLC	25
Sandia National Laboratories	26
Technical University of Chemnitz	27
Tokyo Institute of Technology	28
University of Alabama at Birmingham	29
University of Chicago	30
University of Houston	31
University of Illinois at Urbana-Champaign	32
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	33
University of Tennessee, Knoxville	34
University of Tokyo	35
Funding for the MPI Forum meetings was partially supported by awards $\#$ CCF-0816909	36
and $\#CCF-1144042$ from the National Science Foundation. In addition, the HDF Group	37
and $\frac{1}{\pi}$ correct fitting hold the reaction sectore roundation. In addition, the first order and Sandia National Laboratories provided travel support for one U.S. academic each.	38
and Sandia Rational Laboratories provided traver support for one e.s. academic cach.	39
MPI-3.1:	40
IVIF I-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
	44
• Martin Schulz, MPI-3.1 chair	45
• William Gropp, Steering committee, Front matter, Introduction, One-Sided Commu-	46
nications, and Bibliography; Overall editor	47
	48

### Unofficial Draft for Comment Only

1 2	,	0	nd Definitions, and Fortran Bindings, 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, Datatype	es and Environmental M	Management
10	• Torsten Hoefler, Collectiv	ve Communication and	Process Topologies
11 12	• Pavan Balaji, Groups, C	ontexts, and Communic	cators, and External Interfaces
13 14	• Jeff Hammond, The Info	Object	
15	• David Solt, Process Crea	·	
16 17	• Quincey Koziol, I/O	J	
18 19	• Kathryn Mohror, Tool S	upport	
20	• Rajeev Thakur, One-Side		
21 22 23 24	The following list include meetings or participated in the		articipants who attended MPI Forum
25	Charles Archer	Pavan Balaji	Purushotham V. Bangalore
26	Brian Barrett	Wesley Bland	Michael Blocksome
27	George Bosilca	Aurelien Bouteiller	Devendar Bureddy
28	Yohann Burette	Mohamad Chaarawi	Alexey Cheptsov
29	James Dinan	Dmitry Durnov	Thomas Francois
30	Edgar Gabriel	Todd Gamblin	Balazs Gerofi
31	Paddy Gillies	David Goodell	Manjunath Gorentla Venkata
32	Richard L. Graham	Ryan E. Grant	William Gropp
33	Khaled Hamidouche	Jeff Hammond	Amin Hassani
34	Marc-André Hermanns	Nathan Hjelm	Torsten Hoefler
35	Daniel Holmes	Atsushi Hori	Yutaka Ishikawa
36	Hideyuki Jitsumoto	Jithin Jose	Krishna Kandalla
37	Christos Kavouklis	Takahiro Kawashima	Chulho Kim
38	Michael Knobloch	Alice Koniges	Quincey Koziol
39	Sameer Kumar	Joshua Ladd	Ignacio Laguna
40	Huiwei Lu	Guillaume Mercier	Kathryn Mohror
41	Adam Moody	Tomotake Nakamura	Takeshi Nanri
42	Steve Oyanagi	Antonio J. Pẽna	Sreeram Potluri
43	Howard Pritchard	Rolf Rabenseifner	Nicholas Radcliffe
44	Ken Raffenetti	Raghunath Raja	Craig Rasmussen
45	Davide Rossetti	Kento Sato	Martin Schulz
46	Sangmin Seo	Christian Siebert	Anthony Skjellum
47			
- I	Brian Smith	David Solt	Jeffrey M. Squyres

	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	Huseyin 5. Thuiz	Julienao Zhang	7
				8
The MF	PI Forum also acknowled	ges and appreciates	the valuable input from people via	9
e-mail and i	n person.			10
The foll	owing institutions suppo	rted the $MPI-3.1$ effe	rt through time and travel support	11
for the peop	le listed above.			12
Argonn	e National Laboratory			13
-	University			14
	ystems, Inc.			15
Cray	ystems, me.			16
v	The University of Edinb	urgh		17
ETH Z	-	Jurgin		18
	ngszentrum Jülich			19
Fujitsu	ingozonici ani o anon			20
•	n Research School for Sir	nulation Sciences		21
	OF Group			22
	tional Business Machines	3		23
Institut	National de Recherche	en Informatique et A	Automatique (Inria)	24
	orporation			25
	-	e, High-Performance	Computing (JARA-HPC)	26
	University	, .		27
Lawren	ce Berkeley National La	boratory		28
Lawren	ce Livermore National L	aboratory		29
Lenovo				30
Los Ala	mos National Laborator	у		31
Melland	ox Technologies, Inc.			32
Microso	oft Corporation			33
NEC C	orporation			34
NVIDL	A Corporation			35
Oak Ri	dge National Laboratory			36
	io State University			37
RIKEN				38
	National Laboratories			39
	Advanced Computing Ce	nter		40
-	Institute of Technology			41
	ity of Alabama at Birmi	ngham		42
	ity of Houston			43
	ity of Illinois at Urbana-	Champaign		44
	ity of Oregon	a		45
			ng Center Stuttgart (HLRS)	46
	ity of Tennessee, Knoxvi	ille		47
Univers	ity of Tokyo			48

1	MPI-4.0:
2 3 4	MPI-4.0 is a major update to the MPI Standard. The editors and organizers of the MPI-4.0 have been:
5	• Martin Schulz, MPI-4.0 chair, Info Object, External Interfaces
6 7	• Wesley Bland, MPI-4.0 Secretary, Backward Incompatibilities
8 9 10	William Gropp, MPI-4.0 Editor, Steering committee, Front matter, Introduction, One- Sided Communications, and Bibliography
11 12	• Rolf Rabenseifner, Steering committee, Process Topologies, Deprecated Functions, Removed Interfaces, Annex Language Bindings Summary, and Annex Change-Log.
13 14	• Puri Bangalore, Language Bindings
15 16	• Claudia Blaas-Schenner, Terms and Conventions
17	• George Bosilca, Datatypes and Environmental Management
18 19	• Ryan Grant, Partitioned Communication
20 21	• Marc-André Hermanns, Tool Support
22	• Daniel Holmes, Point-to-Point Communication, Sessions
23 24	• Guillaume Mercier, Groups, Contexts, Communicators, Caching
25 26	• Howard Pritchard, Process Creation and Management
27	• Anthony Skjellum, Collective Communication, I/O
28 29 30 31	As part of the development of MPI-4.0, a number of working groups were established. In some cases, the work for these groups overlapped with multiple chapters. The following describes the major working groups and the leaders of those groups:
32 33 34	Collective Communication, Topology, Communicators Torsten Hoefler, Andrew Lumsdaine, and Anthony Skjellum
35	Fault Tolerance Wesley Bland, Aurélien Bouteiller, and Richard Graham
36 37	Hardware-Topologies Guillaume Mercier
38	Hybrid & Accelerator Pavan Balaji and Jim Dinan
39 40	Large Counts Jeff Hammond
41 42	Persistence Anthony Skjellum
43	Point to Point Communication Richard Graham and Dan Holmes
44 45	Remote Memory Access William Gropp and Rajeev Thakur
46 47	Semantic Terms Rolf Rabenseifner and Purushotham Bangalore
48	Sessions Daniel Holmes

#### Tools Kathryn Mohror and Marc-André Hermanns

The following list includes some of the active participants who attended MPI Forum meetings or participated in the e-mail discussions.

Julien Adam	Abdelhalim Amer	Charles Archer	6 7
Ammar Ahmad Awan	Pavan Balaji	Marc Gamell Balmana	8
Purushotham Bangalore	Mohammadreza Bayatpour	Jean-Baptiste Besnard	9
Claudia Blaas-Schenner	Wesley Bland	Gil Bloch	10
George Bosilca	Aurelien Bouteiller	Ben Bratu	10
Alexander Calvert	Nicholas Chaimov	Sourav Chakraborty	12
Steffen Christgau	Ching-Hsiang Chu	Mikhail Chuvelev	12
James Clark	Carsten Clauss	Giuseppe Congiu	
Brandon Cook	James Custer	Anna Daly	14
Hoang-Vu Dang	James Dinan	Matthew Dosanjh	15
Murali Emani	Christian Engelmann	Noah Evans	16
Ana Gainaru	Esthela Gallardo	Balazs Gerofi	17
Salvatore Di Girolamo	Brice Goglin	Richard Graham	18
Ryan Grant	Stanley Graves	William Gropp	19
Siegmar Gross	Taylor Groves	Yanfei Guo	20
Khaled Hamidouche	Jeff Hammond	Marc-André Hermanns	21
Nathan Hjelm	Torsten Hoefler	Daniel Holmes	22
Atsushi Hori	Josh Hursey	Ilya Ivanov	23
Julien Jaeger	Emmanuel Jeannot	Sylvain Jeaugey	24
Jithin Jose	Krishna Kandalla	Takahiro Kawashima	25
Chulho Kim	Michael Knobloch	Alice Koniges	26
Sameer Kumar	Kim Kyunghun	Ignacio Laguna	27
Stefan Lankes	Tonglin Li	Xioyi Lu	28
Kavitha Madhu	Alexey Malhanov	Ryan Marshall	29
William Marts	Guillaume Mercier	Kathryn Mohror	30
Takeshi Nanri	Thomas Naughton	Takafumi Nose	31
Lena Oden	Steve Oyanagi	Guillaume Papauré	32
Ivy Peng	Ignacio Laguna Peralta	Antonio Peña	33
Simon Pickartz	Artem Polyakov	Sreeram Potluri	34
Howard Pritchard	Martina Prugger	Marc Pérache	35
Rolf Rabenseifner	Nicholas Radcliffe	Ken Raffenetti	36
Craig Rasmussen	Soren Rasmussen	Hubert Ritzdorf	37
Sergio Rivas-Gomez	Davide Rossetti	Martin Ruefenacht	38
Amit Ruhela	Joseph Schuchart	Martin Schulz	39
Sangmin Seo	Sameh Sharkawi	Sameer Shende	40
Min Si	Anthony Skjellum	Brian Smith	41
David Solt	Jeff Squyres	Srinivas Sridharan	42
Hari Subramoni	Nawrin Sultana	Shinji Sumimoto	43
Sayantan Sur	Hugo Taboada	Keita Teranishi	44
Rajeev Thakur	Keith Underwood	Isaias Alberto Compres Urena	45
Geoffroy Vallee	Manjunath Gorentla Venkata	Akshay Venkatesh	46
	J		47

48

1

 $^{2}$ 

3

 $\frac{4}{5}$ 

1	Jerome Vienne Anh Vo Justin Wozniak
2	Junchao Zhang Dong Zhong Hui Zhou
3	
4	
5	The MPI Ferry also admovided and appreciates the valuable input from people via
6	The MPI Forum also acknowledges and appreciates the valuable input from people via
7	e-mail and in person.
8	The following institutions supported the MPI-4.0 effort through time and travel support for the people listed above.
9	tor the people listed above.
10	ATOS/BULL
11	Argonne National Laboratory
12	Arm
13	Auburn University
14	Barcelona Supercomputing Center
15	CEA
16	Cisco Systems Inc.
17 18	Cray Inc.
19	EPCC, The University Of Edinburgh
20	ETH Zürich
20	Forschungszentrum Jülich
22	Fujitsu Fulda University of Applied Sciences
23	German Research School for Simulation Sciences
24	Hewlett Packard Enterprise
25	International Business Machines
26	Institut National de Recherche en Informatique et Automatique (Inria)
27	Intel Corporation
28	Jülich Aachen Research Alliance, High-Performance Computing (JARA-HPC)
29	Jülich Supercomputing Center
30	KTH Royal Institute Of Technology
31	Kyushu University
32	Lawrence Berkeley National Laboratory
33	Lawrence Livermore National Laboratory
34	Lenovo
35	Los Alamos National Laboratory
36	Mellanox Technologies, Inc.
37	Microsoft Corporation
38	NEC Corporation
39	NVIDIA Corporation
40	Oak Ridge National Laboratory
41	PAR-TEC
42	Paratools, Inc.
43	RIKEN AICS (R-CCS as of 2017)
44	RWTH Aachen University
45 46	Rutgers University
40 47	Sandia National Laboratories
48	Silicon Graphics, Inc.
10	VSC Research Center, TU Wien

Technical University Of Munich	1
The HDF Group	2
The Ohio State University	3
Texas Advanced Computing Center	4
Tokyo Institute of Technology	5
University of Alabama at Birmingham	6
University of Houston	7
University of Illinois at Urbana-Champaign and the National Center for Supercompu	ıt- <sup>8</sup>
ing Applications	9
University of Innsbruck	10
University of Oregon	11
University of Pottsdam	12
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	13
University of Tennessee, Chattanooga	14
University of Tennessee, Knoxville	15
University of Texas at El Paso	16
University of Tokyo	17
	18
	19
	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30
	31 32
	32
	34
	34
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

# Chapter 1

# Introduction to MPI

### 1.1 Overview and Goals

MPI (Message-Passing Interface) is a *message-passing library interface specification*. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O. MPI is a *specification*, not an implementation; there are multiple implementations of MPI. This specification is for a *library interface*; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C 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 [2, 3], Intel's NX/2 [57], Express [14], nCUBE's Vertex [53], p4 [9, 10], and PARMACS [6, 11]. Other important contributions have come from Zipcode [60, 61], Chimp [20, 21], PVM [5, 18], Chameleon [31], and PICL [26].

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 69. At this work- $^{24}$ shop the basic features essential to a standard message-passing interface were discussed, 25and a working 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 [19]. 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

43

45

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 [23]. The first product of these deliberations

#### Unofficial Draft for Comment Only

7 8

9 10

11 12

13

14

15

16

17

was Version 1.1 of the MPI specification, released in June of 1995 [24] (see http://www.mpi-forum.org for official MPI document releases). At that time, effort focused in five areas.

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

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

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

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

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

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

After the release of MPI-2.0, the MPI Forum kept working on errata and clarifications for42both standard documents (MPI-1.1 and MPI-2.0). The short document "Errata for MPI-1.1"43was released October 12, 1998. On July 5, 2001, a first ballot of errata and clarifications for44MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done45electronically. Both ballots were combined into one document: "Errata for MPI-2," May4615, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors47kept working on new requests for clarification.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 35

36

37

38 39

40

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.

1516

1718

19

20

2122

23

 $^{24}$ 25

26

2728

29

30  $^{31}$ 

32 33

34

35

36

37

#### Background of MPI-2.2 1.5

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.

#### 1.6Background 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

#### Background of MPI-3.1 1.7

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

47

48

# 1.8 Background of 2020 Draft Specification

The 2020 draft specification is expected to become the MPI-4.0 specification once all features have been merged. MPI-4.0 is a major update to the MPI standard. This update includes a number of new features which will be present in the final MPI-4.0 document. The largest changes are then addition of persistent collectives, partitioned communications, application info assertions, and improvements to the definitions of error handling. In addition, there are a number of smaller improvements and corrections.

# 1.9 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.

# 1.10 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.

## 1.11 What Is Included in the Standard?

The standard includes:

- Point-to-point communication,
- Datatypes,
- Collective operations,

 $^{24}$ 

1	• Process groups,
2 3	• Communication contexts,
4	• Process topologies,
5 6	• Environmental management and inquiry,
7 8	• The Info object,
9	• Process creation and management,
10 11	• One-sided communication,
12 13	• External interfaces,
14	• Parallel file I/O,
15 16	• Language bindings for Fortran and C,
17 18	• Tool support.
19	
20	1.12 What Is Not Included in the Standard?
21 22	The standard does not specify:
23 24 25	• Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,
26	• Program construction tools,
27 28	• Debugging facilities.
29 30 31 32 33 34 35	There are many features that have been considered and not included in this standard. This happened for a number of reasons, one of which is the time constraint that was self- imposed in finishing the standard. Features that are not included can always be offered as extensions by specific implementations. Perhaps future versions of MPI will address some of these issues.
36	1.13 Organization of This Document
37 38 39	The following is a list of the remaining chapters in this document, along with a brief description of each.
40 41 42	• Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
43 44 45 46	• Chapter 3, Point-to-Point Communication, defines the basic, pairwise communication subset of MPI. <i>Send</i> and <i>receive</i> are found here, along with many associated functions designed to make basic communication powerful and efficient.
47 48	• Chapter 5, 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.

# Unofficial Draft for Comment Only

- Chapter 6, 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 inter-communicators. It also adds two new collective operations. MPI-3 adds nonblocking collective operations.
- Chapter 7, Groups, Contexts, Communicators, and Caching, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*.
- Chapter 8, 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 9, 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 10, The Info Object, defines an opaque object, that is used as input in several MPI routines.
- Chapter 11, Process Initialization, Creation, and Management, defines routines that allow for creation of processes.
- Chapter 12, 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 13, 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 14, I/O, defines MPI support for parallel I/O.
- Chapter 15, 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 15.2 (Profiling Interface), which was a chapter in previous versions of MPI.
- Chapter 16, 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 17, Removed Interfaces, describes routines and constructs that have been removed from MPI. Some of these were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard. Others of these were deprecated in MPI-3, and the MPI Forum decided to remove these from the MPI-4 standard.
- Chapter 18, Backward Incompatibilities, describes incompatibilities with previous versions of MPI.

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 $45 \\ 46$ 

1 • Chapter 19, Language Bindings, discusses Fortran issues, and describes language in- $\mathbf{2}$ teroperability aspects between C and Fortran. 3 The Appendices are: 4 5• Annex A, Language Bindings Summary, gives specific syntax in C and Fortran, for 6 all MPI functions, constants, and types. 7 8 • Annex B, Change-Log, summarizes some changes since the previous version of the 9 standard. 10 11 • Several Index pages show the locations of examples, constants and predefined han-12dles, declarations of C and Fortran types, callback routine prototypes, and all MPI 13 functions. 1415MPI provides various interfaces to facilitate interoperability of distinct MPI imple-16mentations. Among these are the canonical data representation for MPI I/O and for 17MPI\_PACK\_EXTERNAL and MPI\_UNPACK\_EXTERNAL. The definition of an actual bind-18 ing of these interfaces that will enable interoperability is outside the scope of this document. 19A 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. 2021They are part of the "Journal of Development" (JOD), lest good ideas be lost and in order 22 to provide a starting point for further work. The chapters in the JOD are 23• Chapter 2, Spawning Independent Processes, includes some elements of dynamic pro- $^{24}$ cess management, in particular management of processes with which the spawning 2526processes do not intend to communicate, that the Forum discussed at length but ultimately decided not to include in the MPI Standard. 2728• Chapter 3, Threads and MPI, describes some of the expected interaction between an 29 MPI implementation and a thread library in a multithreaded environment. 30  $^{31}$ • Chapter 4, Communicator ID, describes an approach to providing identifiers for com-32 municators. 33 • Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particu-34 lar single-copy routines for use in shared-memory environments and new datatype 35 constructors. 36 37 • Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a 38 more elaborate Fortran 90 interface. 39 40 • Chapter 7, Split Collective Communication, describes a specification for certain non-41 blocking collective operations. 42• Chapter 8, Real-Time MPI, discusses MPI support for real time processing. 43 44 4546 47 48

# Chapter 2

# **MPI** Terms and Conventions

This chapter explains notational terms and conventions used throughout the MPI document, some of the choices that have been made, and the rationale behind those choices.

# 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.

MPI identifiers are limited to 32 characters (35 with the profiling and large count interfaces, and Fortran specific procedure names upto 41). 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.

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

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

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

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

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

48

1

2

3

4

5

6

7 8

9

10

11

13

16

17

18 19

20 21

22

23

 $^{24}$ 

25 26

27 28

29

30

31 32

33

34

35

36

37

Done

#-other

#-error

#-335

2.2. p10

**#-358+359**<sub>12</sub>

18.1 p.777

19.1.5 p789

B.1.2 p.103 1ff

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

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

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

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

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

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

An exception is Section 15.3 "The MPI Tool Information Interface", which only provides ISO C interfaces.

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

The words function, routine, procedure, procedure call, and call are often used as 23 synonyms within this standard Some MPI procedures have two interfaces for a given language support (see Sections 2.5.6 and 2.5.8).

# 2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used. The term **message** data buffer refers to the send/receive buffer used in a communication procedure. The term file data buffer refers to the data buffers used by MPI I/O procedures. In this section we use the term data buffer and depending on the MPI procedure it will refer to message data buffer or file data buffer.

# 2.4.1 MPI Operations

- **MPI operation** An MPI operation is a sequence of steps performed by the MPI library to establish and enable data transfer and/or synchronization. It consists of four stages: initialization, starting, completion, and freeing, and it is implemented as a set of one or more MPI procedures, see Section 2.4.2.
  - **Initialization** hands over the argument list to the operation but not the content of the data buffers, if any. The specification of an operation may state that array arguments must not be changed until the operation is freed.
  - **Starting** hands over the control of the data buffers, if any, to the associated operation.

Note that **initiation** refers to the combination of the initialization and starting stages.

2

10 11

12 13

14

15

16

17

18

<sup>19</sup> in

20

21

 $^{22}$ 

27

28

29

30

31

32 33 34

35

36

37

38

39 40

41

42

43

44

 $45 \\ 46$ 

47

48

#-other

#-error!

Done

PR437

	12 CHAPTER 2. MPI TERMS AND CONVENTIONS
1 2 3 4 5 6 7	<ul> <li>Completion returns control of the content of the data buffers to the application and indicates that output buffers and arguments, if any, have been updated. Note that an MPI operation is complete when the MPI procedure implementing the completion stage returns.</li> <li>Freeing returns control of the rest of the argument list (e.g., the data buffer address and array arguments).</li> </ul>
8 9	MPI operations are available in one or more of these forms: blocking, nonblocking, and persistent.
10 11 12	<b>Blocking operation</b> For a <b>blocking operation</b> , all four stages are combined in a single procedure call (as shown in Figure 2.1 and defined in Section 2.4.2).
13 14 15 16 17	Initialization & Starting Completion & Freeing Figure 2.1: State Transition Diagram for Blocking Operations
<ol> <li>18</li> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> </ol>	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 blocking or nonblocking (as shown in Figure 2.2 and defined in Section 2.4.2).
24 25 26 27 28 29 30	Figure 2.2: State Transition Diagram for Nonblocking Operations
31 32 33 34	<b>Persistent operation</b> For a <b>persistent operation</b> , there is a separate procedure for each of the four stages (as shown in Figure 2.3 and defined in Section 2.4.2). Each of these procedures may be blocking or nonblocking.
35 36 37 38 39 40	For a partitioned send operation, an additional call to activate each partition of the send buffer (see Section 4.2.1) is required to finish the starting stage. For a partitioned receive operation, before the operation is complete the user is allowed to access a partition of the output buffer after verifying that it has arrived (see Section 4.2.2). Additionally, an MPI operation can be collective or noncollective.
41 42 43 44 45 46	<ul> <li>Collective operation Collective operations are defined as operations that involve a group or groups of MPI processes. For collective operations the completion stage may or may not finish before all processes in the group have started the operation.</li> <li>Collective MPI operations are also available as blocking, nonblocking, or persistent operations.</li> </ul>
47 48	<b>Noncollective operation</b> Noncollective operations are defined as operations that are not collective.



Figure 2.3: State Transition Diagram for Persistent Operations

### 2.4.2 MPI Procedures

All MPI procedures can either be local or non-local - defined as follows:

Non-local procedure An MPI procedure is non-local if returning may require, during its execution, some specific semantically-related MPI procedure to be called on another MPI process.

Local procedure An MPI procedure is local if it is not non-local.

An MPI operation is implemented as a set of one or more MPI procedures. An MPI **operation-related procedure** implements at least a part of a stage of an MPI operation as described in Section 2.4.1. An MPI operation-related procedure may also implement one or more stages of one or several MPI operations. In certain cases, more than one MPI operation-related procedure may be needed to implement a single stage.

There are also other MPI procedures that do not implement any stage of any MPI operation.

The semantics of MPI operation-related procedures are described using two orthogonal (independent) concepts: completeness (depends on which stages are included) and locality. Such procedures can be either incomplete, or completing, or freeing, or completing and freeing based on the status of the associated operation at the time the procedure returns. Also, all such procedures can be described as either blocking or nonblocking, but these latter two terms refer to combinations of the completeness and locality concepts. Additionally, all MPI operation-related procedures can be collective or noncollective.

The following are properties of MPI operation-related procedures:

- **Initialization procedure** An MPI procedure is an **initialization procedure** if return from the procedure indicates that the associated operation has completed its initialization stage, which implies that the user has handed over control of the argument list (but not contents of the data buffers) to MPI. The user is still allowed to read or modify the contents of the data buffers. If an initializing procedure is not also the freeing procedure of the associated operation (see below) then the user is not permitted to deallocate the data buffers or to modify the array arguments.
- **Starting procedure** An MPI procedure is a **starting procedure** if return from the procedure indicates that the associated operation has completed its starting stage, which implies that the user has handed over control of the data buffers to MPI. If a starting procedure is not also a completing procedure of the associated operation (see below) then the user is not permitted to modify input data buffers or to read output data buffers.

 $^{24}$ 

1 2 3	<b>Initiation procedure</b> An MPI procedure is an <b>initiation procedure</b> if return procedure indicates that both the initialization and the starting stage have co which implies control of the entire argument list is handed over to MPI.				
4 5 6 7 8 8 8	<b>Completing procedure</b> An MPI procedure is called <b>completing</b> if return from cedure indicates that at least one associated operation has finished its constage, which implies that the user can rely on the content of the output dat and modify the content of input and output data buffers of such operation completing procedure is not also a freeing procedure (see below) then the upermitted to deallocate the data buffers or to modify the array arguments.	ompletion ta buffers n(s). If a			
1	Incomplete procedure An MPI procedure is called incomplete if it is not a construction procedure.	ompleting			
13 14 15 16 17 18	<ul> <li>Freeing procedure An MPI procedure is freeing if return from the procedure indicates that at least one associated operation has finished its freeing stage, which implies that the user can reuse all parameters specified when initializing such associated operation(s).</li> </ul>				
1:		and iocai.			
2 2: 2: 2: 2: 2: 2: 3: 3: 3: 3: 3: 3: 3: 3:	<ul> <li>Advice to users. Note that for operation-related MPI procedures, in mincomplete procedures are local and completing procedures are non-local. E are noted where such procedures are defined. In many cases an additional procedures in the procedure name incomplete and immediate marks no procedures in the procedure name.</li> <li>Some categorization examples are listed below.</li> <li>Nonblocking procedures:</li> <li>incomplete and local: MPI_ISEND, MPI_IRECV, MPI_IBCAST, MPI_INMPI_SEND_INIT, MPI_RECV_INIT,</li> <li>Blocking procedures:</li> </ul>	xceptions efix letter nblocking			
3 #-error #-update6 3 3 3 4 2 4 4 4 4 4 4 4 4 4 4 4 4 4	<ul> <li>completing and non-local: MPI_SEND, MPI_RECV, MPI_BCAST,</li> <li>incomplete and non-local: MPI_MPROBE, MPI_BCAST_INIT,, MPI_FILE_{READ WRITE}_{AT_ALL ALL ORDERED}_BEGIN.</li> <li>completing and local: MPI_BSEND, MPI_RSEND, MPI_MRECV.</li> <li>MPI procedures that are not MPI operation-related:</li> <li>MPI_COMM_RANK, MPI_WTIME, MPI_PROBE, MPI_IPROBE,</li> <li>(End of advice to users.)</li> </ul> Collective procedure An MPI procedure is collective if all processes in a group	The File routines end in the Function Index with _FN instead of _Begin -> to be fixed			
4 4 4	Initialization procedures of collective operations over the same process group	p must be			

# Unofficial Draft for Comment Only

An MPI collective procedure is **synchronizing** if it will only return once all processes in the associated group or groups of MPI processes have called the appropriate matching MPI procedure.

The initiation procedures for nonblocking collective operations and the starting procedures for persistent collective operations are local and shall not be synchronizing.

All other procedures for collective operations, such as for blocking collective operations and the initialization procedures for persistent collective operations, may or may not be synchronizing.

Calling any synchronizing function is erroneous when there is no Advice to users. possibility of corresponding calls at all other processes in the associated process group.

Waiting for completion of any collective operation is erroneous when there is no possibility that all other processes in the associated group will be able to start the corresponding operation. (End of advice to users.)

#### 2.4.3 **MPI** Datatypes

For datatypes, the following terms are defined:

**predefined** A predefined datatype is a datatype with a predefined (constant) name (such as MPI\_INT, MPI\_FLOAT\_INT, or MPI\_PACKED) or a datatype constructed with MPI\_TYPE\_CREATE\_F90\_INTEGER, MPI\_TYPE\_CREATE\_F90\_REAL, or MPI\_TYPE\_CREATE\_F90\_COMPLEX. The former are named whereas the latter are unnamed.

**derived** A derived datatype is any datatype that is not predefined.

- portable A datatype is portable if it is a predefined datatype, or it is derived from 28 a portable datatype using only the type constructors MPI\_TYPE\_CONTIGUOUS, 29MPI\_TYPE\_VECTOR, MPI\_TYPE\_INDEXED, 30 MPI\_TYPE\_CREATE\_INDEXED\_BLOCK, MPI\_TYPE\_CREATE\_SUBARRAY, MPI\_TYPE\_DUP, and MPI\_TYPE\_CREATE\_DARRAY. Such a datatype is portable 32 because all displacements in the datatype are in terms of extents of one predefined 33 datatype. Therefore, if such a datatype fits a data layout in one memory, it will 34 fit the corresponding data layout in another memory, if the same declarations were 35 used, even if the two systems have different architectures. On the other hand, if a 36 datatype was constructed using MPI\_TYPE\_CREATE\_HINDEXED, 37 MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK, MPI\_TYPE\_CREATE\_HVECTOR or 38 MPI\_TYPE\_CREATE\_STRUCT, then the datatype contains explicit byte displace-39 ments (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 41 used for data layouts on another process, running on a processor with a different 42architecture. 44
- 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 3

1

4

5

6

 $\overline{7}$ 

8

9 10

11

12

13

14

151617

18

19 20

21

22

23

24

2526

27

 $^{31}$ 

40

43

45

46

4748

# 2.5 Data Types

### 2.5.1 Opaque Objects

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

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

## <sup>20</sup> TYPE, BIND(C) :: MPI\_Comm

- INTEGER :: MPI\_VAL
- <sup>22</sup> END TYPE MPI\_Comm

The C types must support the use of the assignment and equality operators.

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

Rationale. Since the Fortran integer values are equivalent, applications can easily convert MPI handles between all three supported Fortran methods. For example, an integer communicator handle COMM can be converted directly into an exactly equivalent mpi\_f08 communicator handle named comm\_f08 by comm\_f08%MPI\_VAL=COMM, and vice versa. The use of the INTEGER defined handles and the BIND(C) derived type handles is different: Fortran 2003 (and later) define that BIND(C) derived types can be used within user defined common blocks, but it is up to the rules of the companion C compiler how many numerical storage units are used for these BIND(C) derived type handles. Most compilers use one unit for both, the INTEGER handles and the handles defined as BIND(C) derived types. (End of rationale.)

Advice to users. If a user wants to substitute mpif.h or the mpi module by the mpi\_f08 module and the application program stores a handle in a Fortran common block then it is necessary to change the Fortran support method in all application routines that use this common block, because the number of numerical storage units of such a handle can be different in the two modules. (End of advice to users.)

<sup>46</sup> Opaque objects are allocated and deallocated by calls that are specific to each object
 <sup>47</sup> type. These are listed in the sections where the objects are described. The calls accept a
 <sup>48</sup> handle argument of matching type. In an allocate call this is an OUT argument that returns

### Unofficial Draft for Comment Only

3 4

5

6

7

8

9

10

19

21

23

24 25

26

27

28 29

30

31

32

33

34

35

36

37

38

39 40

41

42

43

44

45

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 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 than 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 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

must maintain reference counts, and allocate and deallocate objects in such a way that the visible effect is as if the objects were copied. (End of advice to implementors.)

#### 2.5.2 Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of 6 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, 8 an additional len argument is required to indicate the number of valid entries (unless this 9 number can be derived otherwise). The valid entries are at the beginning of the array; 10 len indicates how many of them there are, and need not be the size of the entire array. 11 The same approach is followed for other array arguments. In some cases NULL handles are 12considered valid entries. When a NULL argument is desired for an array of statuses, one 13 uses MPI\_STATUSES\_IGNORE. 14

2.5.3 State

17MPI procedures use at various places arguments with state types. The values of such a data 18 type are all identified by names, and no operation is defined on them. For example, the 19MPI\_TYPE\_CREATE\_SUBARRAY routine has a state argument order with values 20MPI\_ORDER\_C and MPI\_ORDER\_FORTRAN. 21

2223

15

16

#### 2.5.4 Named Constants

 $^{24}$ MPI procedures sometimes assign a special meaning to a special value of a basic type argu-25ment; e.g., tag is an integer-valued argument of point-to-point communication operations, 26with a special wild-card value, MPI\_ANY\_TAG. Such arguments will have a range of regular 27values, which is a proper subrange of the range of values of the corresponding basic type; 28 special values (such as MPI\_ANY\_TAG) will be outside the regular range. The range of regu-29lar values, such as tag, can be queried using environmental inquiry functions, see Chapter 9. 30 The range of other values, such as source, depends on values given by other MPI routines  $^{31}$ (in the case of **source** it is the communicator size).

32 MPI also provides predefined named constant handles, such as MPI\_COMM\_WORLD.

33 All named constants, with the exceptions noted below for Fortran, can be used in 34 initialization expressions or assignments, but not necessarily in array declarations or as 35 labels in C switch or Fortran select/case statements. This implies named constants 36 to be link-time but not necessarily compile-time constants. The named constants listed 37 below are required to be compile-time constants in both C and Fortran. These constants 38do not change values during execution. Opaque objects accessed by constant handles are 39 defined and do not change value between MPI initialization (MPI\_INIT) and MPI completion 40(MPI\_FINALIZE). The handles themselves are constants and can be also used in initialization  $^{41}$ expressions or assignments.

42The constants that are required to be compile-time constants (and can thus be used 43for array length declarations and labels in C switch and Fortran case/select statements)

- 44
- 45
- 4647
- 48

1

 $\mathbf{2}$ 

3 4

5

oro	٠
are	

MPI_MAX_PROCESSOR_NAME
MPI_MAX_LIBRARY_VERSION_STRING
MPI_MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING
MPI_MAX_INFO_KEY
MPI_MAX_INFO_VAL
MPI_MAX_OBJECT_NAME
MPI_MAX_PORT_NAME
MPI_VERSION
MPI_SUBVERSION
MPI_F_STATUS_SIZE (C only)
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
MPI_COUNT_KIND (Fortran only)
MPI_INTEGER_KIND (Fortran only)
MPI_OFFSET_KIND (Fortran only)
MPI_SUBARRAYS_SUPPORTED (Fortran only)
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
The constants that cannot be used in initialization expressions or assignments in For-
tran are as follows:
MPI_BOTTOM
MPI_STATUS_IGNORE
MPI_STATUSES_IGNORE
MPI_ERRCODES_IGNORE
MPI_IN_PLACE MPI_ARGV_NULL
MPI_ARGV_NULL
MPI_UNWEIGHTED
MPI_WEIGHTS_EMPTY
Advice to implementors. In Fortran the implementation of these special constants

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through PARAMETER statements) is not possible because an implementation cannot distinguish these values from valid data. Typically, these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C). (End of advice to implementors.)

### 2.5.5 Choice

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 + TS 29113 syntax TYPE(\*), DIMENSION(..); for C, we use void\*.

### Unofficial Draft for Comment Only

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 19.1.1. (End of advice to implementors.)

### 2.5.6 Absolute Addresses and Relative Address Displacements

8 Some MPI procedures use *address* arguments that represent an *absolute address* in the call-9 ing program, or *relative displacement* arguments that represent differences of two absolute 10 addresses. The datatype of such arguments is MPI\_Aint in C and INTEGER (KIND= 11 MPI\_ADDRESS\_KIND) in Fortran. These types must have the same width and encode address 12values in the same manner such that address values in one language may be passed directly 13 to another language without conversion. There is the MPI constant MPI\_BOTTOM to in-14dicate the start of the address range. For retrieving absolute addresses or any calculation 15with absolute addresses, one should use the routines and functions provided in Section 5.1.5. 16Section 5.1.12 provides additional rules for the correct use of absolute addresses. For ex-17pressions with relative displacements or other usage without absolute addresses, intrinsic 18 operators (e.g., +, -, \*) can be used. 19

20 21

22

23

24

25

26

27

28

Rationale. Byte displacement values need to be large enough to encode any value used for expressing absolute or relative memory addresses. Prior to MPI-4.0, some MPI routines used int in C and INTEGER in Fortran as the type for byte displacement arguments. To avoid breaking backward compatibility, this version of the standard continues to support int in C as well as INTEGER in Fortran in such routines. In addition, this version of the standard supports using MPI\_Aint in C (via separate "\_c"suffixed procedures) as well as INTEGER (KIND=MPI\_ADDRESS\_KIND) in Fortran (via polymorphic interfaces in newer MPI Fortran bindings (USE mpi\_f08)) in such routines. See Section 19.2 for a full explanation. (End of rationale.)

29 30 31

32

33

34

35

36

37

38 39

40

## 2.5.7 File Offsets

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

## 2.5.8 Counts

 $^{41}$ As described above, MPI defines types (e.g., MPI\_Aint) to address locations within memory 42and other types (e.g., MPI\_Offset) to address locations within files. In addition, some MPI procedures use *count* arguments that represent a number of MPI datatypes on which to 43 operate. Furthermore, *timestamps* in the context of the MPI Tool Information Interface are 44a count of clock ticks elapsed since some time in the past. At times, one needs a single 4546type that can be used to address locations within either memory or files as well as express 47count values, and that type is MPI\_Count in C and INTEGER(KIND=MPI\_COUNT\_KIND) in 48Fortran. These types must have the same width and encode values in the same manner

### Unofficial Draft for Comment Only

1

2

3

4

5

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. Even though the MPI\_Count type is large enough to encode address locations, MPI\_Count type shall not be used to represent an *absolute address*.

Rationale. Count values need to be large enough to encode any value used for expressing element counts, stride, offset, index, displacement, typemaps in memory, typemaps in file views, etc. Prior to MPI-4.0, many MPI routines used int in C and INTEGER in Fortran as the type for *count* arguments. To avoid breaking backward compatibility, this version of the standard continues to support int in C as well as INTEGER in Fortran in such routines. In addition, this version of the standard supports using MPI\_Count in C (via separate "\_c"suffixed procedures) as well as INTEGER (KIND=MPI\_COUNT\_KIND) in Fortran (via polymorphic interfaces in newer MPI Fortran bindings (USE mpi\_f08)) in such routines. See Section 19.2 for a

full explanation. (End of rationale.)

The phrase **large count** refers to the use of MPI\_Count and INTEGER (KIND=MPI\_COUNT\_KIND) parameter types.

There are cases where MPI\_UNDEFINED can be returned in a large count OUT parameter. Per Table A.1.1 (page 847), the MPI\_UNDEFINED constant is defined to be a C int (or unnamed enum) and a Fortran INTEGER. Implementations shall therefore choose the underlying types for MPI\_Count and INTEGER(KIND=MPI\_COUNT\_KIND) such that they can be compared to MPI\_UNDEFINED.

Advice to implementors. The comparison of MPI\_UNDEFINED to an MPI\_Count or INTEGER(KIND=MPI\_COUNT\_KIND) may need to be via a casting operation. (End of advice to implementors.)

# 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 <sup>46</sup> resolving Fortran names. Users of case sensitive languages should avoid any prefix of the <sup>47</sup> form "MPI\_" and "PMPI\_", where any of the letters are either upper or lower case. <sup>48</sup>

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

#-other

Done

#### Deprecated and Removed Interfaces 2.6.1

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 16, 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. previously

Some of the deprecated constructs are now removed, as documented in Chapter 17. They may still be provided by an implementation for backwards compatibility, but are not required.

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

#### Fortran Binding Issues 2.6.2

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 + TS 29113 and later if the mpi\_f08 module is used.

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

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

MPI\_NULL\_COPY\_FN). A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI\_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 9 and Annex A. Constants representing the maximum length of a string are one smaller in Fortran than

in C as discussed in Section 19.3.9.

Handles are represented in Fortran as INTEGERs, or as a BIND(C) derived type with the mpi\_f08 module; see Section 2.5.1. 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 19.1.16.

The support for large count and displacement in Fortran is only available when using newer MPI Fortran bindings (USE mpi\_f08).

4748

#-other 13 Done

1

2

3

4

5

6

7

8

9

10

11

12

14

15

16

17

18 19

2021

22

23

 $^{24}$ 

25

26

27

28

29

30

34

35

36

37

38

39

40

41

42

43

44

45

46

\mpifunc{MPI\_

COMM\_NULL

COPY\_FN}

#-update5

Done

PR449

in

#-error

2.6. LANGUAGE BINDING		prot	other mpi/mpif.h callback otypesFUNCTION are missing in callback index	23		#-error #-update3 Done in
Deprecated or removed	deprecated	removed	Replacement		1	A.1.3
construct	since	since			2	and A.1.4
MPI_ADDRESS	MPI-2.0	MPI-3.0	MPI_GET_ADDRESS		3	Done
MPI_TYPE_HINDEXED	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HINDEXED		4	in all
MPI_TYPE_HVECTOR	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HVECTOR			normal
MPI_TYPE_STRUCT	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_STRUCT		5	chapters
MPI_TYPE_EXTENT	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT		6	PR449
MPI_TYPE_UB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT		7	
MPI_TYPE_LB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT		8	
MPI_LB <sup>1</sup>	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED			
MPI_UB <sup>1</sup>	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED		9	Done
MPI_ERRHANDLER_CREATE	MPI-2.0	MPI-3.0	MPI_COMM_CREATE_ERRHANDLE	ER	10	in
MPI_ERRHANDLER_GET	MPI-2.0	MPI-3.0	MPI_COMM_GET_ERRHANDLER		11	additiona
MPI_ERRHANDLER_SET	MPI-2.0	MPI-3.0	MPI_COMM_SET_ERRHANDLER		12	I commit
$MPI_Handler_function^2$	MPI-2.0	MPI-3.0	$MPI\_Comm\_errhandler\_function^2$			of
MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL		13	PR449
MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL	FL	JNCT	
MPI_DUP_FN <sup>3</sup>	MPI-2.0		MPI_COMM_DUP_FN <sup>3</sup>			nust be 2!!!
MPI_NULL_COPY_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_COPY_FN <sup>3</sup>			
MPI_NULL_DELETE_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_DELETE_FN <sup>3</sup>		_	#-error
MPI_Copy_function <sup>2</sup>	MPI-2.0		MPI_Comm_copy_attr_function <sup>2</sup>		17	#-update7
COPY_FUNCTION <sup>3</sup> Wro	ng macro is use	ed.	COMM_COPY_ATTR_FUNCTION <sup>3</sup>		18	•
MPI_Delete_function <sup>2</sup> Cau	sed Function in	stead	MPI_Comm_delete_attr_function <sup>2</sup>		19	#-other
DELETE_FUNCTION <sup>3</sup> of C	allback Index!		COMM_DELETE_ATTR_FUNCTION	۷ <sup>3</sup>	20	#-error!
MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR			Done
MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR		21	DONE
MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR		22	
MPI_COMBINER_HVECTOR_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_HVECTOR <sup>4</sup>		23	
MPI_COMBINER_HINDEXED_INTEGER	4 _	MPI-3.0	MPI_COMBINER_HINDEXED <sup>4</sup>		24	
MPI_COMBINER_STRUCT_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_STRUCT <sup>4</sup>		25	
MPI::	MPI-2.2	MPI-3.0	C language binding			
MPI_CANCEL for send requests	MPI-3.2 4	0	no direct replacement		26	
MPI_INFO_GET	MPI-4.0		MPI_INFO_GET_STRING		27	
MPI_INFO_GET_VALUELEN	MPI-4.0		MPI_INFO_GET_STRING		28	
MPI_T_ERR_INVALID_ITEM	MPI-3.2 4.(		MPI_T_ERR_INVALID_INDEX		29	
MPI_SIZEOF	MPI-4.0	-	storage_size() <sup>5</sup> and c_sizeof()		29	#-other
<sup>1</sup> Predefined datatype.					30	
<sup>2</sup> Callback prototype definition.					31	Done
<sup>3</sup> Predefined callback routine.					32	
<sup>4</sup> Constant. storage_size()					33	
<sup>5</sup> Fortran intrinsic. It returns the size	e in bits instead	of bytes.				
Other entries are regular MPI routine		v			34	
					35	

### Table 2.1: Deprecated and Removed constructs

 $^{41}$ 

### 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

### Unofficial Draft for Comment Only

1 supplied in an include file mpi.h.  $\mathbf{2}$ Almost all C functions return an error code. The successful return code will be 3 MPI\_SUCCESS, but failure return codes are implementation dependent. 4 Type declarations are provided for handles to each category of opaque objects. 5Array arguments are indexed from zero. 6 Logical flags are integers with value 0 meaning "false" and a non-zero value meaning  $\overline{7}$ "true." 8 Choice arguments are pointers of type void\*. 9 10 2.6.4 Functions and Macros 11 An implementation is allowed to implement MPI\_WTIME, PMPI\_WTIME, MPI\_WTICK, 12PMPI\_WTICK, MPI\_AINT\_ADD, PMPI\_AINT\_ADD, MPI\_AINT\_DIFF, PMPI\_AINT\_DIFF, 13 and the handle-conversion functions (MPI\_Group\_f2c, etc.) in Section 19.3.4, and no others, 14as macros in C. 1516Advice to implementors. Implementors should document which routines are imple-17 mented as macros. (End of advice to implementors.) 18 19 If these routines are implemented as macros, they will not work Advice to users. 20with the MPI profiling interface. (End of advice to users.) 21222.7 Processes 23 $^{24}$ An MPI program consists of autonomous processes, executing their own code, in an MIMD 25style. The codes executed by each process need not be identical. The processes communicate 26via calls to MPI communication primitives. Typically, each process executes in its own 27address space, although shared-memory implementations of MPI are possible. 28This document specifies the behavior of a parallel program assuming that only MPI 29calls are used. The interaction of an MPI program with other possible means of commu-30 nication, I/O, and process management is not specified. Unless otherwise stated in the  $^{31}$ specification of the standard, MPI places no requirements on the result of its interaction 32 with external mechanisms that provide similar or equivalent functionality. This includes, 33 but is not limited to, interactions with external mechanisms for process control, shared and 34remote memory access, file system access and control, interprocess communication, process 35 signaling, and terminal I/O. High quality implementations should strive to make the results 36 of such interactions intuitive to users, and attempt to document restrictions where deemed 37 necessary. 38 39

40 41

42

43 44 45

46

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

The interaction of MPI and threads is defined in Section 11.6.

# 2.8 Error Handling

<sup>47</sup> MPI provides the user with reliable message transmission. A message sent is always re-<sup>48</sup> ceived correctly, and the user does not need to check for transmission errors, time-outs,

## Unofficial Draft for Comment Only

or other error conditions. In other words, MPI does not provide mechanisms for dealing with **transmission failures** in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, and to reflect only unrecoverable transmission failures. Whenever possible, such failures will be reflected as errors in the relevant communication call.

Similarly, MPI itself provides no mechanisms for handling MPI **process failures**, that is, when an MPI process unexpectedly and permanently stops communicating (e.g., a software or hardware crash results in an MPI process terminating unexpectedly).

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

In C and Fortran, almost all MPI calls return a code that indicates successful completion 19of the operation. Whenever possible, MPI calls return an error code if an error occurred 2021during 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 22mechanisms for users to change this default and to handle recoverable errors. The user may 23 $^{24}$ specify that no error is fatal, and handle error codes returned by MPI calls by himself or 25herself. Also, the user may provide his or her own error-handling routines, which will be 26invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 9.3. 27

Several factors limit the ability of MPI calls to return with meaningful error codes 2829when an error occurs. MPI may not be able to detect some errors; other errors may be too 30 expensive to detect in normal execution mode: some faults (e.g., memory faults) may corrupt the state of the MPI library and its outputs; finally some errors may be "catastrophic" and 31may prevent MPI from returning control to the caller. On the other hand, some errors 32 33 may be detected after the associated operation has completed; some errors may not have 34a 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. When MPI\_COMM\_SELF is not 35 initialized (i.e., before MPI\_INIT / MPI\_INIT\_THREAD or after MPI\_FINALIZE) the error 36 37 raises the **initial error handler** (set during the launch operation, see 11.8.4).

An example of such a case arises because of the nature of asynchronous communications: 38 39 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 40 41 error to be raised. If there is a subsequent call that relates to the same operation (e.g., a 42call that verifies that an asynchronous operation has completed) then the error argument 43associated with this call will be used to indicate the nature of the error. In a few cases, the 44error 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 4546with the ready mode).

This document does not specify the state of a computation after an erroneous MPI call <sup>47</sup> has occurred. The desired behavior is that a relevant error code be returned, and the effect <sup>48</sup>

#### Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$ 

6

7

8 9

10

11

12

13

14

15

16

17

1of the error be localized to the greatest possible extent. E.g., it is highly desirable that an  $\mathbf{2}$ erroneous receive call will not cause any part of the receiver's memory to be overwritten, 3 beyond the area specified for receiving the message.

4 Implementations may go beyond this document in supporting in a meaningful manner 5MPI calls that are defined here to be erroneous. For example, MPI specifies strict type 6 matching rules between matching send and receive operations: it is erroneous to send a 7floating point variable and receive an integer. Implementations may go beyond these type 8 matching rules, and provide automatic type conversion in such situations. It will be helpful 9 to generate warnings for such non-conforming behavior.

10 11

12

13

MPI defines a way for users to create new error codes as defined in Section 9.5.

#### 2.9 Implementation Issues

14There are a number of areas where an MPI implementation may interact with the operating 15environment and system. While MPI does not mandate that any services (such as signal 16handling) be provided, it does strongly suggest the behavior to be provided if those services 17are available. This is an important point in achieving portability across platforms that provide the same set of services. 19

2021

30

18

#### 2.9.1Independence of Basic Runtime Routines

22MPI programs require that library routines that are part of the basic language environment 23(such as write in Fortran and printf and malloc in ISO C) and are executed after

 $^{24}$ MPI\_INIT and before MPI\_FINALIZE operate independently and that their *completion* is 25independent of the action of other processes in an MPI program.

26Note that this in no way prevents the creation of library routines that provide parallel 27services whose operation is collective. However, the following program is expected to com-28plete in an ISO C environment regardless of the size of MPI\_COMM\_WORLD (assuming that 29printf is available at the executing nodes).

```
int rank;
^{31}
```

```
MPI_Init((void *)0, (void *)0);
32
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
33
```

```
if (rank == 0) printf("Starting program\n");
34
```

```
MPI_Finalize();
35
```

36 The corresponding Fortran programs are also expected to complete. 37

An example of what is *not* required is any particular ordering of the action of these 38 routines when called by several tasks. For example, MPI makes neither requirements nor 39 recommendations for the output from the following program (again assuming that I/O is 40 available at the executing nodes). 41

```
42
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

43 printf("Output from task rank %d\n", rank);

In addition, calls that fail because of resource exhaustion or other error are not con-45sidered a violation of the requirements here (however, they are required to complete, just 46 not to complete successfully). 47

48

### 2.9.2 Interaction with Signals

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

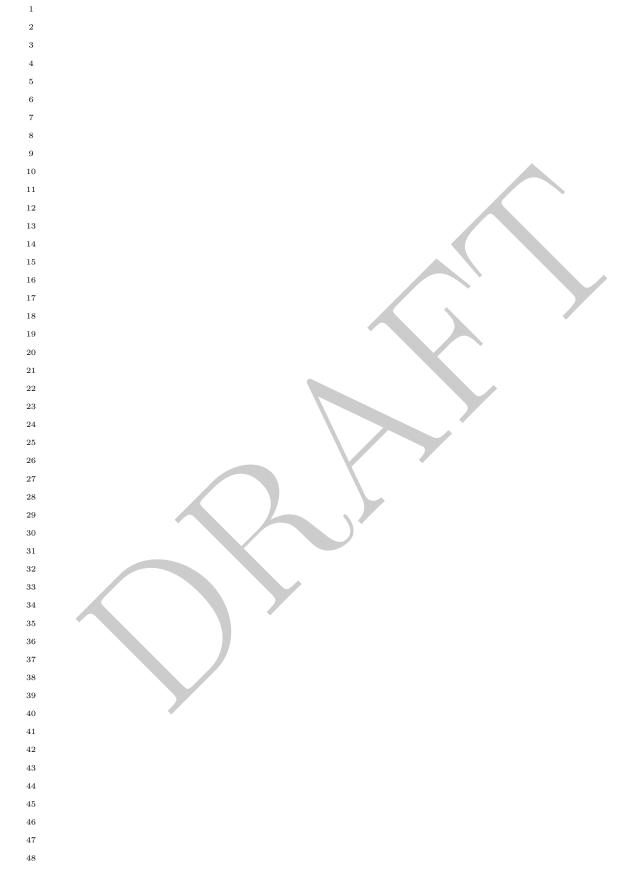
In multithreaded environments, users can avoid conflicts between signals and the MPI library by catching signals only on threads that do not execute MPI calls. High quality single-threaded implementations will be signal safe: an MPI call suspended by a signal will resume and complete normally after the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

# 2.10 Examples

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

**Unofficial Draft for Comment Only** 

 $\overline{7}$ 



# 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);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0)
                       /* code for process zero */
                                                                                   29
  {
      strcpy(message,"Hello, there")
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1) /* code for process one */
                                                                                   34
  Ł
                                                                                   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 43 44from 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, size 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

1 2 3

6

1516

17

18

19

24

25

26

27

28

30

31

32

33

38

39

40

41

4546

47

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

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

# 3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

26	IN	buf	initial address of send buffer (choice)
27 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 33	IN	tag	message tag (integer)
33 34	IN	comm	communicator (handle)

### 36 C binding

- int MPI\_Send\_c(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)
- 42 Fortran 2008 binding

```
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
```

### Unofficial Draft for Comment Only

17 18

19 20

21

22 23 24

25

35

37

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

#### Fortran binding

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

#### 3.2.2 Message Data

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

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

	MPI datatype	Fortran datatype
	MPI_INTEGER	INTEGER
	MPI_REAL	REAL
	MPI_DOUBLE_PRECISION	DOUBLE PRECISION
	MPI_COMPLEX	COMPLEX
	MPI_LOGICAL	LOGICAL
	MPI_CHARACTER	CHARACTER(1)
7	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 5.2.

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 <sup>48</sup>

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

 $45 \\ 46$ 

1	MPI datatype	C datatype
2	MPI_CHAR	char
3		(treated as printable character)
4	MPI_SHORT	signed short int
5	MPI_INT	signed int
6	MPI_LONG	signed long int
7	MPI_LONG_LONG_INT	signed long long int
8	MPI_LONG_LONG (as a synonym)	signed long long int
9	MPI_SIGNED_CHAR	signed char
0		(treated as integral value)
1	MPI_UNSIGNED_CHAR	unsigned char
2		(treated as integral value)
3	MPI_UNSIGNED_SHORT	unsigned short int
4	MPI_UNSIGNED	unsigned int
5	MPI_UNSIGNED_LONG	unsigned long int
.6	MPI_UNSIGNED_LONG_LONG	unsigned long long int
7	MPI_FLOAT	float
.8	MPI_DOUBLE	double
.9	MPI_LONG_DOUBLE	long double
0	MPI_WCHAR	wchar_t
21		(defined in <stddef.h>)</stddef.h>
22		(treated as printable character)
23	MPI_C_BOOL	_Bool
4	MPI_INT8_T	int8_t
5	MPI_INT16_T	int16_t
6	MPI_INT32_T	int32_t
7	MPI_INT64_T	int64_t
8	MPI_UINT8_T	uint8_t
9	MPI_UINT16_T	uint16_t
0	MPI_UINT32_T	uint32_t
1	MPI_UINT64_T	uint64_t
2	MPI_C_COMPLEX	float _Complex
3	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
34	MPI_C_DOUBLE_COMPLEX	double _Complex
35	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
36	MPI_BYTE	
37	MPI_PACKED	
38		
39	Table 2.2. Prodefined MDI detationed	corresponding to C datatupos
40	Table 3.2: Predefined MPI datatypes	corresponding to C datatypes
41		
42	data types <sup>1</sup> : $MPI_DOUBLE_COMPLEX$ for double	* *
3	L	FALA MDI DEALO and

- <sup>43</sup> be of type DOUBLE COMPLEX; MPI\_REAL2, MPI\_REAL4, MPI\_REAL8, and
- <sup>44</sup> MPI\_REAL16 for Fortran reals, declared to be of type REAL\*2, REAL\*4,
- $^{45}$   $REAL*8, \ {\rm and}\ REAL*16, \ {\rm respectively}; \ {\sf MPI_INTEGER1}, \ {\sf MPI_INTEGER2}, \ {\sf MPI_INTEGER4}, \ {\rm and}$

 <sup>&</sup>lt;sup>46</sup> <sup>1</sup>These types, such as DOUBLE COMPLEX and INTEGER\*4, are not specified by any Fortran standard but are extensions commonly accepted by Fortran compilers.
 <sup>48</sup>

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\_INTEGER8 for Fortran integers, declared to be of type INTEGER\*1, INTEGER\*2, INTEGER\*4, and INTEGER\*8, respectively; MPI\_COMPLEX4, MPI\_COMPLEX8, MPI\_COMPLEX16, and MPI\_COMPLEX32 for complex numbers in Fortran declared to be of type COMPLEX\*4, COMPLEX\*8, COMPLEX\*16, and COMPLEX\*32, 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 MPI-defined C types MPI\_Aint, MPI\_Offset, and MPI\_Count and their Fortran equivalents 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 19.3.6 and 19.3.10 on page 834 and 842 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
MPI_CXX_BOOL	bool
MPI_CXX_FLOAT_COMPLEX	<pre>std::complex<float></float></pre>
MPI_CXX_DOUBLE_COMPLEX	<pre>std::complex<double></double></pre>
MPI_CXX_LONG_DOUBLE_COMPLEX	<pre>std::complex<long double=""></long></pre>

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

	44
source	45
destination	46
$\operatorname{tag}$	47
communicator	48

## Unofficial Draft for Comment Only

 $^{24}$ 

1 The message source is implicitly determined by the identity of the message sender. The  $\mathbf{2}$ other fields are specified by arguments in the send operation. 3

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

4 The integer-valued message tag is specified by the tag argument. This integer can be 5used by the program to distinguish different types of messages. The range of valid tag 6 values is  $0, \ldots, \mathsf{UB}$ , where the value of  $\mathsf{UB}$  is implementation dependent. It can be found by 7querying the value of the attribute MPI\_TAG\_UB, as described in Chapter 9. MPI requires 8 that UB be no less than 32767.

9 The comm argument specifies the **communicator** that is used for the send operation. 10 Communicators are explained in Chapter 7; below is a brief summary of their usage.

11 A communicator specifies the communication context for a communication operation. 12Each communication context provides a separate "communication universe": messages are 13 always received within the context they were sent, and messages sent in different contexts 14do not interfere.

15The communicator also specifies the set of processes that share this communication 16context. This **process group** is ordered and processes are identified by their rank within 17this group. Thus, the range of valid values for dest is  $0, \ldots, n-1 \cup \{MPI\_PROC\_NULL\}$ , where 18 *n* is the number of processes in the group. (If the communicator is an inter-communicator, 19then destinations are identified by their rank in the remote group. See Chapter 7.)

20When using the World Model (see Section 11.1), a predefined communicator 21MPI\_COMM\_WORLD is provided by MPI. It allows communication with all processes that 22are accessible after MPI initialization and processes are identified by their rank in the group 23of MPI\_COMM\_WORLD.

 $^{24}$ 

Users that are comfortable with the notion of a flat name space Advice to users. 2526for processes, and a single communication context, as offered by most existing communication libraries, need only use the World Model for MPI initialization, and the 27predefined variable MPI\_COMM\_WORLD as the comm argument. This will allow com-28munication with all the processes available at initialization time. 29

30 Users may define new communicators, as explained in Chapter 7. Communicators  $^{31}$ provide an important encapsulation mechanism for libraries and modules. They allow 32 modules to have their own disjoint communication universe and their own process 33 numbering scheme. (End of advice to users.) 34

The message envelope would normally be encoded by a Advice to implementors. fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (End of advice to implementors.)

3.2.4 **Blocking Receive** 

The syntax of the blocking receive operation is given below.

44 45

46

35

36

37

38

39

40 41

42

43

MPI_REC	CV(buf, count, datatype, sourc	e, tag, comm, status)	1				
OUT	buf	initial address of receive buffer (choice)	2 3				
IN	count	number of elements in receive buffer (non-negative integer)	3 4 5				
IN	datatype	datatype of each receive buffer element (handle)	6				
IN	source	rank of source or MPI_ANY_SOURCE (integer)	7				
IN	tag	message tag or MPI_ANY_TAG (integer)	8 9				
IN	comm	communicator (handle)	10				
OUT	status	status object (Status)	11				
001			12				
C bindi	ng		13				
	_Recv(void *buf, int cour	nt, MPI_Datatype datatype, int source,	14 15				
	int tag, MPI_Comm	comm, MPI_Status *status)	16				
int MPI_		ount count, MPI_Datatype datatype, g, MPI_Comm comm, MPI_Status *status)	17 18				
<b>D</b> (		s,,,,	19				
	2008 binding	······	20				
	E(*), DIMENSION() :: bu	source, tag, comm, status, ierror)	21				
	EGER, INTENT(IN) :: count		22 23				
	TYPE(MPI_Datatype), INTENT(IN) :: datatype						
TYPE(MPI_Comm), INTENT(IN) :: comm							
TYPE(MPI_Status) :: status							
INTE	INTEGER, OPTIONAL, INTENT(OUT) :: ierror						
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)							
	E(*), DIMENSION() :: bu		29				
INTE	EGER(KIND=MPI_COUNT_KIND)	), INTENT(IN) :: count	30				
	E(MPI_Datatype), INTENT()		31				
	EGER, INTENT(IN) :: sourc	•	32				
	E(MPI_Comm), INTENT(IN)	:: comm	33				
	E(MPI_Status) :: status EGER, OPTIONAL, INTENT(OU		34 35				
LNIF	GER, OPIIONAL, INIENI(OC	JI) :: lerror	36				
Fortran	J		37				
		SOURCE, TAG, COMM, STATUS, IERROR)	38				
01	De> BUF(*)		39				
LNTE	IGER COUNT, DATATYPE, SUC IERROR	JRCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),	40				
			41				
	0	ll are described in Section 3.4.	42				
'l'he	receive butter consists of the	storage containing count consecutive elements of the	43				

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.

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

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.*)

14The selection of a message by a receive operation is governed by the value of the 15message envelope. A message can be received by a receive operation if its envelope matches 16the source, tag and comm values specified by the receive operation. The receiver may specify 17a wildcard MPI\_ANY\_SOURCE value for source, and/or a wildcard MPI\_ANY\_TAG value for 18 tag, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value 19for comm. Thus, a message can be received by a receive operation only if it is addressed 20to the receiving process, has a matching communicator, has matching source unless source 21= MPI\_ANY\_SOURCE in the pattern, and has a matching tag unless tag = MPI\_ANY\_TAG in 22 the pattern. 23

The message tag is specified by the tag argument of the receive operation. The argument source, if different from MPI\_ANY\_SOURCE, is specified as a rank within the process group associated with that same communicator (remote process group, for intercommunicators). Thus, the range of valid values for the source argument is  $\{0, ..., n-1\} \cup$ {MPI\_ANY\_SOURCE} $\cup$ {MPI\_PROC\_NULL}, where *n* is the number of processes in this group.

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

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

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

The use of  $dest = MPI_PROC_NULL$  or source = MPI\_PROC\_NULL to define a "dummy" destination or source in any send or receive call is described in Section 3.10.

#### 45 46 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

## Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

12

13

34

35

36 37

38

39

40

 $41 \\ 42$ 

43

(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 MPI-defined. 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, MPI\_ERROR, and TYPE(MPI\_Status) are defined to allow conversion between both status representations. Conversion routines are provided in Section 19.3.5.

*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.*)

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

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

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

 $44 \\ 45$ 

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

1	MPI_GET	_COUNT(status, datatype, cou	nt)
2	IN	status	return status of receive operation (Status)
3	IN	datatype	datatype of each receive buffer entry (handle)
4 5	OUT	count	number of received entries (integer)
6	001	count	number of received entries (integer)
7	C bindin	ıg	
8		0	s *status, MPI_Datatype datatype,
9 10		int *count)	
11 12	int MPI_	Get_count_c(const MPI_Stat MPI_Count *count)	tus *status, MPI_Datatype datatype,
13	Fortran	2008 binding	
14		count(status, datatype, co	punt, ierror)
15 16		(MPI_Status), INTENT(IN)	
17		(MPI_Datatype), INTENT(IN) GER, INTENT(OUT) :: count	) :: datatype
18		GER, INTENI(UUI) :: COUNT GER, OPTIONAL, INTENT(OUT)	) :: jerror
19			
20 21		count(status, datatype, co (MPI_Status), INTENT(IN)	
22		(MPI_Datatype), INTENT(IN)	
23	INTE	GER(KIND=MPI_COUNT_KIND),	INTENT(OUT) :: count
24	INTE	GER, OPTIONAL, INTENT(OUT)	) :: ierror
25	Fortran	binding	
26 27		COUNT(STATUS, DATATYPE, CO	
28	INTE	GER STATUS(MPI_STATUS_SIZ	E), DATATYPE, COUNT, IERROR
29			red. (Again, we count <i>entries</i> , each of type <i>datatype</i> ,
30			ld match the argument provided by the receive call
31 32			er of entries received exceeds the limits of the count is the value of count to MPI_UNDEFINED. There are
33	-		t can be set to MPI_UNDEFINED; see Section 5.1.11.
34			
35			g libraries use INOUT count, tag and source argu-
36 27			pecify the selection criteria for incoming messages ues of the received message. The use of a separate
37 38			hat are often attached with INOUT argument (e.g.,
39			as the tag in a receive). Some libraries use calls
40	that	refer implicitly to the "last m	essage received." This is not thread safe.
41			• MPI_GET_COUNT so as to improve performance.
42 43		6 6	nout counting the number of elements it contains,
43 44			needed. Also, this allows the same function to be or MPI_IPROBE. With a status from MPI_PROBE
45			bes are allowed as in a call to MPI_RECV to receive
46		message. (End of rationale.)	
47		/	
48			

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 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.

## 3.2.6 Passing MPI\_STATUS\_IGNORE for Status

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

To cope with this problem, there are two predefined constants, MPI\_STATUS\_IGNORE and MPI\_STATUSES\_IGNORE, which when passed to a receive, probe, wait, or test function, inform the implementation that the status fields are not to be filled in. Note that MPI\_STATUS\_IGNORE is not a special type of MPI\_Status object; rather, it is a special value for the argument. In C one would expect it to be NULL, not the address of a special MPI\_Status.

MPI\_STATUS\_IGNORE, and the array version MPI\_STATUSES\_IGNORE, can be used everywhere a status argument is passed to a receive, wait, or test function. MPI\_STATUS\_IGNORE cannot be used when status is an IN argument. Note that in Fortran MPI\_STATUS\_IGNORE and MPI\_STATUSES\_IGNORE are objects like MPI\_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

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

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

### Unofficial Draft for Comment Only

 $\overline{7}$ 

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.

## 3.2.7 Send-Receive

The send-receive operations combine in one call the sending of a message to one desti-8 nation and the receiving of another message, from another process. The two (source and 9 destination) are possibly the same. A send-receive operation is very useful for executing 10 a shift operation across a chain of processes. If blocking sends and receives are used for 11 such a shift, then one needs to order the sends and receives correctly (for example, even 12processes send, then receive, odd processes receive first, then send) so as to prevent cyclic 13 dependencies that may lead to deadlock. When a send-receive operation is used, the com-14 munication subsystem takes care of these issues. The send-receive operation can be used 15in conjunction with the functions described in Chapter 8 in order to perform shifts on var-16ious logical topologies. Also, a send-receive operation is useful for implementing remote 17procedure calls. 18

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, status)

25		-	
26	IN	sendbuf	initial address of send buffer (choice)
27	IN	sendcount	number of elements in send buffer (non-negative
28			integer)
29 30	IN	sendtype	type of elements in send buffer (handle)
31	IN	dest	rank of destination (integer)
32	IN	sendtag	send tag (integer)
33 34	OUT	recvbuf	initial address of receive buffer (choice)
35	1N	recvcount	number of elements in receive buffer (non-negative
36			integer)
37	IN	recvtype	type of elements receive buffer element (handle)
38 39	IN	source	rank of source or $MPI\_ANY\_SOURCE$ (integer)
40	IN	recvtag	receive tag or $MPI\_ANY\_TAG$ (integer)
41	IN	comm	communicator (handle)
42 43	OUT	status	status object (Status)
44			
45	C bindir	ıg	
46	int MPI_		lbuf, int sendcount, MPI_Datatype sendtype,
47		int dest, int sendta	ag, void *recvbuf, int recvcount,

48

5 6

7

22 23

24

1 MPI\_Datatype recvtype, int source, int recvtag, MPI\_Comm comm, 2 MPI\_Status \*status) 3 Fortran 2008 binding MPI\_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror) TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, recvtag TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 10 TYPE(\*), DIMENSION(...) :: recvbuf 11 TYPE(MPI\_Comm), INTENT(IN) :: comm 12TYPE(MPI\_Status) :: status 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 14 15Fortran binding 16MPI\_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, 17RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR) 18 <type> SENDBUF(\*), RECVBUF(\*) 19 INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR 2021Execute a blocking send and receive operation. Both send and receive use the same 22 communicator, but possibly different tags. The send buffer and receive buffers must be 23 disjoint, and may have different lengths and datatypes.  $^{24}$ The semantics of a send-receive operation is what would be obtained if the caller forked 25two concurrent threads, one to execute the send, and one to execute the receive, followed 26by a join of these two threads. 272829 MPI\_SENDRECV\_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, 30 status) 31INOUT buf initial address of send and receive buffer (choice) 32 IN number of elements in send and receive buffer 33 count (non-negative integer) 34 35IN datatype type of elements in send and receive buffer (handle) 36 IN dest rank of destination (integer) 37 IN sendtag send message tag (integer) 38 39 rank of source or MPI\_ANY\_SOURCE (integer) IN source 40 receive message tag or MPI\_ANY\_TAG (integer) IN recvtag 41 IN comm communicator (handle) 42OUT status status object (Status) 43 44 C binding 4546int MPI\_Sendrecv\_replace(void \*buf, int count, MPI\_Datatype datatype, 47int dest, int sendtag, int source, int recvtag, MPI\_Comm comm,

MPI\_Status \*status)

## Unofficial Draft for Comment Only

1 Fortran 2008 binding  $\mathbf{2}$ MPI\_Sendrecv\_replace(buf, count, datatype, dest, sendtag, source, recvtag, 3 comm, status, ierror) 4 TYPE(\*), DIMENSION(..) :: buf 5INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag 6 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 7 TYPE(MPI\_Comm), INTENT(IN) :: comm 8 TYPE(MPI\_Status) :: status 9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 Fortran binding 11 MPI\_SENDRECV\_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, 12COMM, STATUS, IERROR) 13 <type> BUF(\*) 14 INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, 15STATUS(MPI\_STATUS\_SIZE), IERROR 1617 Execute a blocking send and receive. The same buffer is used both for the send and 18 for the receive, so that the message sent is replaced by the message received. 19Advice to implementors. Additional intermediate buffering is needed for the "replace" 20variant. (End of advice to implementors.) 2122 23Data Type Matching and Data Conversion 3.3 2425Type Matching Rules 3.3.1 26One can think of message transfer as consisting of the following three phases. 27281. Data is pulled out of the send buffer and a message is assembled. 29 30 2. A message is transferred from sender to receiver. 313. Data is pulled from the incoming message and disassembled into the receive buffer. 32 33 Type matching has to be observed at each of these three phases: The type of each 34 variable in the sender buffer has to match the type specified for that entry by the send 35 operation; the type specified by the send operation has to match the type specified by the 36 receive operation; and the type of each variable in the receive buffer has to match the type 37 specified for that entry by the receive operation. A program that fails to observe these three 38 rules is erroneous. 39 To define type matching more precisely, we need to deal with two issues: matching of 40 types of the host language with types specified in communication operations; and matching 41 of types at sender and receiver. 42The types of a send and receive match (phase two) if both operations use identical 43 names. That is, MPI\_INTEGER matches MPI\_INTEGER, MPI\_REAL matches MPI\_REAL, and 44 so on. There is one exception to this rule, discussed in Section 5.2: the type  $MPI_PACKED$ 45 can match any other type. 46 The type of a variable in a host program matches the type specified in the commu-47 nication operation if the datatype name used by that operation corresponds to the basic

type 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 and 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 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 5.2. 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.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

This code is correct if both a and b are real arrays of size  $\geq 10$ . (In Fortran, it might be correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced to an array with ten reals.)

**Example 3.2** Sender and receiver do not specify matching types.

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

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

**Example 3.3** Sender and receiver specify communication of untyped values.

 $\overline{7}$ 

 $45 \\ 46$ 

```
1
     CALL MPI_COMM_RANK(comm, rank, ierr)
\mathbf{2}
      IF (rank .EQ. 0) THEN
3
         CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
4
     ELSE IF (rank .EQ. 1) THEN
5
         CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
6
     END IF
7
          This code is correct, irrespective of the type and size of a and b (unless this results in
8
      an out of bounds memory access).
9
10
           Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND,
11
           then MPI will send the data stored at contiguous locations, starting from the address
12
           indicated by the buf argument. This may have unexpected results when the data
13
           layout is not as a casual user would expect it to be. For example, some Fortran
14
           compilers implement variables of type CHARACTER as a structure that contains the
15
           character length and a pointer to the actual string. In such an environment, sending
16
           and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have
17
           the anticipated result of transferring the character string. For this reason, the user is
18
           advised to use typed communications whenever possible. (End of advice to users.)
19
20
      Type MPI_CHARACTER
21
22
      The type MPI_CHARACTER matches one character of a Fortran variable of type
23
      CHARACTER, rather than the entire character string stored in the variable. Fortran variables
^{24}
      of type CHARACTER or substrings are transferred as if they were arrays of characters. This
25
     is illustrated in the example below.
26
27
     Example 3.4 Transfer of Fortran CHARACTERs.
28
     CHARACTER*10 a
29
      CHARACTER*10 b
30
^{31}
     CALL MPI_COMM_RANK(comm, rank, ierr)
32
      IF (rank .EQ. 0) THEN
33
34
         CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
     ELSE IF (rank .EQ. 1) THEN
35
         CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
36
     END IF
37
38
          The last five characters of string b at process 1 are replaced by the first five characters
39
      of string a at process 0.
40
41
           Rationale. The alternative choice would be for MPI_CHARACTER to match a character
42
           of arbitrary length. This runs into problems.
43
44
           A Fortran character variable is a constant length string, with no special termina-
45
           tion symbol. There is no fixed convention on how to represent characters, and how
46
           to store their length. Some compilers pass a character argument to a routine as a
47
           pair of arguments, one holding the address of the string and the other holding the
48
           length of string. Consider the case of an MPI communication call that is passed a
```

## Unofficial Draft for Comment Only

communication buffer with type defined by a derived datatype (Section 5.1). 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. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical and character values, and to convert a floating point value to the nearest value that can be represented on the target system.

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

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

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

Consider the three examples, 3.1-3.3. The first program is correct, assuming that **a** and **b** are REAL arrays of size  $\geq 10$ . If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number 

### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

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

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

- <sup>11</sup> Data representation conversion also applies to the envelope of a message: source, des-<sup>12</sup> tination and tag are all integers that may need to be converted.
- $13 \\ 14$

15 16

17

18 19

20

21

22 23

 $^{24}$ 

25

26 27

28

1

 $\mathbf{2}$ 

3

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

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

MPI requires support for inter-language communication, e.g., if messages are sent using an MPI procedure from the MPI C language interface and received using an MPI procedure from one of the MPI Fortran language interfaces. The behavior is defined in Section 19.3.

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 2.4.2, MPI\_BSEND is a completing 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 completing and therefore blocking operation-related procedure is 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 communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is *non-local*.

39 A send that uses the **ready** communication mode may be started *only* if the matching 40 receive is already posted. Otherwise, the operation is erroneous and its outcome is unde-41 fined. On some systems, this allows the removal of a hand-shake operation that is otherwise 42required and results in improved performance. The completion of the send operation does 43not depend on the status of a matching receive, and merely indicates that the send buffer 44can be reused. A send operation that uses the ready mode has the same semantics as a 45standard send operation, or a synchronous send operation; it is merely that the sender 46provides additional information to the system (namely that a matching receive is already 47posted), that can save some overhead. In a correct program, therefore, a ready send could 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

21

22

23

 $^{24}$ 

25

26 27

28

29 30

 $^{31}$ 

32

33

34

35

36

37

```
1
     be replaced by a standard send with no effect on the behavior of the program other than
\mathbf{2}
     performance.
3
         Three additional send functions are provided for the three additional communication
4
     modes. The communication mode is indicated by a one letter prefix: B for buffered, S for
5
     synchronous, and R for ready.
6
7
     MPI_BSEND(buf, count, datatype, dest, tag, comm)
8
9
       IN
                 buf
                                             initial address of send buffer (choice)
10
       IN
                                             number of elements in send buffer (non-negative
                 count
11
                                             integer)
12
                                             datatype of each send buffer element (handle)
       IN
                datatype
13
14
       IN
                 dest
                                             rank of destination (integer)
15
       IN
                                             message tag (integer)
                 tag
16
       IN
                 comm
                                             communicator (handle)
17
18
19
     C binding
     int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest,
20
21
                    int tag, MPI_Comm comm)
22
     int MPI_Bsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
23
                    int dest, int tag, MPI_Comm comm)
24
25
     Fortran 2008 binding
26
     MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
27
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
          INTEGER, INTENT(IN) :: count, dest, tag
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
33
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
34
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         INTEGER, INTENT(IN) :: dest, tag
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     Fortran binding
41
     MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
42
          <type> BUF(*)
43
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
44
         Send in buffered mode.
45
46
47
48
```

MPI_SSEND(buf, count, datatype,	dest,	tag,	comm)	
---------------------------------	-------	------	-------	--

		- ,	_
IN	buf	initial address of send buffer (choice)	2
IN	count	number of elements in send buffer (non-negative integer)	4
	1	<u> </u>	6
IN	datatype	datatype of each send buffer element (handle)	7
IN	dest	rank of destination (integer)	8
IN	tag	message tag (integer)	9
IN	comm	communicator (handle)	10
			11

## C binding

int	MPI_Ssend(	const v	void *but	f, int	count,	MPI_Datatype	datatype,	int	dest,
	i	int tag	, MPI_Co	mm com	m)				
÷+	MDT Carad	- ( +		с мт		MDT	Detetore		

<pre>int MPI_Ssend_c(const void *buf,</pre>	MPI_Count count,	MPI_Datatype	datatype,
int dest, int tag, M	(PI_Comm comm)		
Fortran 2008 binding			

## Fortran 2008 binding

Fortran 2008 binding	19
<pre>MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)</pre>	20
TYPE(*), DIMENSION(), INTENT(IN) :: buf	
INTEGER, INTENT(IN) :: count, dest, tag	21
TYPE(MPI_Datatype), INTENT(IN) :: datatype	22
TYPE(MPI_Comm), INTENT(IN) :: comm	23
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
	25
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)	26
TYPE(*), DIMENSION(), INTENT(IN) :: buf	27
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	28
TYPE(MPI_Datatype), INTENT(IN) :: datatype	29
INTEGER, INTENT(IN) :: dest, tag	30
TYPE(MPI_Comm), INTENT(IN) :: comm	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
Teature his dias	33
Fortran binding	34
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	35
<type> BUF(*)</type>	36
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	37
Send in synchronous mode.	38
	39
	40
	10

1 MPI\_RSEND(buf, count, datatype, dest, tag, comm) 2 IN buf initial address of send buffer (choice) 3 IN count number of elements in send buffer (non-negative 4 integer) 56 IN datatype datatype of each send buffer element (handle) 7 IN dest rank of destination (integer) 8 IN tag message tag (integer) 9 10 IN comm communicator (handle) 11 12C binding 13 int MPI\_Rsend(const void \*buf, int count, MPI\_Datatype datatype, int dest, 14int tag, MPI\_Comm comm) 15int MPI\_Rsend\_c(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 16int dest, int tag, MPI\_Comm comm) 1718 Fortran 2008 binding 19MPI\_Rsend(buf, count, datatype, dest, tag, comm, ierror) 20TYPE(\*), DIMENSION(..), INTENT(IN) :: buf 21INTEGER, INTENT(IN) :: count, dest, tag 22 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 23TYPE(MPI\_Comm), INTENT(IN) :: comm 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526MPI\_Rsend(buf, count, datatype, dest, tag, comm, ierror) TYPE(\*), DIMENSION(..), INTENT(IN) :: buf 27INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 28TYPE(MPI\_Datatype), INTENT(IN) :: datatype 29 INTEGER, INTENT(IN) :: dest, tag 30 TYPE(MPI\_Comm), INTENT(IN) :: comm 31INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 33 Fortran binding 34MPI\_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 35 <type> BUF(\*) 36 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 37 Send in ready mode. 38 There is only one receive operation, but it matches any of the send modes. The receive 39 operation described in the last section is *blocking*: it returns only after the receive buffer 40contains the newly received message. A receive can complete before the matching send has 41 42completed (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 43 is blocked on a send or receive operation, and schedule another thread for execution in 44the same address space. In such a case it is the user's responsibility not to modify a 45communication buffer until the communication completes. Otherwise, the outcome of the 4647computation is undefined. 48

	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.	1 $2$
	It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the sender until a matching receive occurs by using the synchronous send mode.	3 4 5 6
	A possible communication protocol for the various communication modes is outlined below.	7
	ready send: The message is sent as soon as possible.	9
	<i>synchronous send</i> : 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.	10 11 12 13
	<i>standard send</i> : First protocol may be used for short messages, and second protocol for long messages.	13 14 15
	<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).	16 17
	Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.	18 19 20
	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.	20 21 22
	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.	23 24
	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> )	25 26 27 28
3.5	Semantics of Point-to-Point Communication	29 30
	lid MPI implementation guarantees certain general properties of point-to-point com- cation, which are described in this section.	31 32 33
		34
	Messages are <i>non-overtaking</i> : If a sender sends two messages in succession to the	35
	destination, and both match the same receive, then this operation cannot receive the ad message if the first one is still pending. If a receiver posts two receives in succession,	36
	both match the same message, then the second receive operation cannot be satisfied	37 38
	is message, if the first one is still pending. This requirement facilitates matching of	39
-	to receives. It guarantees that message-passing code is deterministic, if processes are	40
	e-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the	41
-	described later, such as MPI_CANCEL or MPI_WAITANY, are additional sources of	42
	eterminism.)	43
	If a process has a single thread of execution, then any two communications executed	44
-	is process are ordered. On the other hand, if the process is multithreaded, then the	45
	ntics of thread execution may not define a relative order between two send operations	46
execu	ited by two distinct threads. The operations are logically concurrent, even if one	47

## **Unofficial Draft for Comment Only**

physically precedes the other. In such a case, the two messages sent can be received in

1 any order. Similarly, if two receive operations that are logically concurrent receive two  $\mathbf{2}$ successively sent messages, then the two messages can match the two receives in either 3 order. 4 **Example 3.5** An example of non-overtaking messages. 56 CALL MPI\_COMM\_RANK(comm, rank, ierr) 7 IF (rank .EQ. 0) THEN 8 CALL MPI\_BSEND(buf1, count, MPI\_REAL, 1, tag, comm, ierr) 9 CALL MPI\_BSEND(buf2, count, MPI\_REAL, 1, tag, comm, ierr) 10 ELSE IF (rank .EQ. 1) THEN 11 CALL MPI\_RECV(buf1, count, MPI\_REAL, 0, MPI\_ANY\_TAG, comm, status, ierr) 12CALL MPI\_RECV(buf2, count, MPI\_REAL, 0, tag, comm, status, ierr) 13 END IF 1415The message sent by the first send must be received by the first receive, and the message 16sent by the second send must be received by the second receive. 1718 **Progress** If a pair of matching send and receives have been initiated on two processes, then 19at least one of these two operations will complete, independently of other actions in the 20system: the send operation will complete, unless the receive is satisfied by another message, 21and completes; the receive operation will complete, unless the message sent is consumed by 22 another matching receive that was posted at the same destination process. 23 $^{24}$ **Example 3.6** An example of two, intertwined matching pairs. 2526CALL MPI\_COMM\_RANK(comm, rank, ierr) 27IF (rank .EQ. 0) THEN 28CALL MPI\_BSEND(buf1, count, MPI\_REAL, 1, tag1, comm, ierr) 29CALL MPI\_SSEND(buf2, count, MPI\_REAL, 1, tag2, comm, ierr) 30ELSE IF (rank .EQ. 1) THEN  $^{31}$ CALL MPI\_RECV(buf1, count, MPI\_REAL, 0, tag2, comm, status, ierr) 32 CALL MPI\_RECV(buf2, count, MPI\_REAL, 0, tag1, comm, status, ierr) 33 END IF 34Both processes invoke their first communication call. Since the first send of process zero 35 uses the buffered mode, it must complete, irrespective of the state of process one. Since 36 no matching receive is posted, the message will be copied into buffer space. (If insufficient 37 buffer space is available, then the program will fail.) The second send is then invoked. At 38that point, a matching pair of send and receive operation is enabled, and both operations 39 must complete. Process one next invokes its second receive call, which will be satisfied by 40

the buffered message. Note that process one received the messages in the reverse order they

Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose

that a send is posted. Then it is possible that the destination process repeatedly posts a

receive that matches this send, yet the message is never received, because it is each time

overtaken by another message, sent from another source. Similarly, suppose that a receive

was posted by a multithreaded process. Then it is possible that messages that match this

**Unofficial Draft for Comment Only** 

52

41

42 43 44

45

46

47

48

were sent.

receive are repeatedly received, yet the receive is never satisfied, because it is overtaken by other receives posted at this node (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations.

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

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

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

### 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

```
1
     CALL MPI_COMM_RANK(comm, rank, ierr)
\mathbf{2}
     IF (rank .EQ. 0) THEN
3
         CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
4
         CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
5
     ELSE IF (rank .EQ. 1) THEN
6
         CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
7
         CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
8
     END IF
9
     The receive operation of the first process must complete before its send, and can complete
10
     only if the matching send of the second processor is executed. The receive operation of the
11
     second process must complete before its send and can complete only if the matching send
12
     of the first process is executed. This program will always deadlock. The same holds for any
13
     other send mode.
14
15
     Example 3.9 An exchange that relies on buffering.
16
17
     CALL MPI_COMM_RANK(comm, rank, ierr)
18
     IF (rank .EQ. 0) THEN
19
         CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
20
         CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
21
     ELSE IF (rank .EQ. 1) THEN
22
         CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
23
         CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
^{24}
     END IF
25
     The message sent by each process has to be copied out before the send operation returns
26
     and the receive operation starts. For the program to complete, it is necessary that at least
27
     one of the two messages sent be buffered. Thus, this program can succeed only if the
28
     communication system can buffer at least count words of data.
29
30
           Advice to users. When standard send operations are used, then a deadlock situation
^{31}
           may occur where both processes are blocked because buffer space is not available. The
32
           same will certainly happen, if the synchronous mode is used. If the buffered mode is
33
           used, and not enough buffer space is available, then the program will not complete
34
           either. However, rather than a deadlock situation, we shall have a buffer overflow
35
           error.
36
37
           A program is "safe" if no message buffering is required for the program to complete.
38
           One can replace all sends in such program with synchronous sends, and the pro-
39
           gram will still run correctly. This conservative programming style provides the best
40
           portability, since program completion does not depend on the amount of buffer space
41
           available or on the communication protocol used.
42
           Many programmers prefer to have more leeway and opt to use the "unsafe" program-
43
           ming style shown in Example 3.9. In such cases, the use of standard sends is likely
44
           to provide the best compromise between performance and robustness: quality imple-
45
           mentations will provide sufficient buffering so that "common practice" programs will
46
           not deadlock. The buffered send mode can be used for programs that require more
47
           buffering, or in situations where the programmer wants more control. This mode
48
```

might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

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

## 3.6 Buffer Allocation and Usage

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

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

IN	buffer	initial buffer address (choice)	17
IN	size	buffer size, in bytes (non-negative integer)	18
			19
C bind	ling		20
	I_Buffer_attach(void *1	buffer, int size)	21
			22
int MP	L_Buffer_attach_c(void	*buffer, MPI_Count size)	23
Fortra	n 2008 binding		24
MPI_Bu:	ffer_attach(buffer, siz	ze, ierror)	25 26
	<pre>PE(*), DIMENSION(), A</pre>		20
	TEGER, INTENT(IN) :: s:		28
IN	TEGER, OPTIONAL, INTEN	Γ(OUT) :: ierror	29
MPI_Bu	ffer_attach(buffer, siz	ze, ierror)	30
TY	PE(*), DIMENSION(), A	ASYNCHRONOUS :: buffer	31
IN	TEGER(KIND=MPI_COUNT_K	IND), INTENT(IN) :: size	32
IN	TEGER, OPTIONAL, INTEN	Γ(OUT) :: ierror	33
Fortra	n binding		34
	FFER_ATTACH(BUFFER, SIZ	ZE. TEBROR)	35
	<pre>ype&gt; BUFFER(*)</pre>		36
	TEGER SIZE, IERROR		37
			38
Pro	ovides to IVIPI a burner in th	he user's memory to be used for buffering outgoing mes-	39

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 19.1.12).

 $\mathbf{2}$ 

```
1
     MPI_BUFFER_DETACH(buffer_addr, size)
2
       OUT
                 buffer_addr
                                            initial buffer address (choice)
3
       OUT
                 size
                                            buffer size, in bytes (integer)
4
5
6
     C binding
7
     int MPI_Buffer_detach(void *buffer_addr, int *size)
8
     int MPI_Buffer_detach_c(void *buffer_addr, MPI_Count *size)
9
10
     Fortran 2008 binding
11
     MPI_Buffer_detach(buffer_addr, size, ierror)
12
          USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
          TYPE(C_PTR), INTENT(OUT) :: buffer_addr
13
14
          INTEGER, INTENT(OUT) :: size
15
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_Buffer_detach(buffer_addr, size, ierror)
17
          USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
18
          TYPE(C_PTR), INTENT(OUT) :: buffer_addr
19
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
20
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     Fortran binding
23
     MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
24
          <type> BUFFER_ADDR(*)
25
          INTEGER SIZE, IERROR
26
         Detach the buffer currently associated with MPI. The call returns the address and the
27
     size of the detached buffer. This operation will block until all messages currently in the
28
     buffer have been transmitted. Upon return of this function, the user may reuse or deallocate
29
     the space taken by the buffer.
30
         If the size of the detached buffer cannot be represented in size, it is set to
^{31}
     MPI_UNDEFINED.
32
33
     Example 3.10 Calls to attach and detach buffers.
34
35
     #define BUFFSIZE 10000
36
     int size;
37
     char *buff;
38
     MPI_Buffer_attach(malloc(BUFFSIZE), BUFFSIZE);
39
     /* a buffer of 10000 bytes can now be used by MPI_Bsend */
40
     MPI_Buffer_detach(&buff, &size);
41
     /* Buffer size reduced to zero */
42
     MPI_Buffer_attach(buff, size);
     /* Buffer of 10000 bytes available again */
43
44
45
           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
46
47
          used differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address
48
          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 9.1 about the use of C\_PTR pointers. (*End of advice to users.*)

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

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

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

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

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

## 3.6.1 Model Implementation of Buffered Mode

The model implementation uses the packing and unpacking functions described in Section 5.2 and the nonblocking communication functions described in Section 3.7.

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

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

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

Unofficial Draft for Comment Only

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

• Compute the number, n, of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function MPI\_PACK\_SIZE(count, datatype, comm, size), with the count, datatype and comm

arguments used in the MPI\_BSEND call, returns an upper bound on the amount of space needed to buffer the message data (see Section 5.2). The MPI constant MPI\_BSEND\_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).

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

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

• Return

17 18 19

20

## 3.7 Nonblocking Communication

21Nonblocking communication is important both for reasons of correctness and perfor-22 mance. For complex communication patterns, the use of only blocking communication 23(without buffering) is difficult because the programmer must ensure that each send is  $^{24}$ matched with a receive in an order that avoids *deadlock*. For communication patterns that 25are determined only at run time, this is even more difficult. Nonblocking communication 26can be used to avoid this problem, allowing programmers to express complex and possibly 27dynamic communication patterns without needing to ensure that all sends and receives 28 are issued in an order that prevents deadlock (see Section 3.5 and the discussion of "safe" 29programs). Nonblocking communication also allows for the overlap of communication with 30 different communication operations, e.g., to prevent the *serialization* of such operations,  $^{31}$ and for the *overlap* of communication with computation. Whether an implementation is 32 able to accomplish an effective (from a performance standpoint) overlap of operations de-33 pends on the implementation itself and the system on which the implementation is running. 34Using nonblocking operations *permits* an implementation to overlap communication with 35 computation, but does not require it to do so.

36 A nonblocking send start call initiates the send operation, but does not complete it. 37 The send start call can return before the message was copied out of the send buffer. A 38 separate send complete call is needed to complete the communication, i.e., to verify that 39 the data has been copied out of the send buffer. With suitable hardware, the transfer of data 40 out of the sender memory may proceed concurrently with computations done at the sender 41 after the send was initiated and before it completed. Similarly, a nonblocking receive start 42call initiates the receive operation, but does not complete it. The call can return before a 43 message is stored into the receive buffer. A separate **receive complete** call is needed to 44complete the receive operation and verify that the data has been received into the receive 45buffer. With suitable hardware, the transfer of data into the receiver memory may proceed 46concurrently with computations done after the receive was initiated and before it completed. 47The use of nonblocking receives may also avoid system buffering and memory-to-memory 48 copying, as information is provided early on the location of the receive buffer.

## Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11 12

13

14 15

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

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

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

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

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

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

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

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

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

 $47 \\ 48$ 

 $\mathbf{2}$ 

 $\mathbf{5}$ 

1	3.7.1 Cor	mmunication Request Objects	5			
2 3 4 5 6 7 8 9	Nonblocking communications use opaque <b>request</b> objects to identify communication oper- ations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the com- munication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.					
10 11	3.7.2 Communication Initiation					
12 13 14 15 16 17	For the functions defined in this section, we use the same naming conventions as for blocking communication: a prefix of B, S, or R is used for <b>buffered</b> , <b>synchronous</b> or <b>ready</b> mode. In addition, for these functions a prefix of I (for <b>immediate</b> and <b>incomplete</b> ) indicates that the call is nonblocking.					
18	MPI_ISENI	D(buf, count, datatype, dest, ta	ag, comm, request)			
19	IN	buf	initial address of send buffer (choice)			
20 21 22	IN	count	number of elements in send buffer (non-negative integer)			
23	IN	datatype	datatype of each send buffer element (handle)			
24	IN	dest	rank of destination (integer)			
25 26	IN	tag	message tag (integer)			
27	IN	comm	communicator (handle)			
28 29	OUT	request	communication request (handle)			
30	C binding	,				
31	C binding int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,					
32 33	int tag, MPI_Comm comm, MPI_Request *request)					
34	<pre>int MPI_Isend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>					
35	int dest, int tag, MPI_Comm comm, MPI_Request *request)					
36	Fortran 2008 binding					
37 38	MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)					
39	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf					
40	INTEGER, INTENT(IN) :: count, dest, tag					
41	TYPE(MPI_Datatype), INTENT(IN) :: datatype					
42	TYPE(MPI_Comm), INTENT(IN) :: comm					
43	TYPE(MPI_Request), INTENT(OUT) :: request					
44	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
45		• 1	est, tag, comm, request, ierror)			
46 47	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf					
48	INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype					
	1111(	······································	··· acourte			

Unofficial Draft for Comment Only

TYP TYP	INTEGER, INTENT(IN) :: dest, tag TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
			5		
	binding		6		
		TYPE, DEST, TAG, COMM, REQUEST, IERROR)	7		
•	pe> BUF(*)	E, DEST, TAG, COMM, REQUEST, IERROR	8		
1111	LGER COONI, DATAIF	E, DESI, ING, COMM, REQUESI, IERROR	9		
Star	rt a standard mode, no	nblocking send.	10		
			11 12		
MPI IBS	END(buf. count. dataty	pe, dest, tag, comm, request)	12		
IN	buf		14		
		initial address of send buffer (choice)	15		
IN	count	number of elements in send buffer (non-negative	16		
		integer)	17		
IN	datatype	datatype of each send buffer element (handle)	18		
IN	dest	rank of destination (integer)	19		
IN	tag	message tag (integer)	20		
IN	comm	communicator (handle)	21 22		
			22		
OUT	request	communication request (handle)	24		
			25		
C bindi		*buf, int count, MPI_Datatype datatype, int dest,	26		
IIIC MFI		Comm comm, MPI_Request *request)	27		
	_		28		
int MPI		d *buf, MPI_Count count, MPI_Datatype datatype,	29		
	int dest, int	tag, MPI_Comm comm, MPI_Request *request)	30		
Fortran	2008 binding		31		
MPI_Ibs	end(buf, count, dat	atype, dest, tag, comm, request, ierror)	32		
TYP	E(*), DIMENSION()	, INTENT(IN), ASYNCHRONOUS :: buf	33 34		
· · · · · · · · · · · · · · · · · · ·	EGER, INTENT(IN) ::	-	35		
		TENT(IN) :: datatype	36		
TYPE(MPI_Comm), INTENT(IN) :: comm			37		
	E(MPI_Request), INT		38		
TNT	EGER, OPTIONAL, INT	ENI(UUI) :: lerror	39		
		atype, dest, tag, comm, request, ierror)	40		
	-	, INTENT(IN), ASYNCHRONOUS :: buf	41		
		_KIND), INTENT(IN) :: count	42		
	• -	TENT(IN) :: datatype	43		
	EGER, INTENT(IN) ::	6	44 45		
	TYPE(MPI_Comm), INTENT(IN) :: comm				
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
1111	, 0. 110,000, 101		48		

```
1
     Fortran binding
\mathbf{2}
     MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
3
         <type> BUF(*)
4
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
5
         Start a buffered mode, nonblocking send.
6
7
8
     MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
9
       IN
                buf
                                            initial address of send buffer (choice)
10
11
       IN
                count
                                            number of elements in send buffer (non-negative
                                           integer)
12
13
       IN
                datatype
                                           datatype of each send buffer element (handle)
14
                dest
                                           rank of destination (integer)
       IN
15
16
       IN
                                           message tag (integer)
                tag
17
       IN
                comm
                                           communicator (handle)
18
       OUT
                                           communication request (handle)
                request
19
20
     C binding
21
     int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest,
22
                    int tag, MPI_Comm comm, MPI_Request *request)
23
24
     int MPI_Issend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
25
                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
26
     Fortran 2008 binding
27
     MPI_Issend(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
35
     MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
36
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
37
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         INTEGER, INTENT(IN) :: dest, tag
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         TYPE(MPI_Request), INTENT(OUT) :: request
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
     Fortran binding
44
     MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
45
         <type> BUF(*)
46
47
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
48
```

Start a synchronous mode, nonblocking send.

'I_IRSE	END(buf, count, dataty	/pe, dest, tag, comm, request)
N	buf	initial address of send buffer (choice)
N	count	number of elements in send buffer (non-negative integer)
N	datatype	datatype of each send buffer element (handle)
N	dest	rank of destination (integer)
N	tag	message tag (integer)
N	comm	communicator (handle)
DUT	request	communication request (handle)
	- 1	
bindin	ıg	
t MPI_		*buf, int count, MPI_Datatype datatype, int dest,
	int tag, MPI	_Comm comm, MPI_Request *request)
t MPI_		d *buf, MPI_Count count, MPI_Datatype datatype,
	int dest, int	t tag, MPI_Comm comm, MPI_Request *request)
rtran	2008 binding	
	•	atype, dest, tag, comm, request, ierror)
TYPE	(*), DIMENSION()	, INTENT(IN), ASYNCHRONOUS :: buf
	(*), DIMENSION() GER, INTENT(IN) ::	
INTE	GER, INTENT(IN) ::	
INTE TYPE	GER, INTENT(IN) ::	count, dest, tag ITENT(IN) :: datatype
INTE TYPE TYPE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT	count, dest, tag ITENT(IN) :: datatype
INTE TYPE TYPE TYPE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT	count, dest, tag MTENT(IN) :: datatype T(IN) :: comm
INTE TYPE TYPE TYPE INTE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT	<pre>count, dest, tag ITENT(IN) :: datatype T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror</pre>
INTE TYPE TYPE TYPE INTE I_Irse	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat	<pre>count, dest, tag TTENT(IN) :: datatype T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror TENT(OUT) :: ierror</pre>
INTE TYPE TYPE TYPE INTE I_Irse TYPE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION()	<pre>count, dest, tag ITENT(IN) :: datatype '(IN) :: comm EENT(OUT) :: request EENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf</pre>
INTE TYPE TYPE INTE I_Irse TYPE INTE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT	<pre>count, dest, tag TENT(IN) :: datatype "(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf "_KIND), INTENT(IN) :: count</pre>
INTE TYPE TYPE INTE INTE ILTSE TYPE INTE TYPE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN	<pre>count, dest, tag TENT(IN) :: datatype T(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf T_KIND), INTENT(IN) :: count ITENT(IN) :: datatype</pre>
INTE TYPE TYPE INTE INTE ILISE TYPE INTE TYPE INTE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) ::	<pre>count, dest, tag ITENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror satype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count ITENT(IN) :: datatype dest, tag</pre>
INTE TYPE TYPE INTE INTE INTE INTE TYPE INTE TYPE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT	<pre>count, dest, tag ITENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count ITENT(IN) :: datatype dest, tag '(IN) :: comm</pre>
INTE TYPE TYPE TYPE INTE I_Irse TYPE INTE TYPE INTE TYPE TYPE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT	<pre>count, dest, tag ITENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count ITENT(IN) :: datatype dest, tag '(IN) :: comm TENT(OUT) :: request</pre>
INTE TYPE TYPE TYPE INTE I_Irse TYPE INTE TYPE INTE TYPE TYPE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT	<pre>count, dest, tag ITENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count ITENT(IN) :: datatype dest, tag '(IN) :: comm</pre>
INTE TYPE TYPE INTE INTE INTE INTE TYPE INTE TYPE TYPE INTE INTE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT	<pre>count, dest, tag ITENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count ITENT(IN) :: datatype dest, tag '(IN) :: comm TENT(OUT) :: request</pre>
INTE TYPE TYPE INTE INTE TYPE INTE TYPE TYPE INTE INTE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT binding	<pre>count, dest, tag ITENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count ITENT(IN) :: datatype dest, tag '(IN) :: comm TENT(OUT) :: request</pre>
INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE TYPE INTE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT binding	<pre>count, dest, tag HTENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf T_KIND), INTENT(IN) :: count HTENT(IN) :: datatype dest, tag '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror</pre>
INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE INTE INTE INTE INTE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT binding ND(BUF, COUNT, DAT e> BUF(*)	<pre>count, dest, tag HTENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf T_KIND), INTENT(IN) :: count HTENT(IN) :: datatype dest, tag '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror</pre>
INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE INTE CTTAN INTE	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT binding ND(BUF, COUNT, DAT e> BUF(*) GER COUNT, DATATYF	<pre>count, dest, tag TTENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count TENT(IN) :: datatype dest, tag '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR</pre>
INTE TYPE TYPE INTE INTE INTE TYPE INTE TYPE INTE INTE Ctran _IRSE <typ INTE</typ 	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT binding ND(BUF, COUNT, DAT e> BUF(*)	<pre>count, dest, tag TTENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count TENT(IN) :: datatype dest, tag '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR</pre>
INTE TYPE TYPE INTE INTE TYPE INTE TYPE INTE TYPE INTE Ctran _IRSE <typ INTE</typ 	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT binding ND(BUF, COUNT, DAT e> BUF(*) GER COUNT, DATATYF	<pre>count, dest, tag TTENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count TENT(IN) :: datatype dest, tag '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR</pre>
INTE TYPE TYPE INTE INTE TYPE INTE TYPE INTE TYPE INTE INTE tran _IRSE <typ INTE</typ 	GER, INTENT(IN) :: (MPI_Datatype), IN (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT nd(buf, count, dat (*), DIMENSION() GER(KIND=MPI_COUNT (MPI_Datatype), IN GER, INTENT(IN) :: (MPI_Comm), INTENT (MPI_Request), INT GER, OPTIONAL, INT binding ND(BUF, COUNT, DAT e> BUF(*) GER COUNT, DATATYF	<pre>count, dest, tag TTENT(IN) :: datatype '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: ierror catype, dest, tag, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: buf '_KIND), INTENT(IN) :: count TENT(IN) :: datatype dest, tag '(IN) :: comm TENT(OUT) :: request TENT(OUT) :: request TENT(OUT) :: ierror TATYPE, DEST, TAG, COMM, REQUEST, IERROR) PE, DEST, TAG, COMM, REQUEST, IERROR</pre>

1 MPI\_IRECV(buf, count, datatype, source, tag, comm, request)  $\mathbf{2}$ OUT buf initial address of receive buffer (choice) 3 IN count number of elements in receive buffer (non-negative 4 integer) 56 IN datatype datatype of each receive buffer element (handle) 7 rank of source or MPI\_ANY\_SOURCE (integer) IN source 8 IN tag message tag or MPI\_ANY\_TAG (integer) 9 10 IN comm communicator (handle) 11 OUT communication request (handle) request 1213 C binding 14 int MPI\_Irecv(void \*buf, int count, MPI\_Datatype datatype, int source, 15int tag, MPI\_Comm comm, MPI\_Request \*request) 1617 int MPI\_Irecv\_c(void \*buf, MPI\_Count count, MPI\_Datatype datatype, 18 int source, int tag, MPI\_Comm comm, MPI\_Request \*request) 19 Fortran 2008 binding 20MPI\_Irecv(buf, count, datatype, source, tag, comm, request, ierror) 21TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 22 INTEGER, INTENT(IN) :: count, source, tag 23 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 24TYPE(MPI\_Comm), INTENT(IN) :: comm 25TYPE(MPI\_Request), INTENT(OUT) :: request 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728MPI\_Irecv(buf, count, datatype, source, tag, comm, request, ierror) 29TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 30 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 31TYPE(MPI\_Datatype), INTENT(IN) :: datatype 32 INTEGER, INTENT(IN) :: source, tag 33 TYPE(MPI\_Comm), INTENT(IN) :: comm 34 TYPE(MPI\_Request), INTENT(OUT) :: request 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 Fortran binding 37 MPI\_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 38 <type> BUF(\*) 39 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR 40 41 Start a nonblocking receive. 4243 44 4546 47 48

MPI\_ISENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, 2 source, recvtag, comm, request) 3 IN sendbuf initial address of send buffer (choice) IN sendcount number of elements in send buffer (non-negative 5 integer) 6 IN sendtype datatype of each send buffer element (handle) IN dest rank of destination (integer) 9 IN sendtag send tag (integer) 10 11 OUT recvbuf initial address of receive buffer (choice) 12IN recvcount number of elements in receive buffer (non-negative 13 integer) 14IN datatype of each receive buffer element (handle) recvtype 15rank of source or MPI\_ANY\_SOURCE (integer) 16IN source 17receive tag or MPI\_ANY\_TAG (integer) IN recvtag 18 IN communicator (handle) comm 19 OUT request communication request (handle) 202122 C binding 23int MPI\_Isendrecv(const void \*sendbuf, int sendcount,  $^{24}$ MPI\_Datatype sendtype, int dest, int sendtag, void \*recvbuf, 25int recvcount, MPI\_Datatype recvtype, int source, int recvtag, 26MPI\_Comm comm, MPI\_Request \*request) 27Fortran 2008 binding 28MPI\_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, 29 recvcount, recvtype, source, recvtag, comm, request, ierror) 30 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 31INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, 32 recvtag 33 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 34 TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 35TYPE(MPI\_Comm), INTENT(IN) :: comm 36 TYPE(MPI\_Request), INTENT(OUT) :: request 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 Fortran binding 40 MPI\_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, 41 RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR) 42

## <type> SENDBUF(\*), RECVBUF(\*) INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR

Initiate a nonblocking communication request for a send and receive operation.

43

44

45

1 MPI\_ISENDRECV\_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm,  $\mathbf{2}$ request) 3 INOUT buf initial address of send and receive buffer (choice) 4 IN number of elements in send and receive buffer count 5(non-negative integer) 6 7 IN datatype type of elements in send and receive buffer (handle) 8 IN dest rank of destination (integer) 9 IN sendtag send message tag (integer) 10 11 rank of source or MPI\_ANY\_SOURCE (integer) IN source 12IN recvtag receive message tag or MPI\_ANY\_TAG (integer) 13 IN comm communicator (handle) 1415OUT request communication request (handle) 1617C binding 18 int MPI\_Isendrecv\_replace(void \*buf, int count, MPI\_Datatype datatype, 19 int dest, int sendtag, int source, int recvtag, MPI\_Comm comm, 20MPI\_Request \*request) 21Fortran 2008 binding 22 MPI\_Isendrecv\_replace(buf, count, datatype, dest, sendtag, source, recvtag, 23comm, request, ierror) 24TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 25INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag 26TYPE(MPI\_Datatype), INTENT(IN) :: datatype 27TYPE(MPI\_Comm), INTENT(IN) :: comm 28TYPE(MPI\_Request), INTENT(OUT) :: request 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30  $^{31}$ Fortran binding 32 MPI\_ISENDRECV\_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, 33 COMM, REQUEST, IERROR) 34 <type> BUF(\*) 35 INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST, 36 IERROR 37 Initiate a nonblocking communication request for a send and receive operation. The 38 same buffer is used both for the send and for the receive, so that the message sent is replaced 39 by the message received. 40 These calls allocate a communication request object and associate it with the request 41 handle (the argument request). The request can be used later to query the status of the 42communication or wait for its completion. 43 A nonblocking send call indicates that the system may start copying data out of the 44 send buffer. The sender should not modify any part of the send buffer after a nonblocking 45send operation is called, until the send completes. 464748

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10–19.1.20. (End of advice to users.)

#### 3.7.3 Communication Completion

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

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

We shall use the following terminology: A **null handle** is a handle with value MPI\_REQUEST\_NULL. A persistent request and the handle to it are **inactive** if the request is not associated with any ongoing communication (see Section 3.9). A handle is **active** if it is neither null nor inactive. An **empty** status is a status which is set to return **tag** = MPI\_ANY\_TAG, source = MPI\_ANY\_SOURCE, error = MPI\_SUCCESS, and is also internally configured so that calls to MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and MPI\_GET\_ELEMENTS\_X return count = 0 and MPI\_TEST\_CANCELLED returns false. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

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

Error codes belonging to the error class MPI\_ERR\_IN\_STATUS should be returned only by the MPI completion functions that take arrays of MPI\_Status. For the functions 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)INOUTrequestOUTstatusstatusstatus object (Status)

C binding int MPI\_Wait(MPI\_Request \*request, MPI\_Status \*status)

## Unofficial Draft for Comment Only

1

2

3

4

 $\mathbf{5}$ 

6

7 8 9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39 40 41

42

43

44 45 46

47

1	Fortron 2008 binding			
2	Fortran 2008 binding MPI_Wait(request, status, ierror)			
3	TYPE(MPI_Request), INTENT(INOUT) :: request			
4	TYPE(MPI_Request), INTENT(INDOT) :: request TYPE(MPI_Status) :: status			
5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
6				
7	Fortran binding			
8	MPI_WAIT(REQUEST, STATUS, IERROR)			
9	INTEGER REQUEST, STATUS(MPI_S	TATUS_SIZE), IERRUR		
10	A call to $MPI_WAIT$ returns when the	ne operation identified by <b>request</b> is complete. If the		
11	request is an active persistent request, it is marked inactive. Any other type of request is			
12	deallocated and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local			
13	operation.			
14	The call returns, in status, information on the completed operation. The content of			
15	0 I	can be accessed as described in Section 3.2.5. The		
16 17	status object for a send operation may be queried by a call to MPI_TEST_CANCELLED			
17	(see Section 3.8).			
19		th a null or inactive request argument. In this case		
20	the operation returns immediately with	empty status.		
21	Advice to users. Successful retur	n of MPI_WAIT after a MPI_IBSEND implies that		
22	the user send buffer can be reused—i.e., data has been sent out or copied into a buffer			
23	attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer			
24	cancel the send (see Section 3.8). If a matching receive is never posted, then the buffer			
25	cannot be freed. This runs somewhat counter to the stated goal of MPI_CANCEL			
26	(always being able to free program space that was committed to the communication			
27	subsystem). (End of advice to users.)			
28	Advise to implementary. In a mult	ithreaded environment, a call to MPI_WAIT should		
29				
30 31		block only the calling thread, allowing the thread scheduler to schedule another thread for execution. ( <i>End of advice to implementors.</i> )		
32				
33				
34	MPI_TEST(request, flag, status)			
35	· · · · · ,			
36	INOUT request	communication request (handle)		
37	OUT flag	true if operation completed (logical)		
38	OUT status	status object (Status)		
39 40				
40 41	C binding			
42	<pre>int MPI_Test(MPI_Request *request)</pre>	, int *flag, MPI_Status *status)		
43	Fortran 2008 binding			
44	MPI_Test(request, flag, status, id	error)		
45	TYPE(MPI_Request), INTENT(INOU			
46	LOGICAL, INTENT(OUT) :: flag			
47	TYPE(MPI_Status) :: status			
48	INTEGER, OPTIONAL, INTENT(OUT)	) :: ierror		

## Unofficial Draft for Comment Only

	_
Fortran binding	1
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)	2
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR	3
LOGICAL FLAG	4
A call to MPI_TEST returns flag = true if the operation identified by request is complet	5 e.
In such a case, the status object is set to contain information on the completed operation	n
If the request is an active persistent request, it is marked as inactive. Any other type	of
request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call return	ne
flag = false if the operation identified by request is not complete. In this case, the value	of
the status object is undefined. MPI_TEST is a local operation.	10
The return status object for a receive operation carries information that can be accessed	ed 10
as described in Section 3.2.5. The status object for a send operation carries information	n 12
that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).	10
One is allowed to call MPI_TEST with a null or inactive request argument. In such	14 a 15
case the operation returns with $flag = true$ and empty status.	15
The functions MPI_WAIT and MPI_TEST can be used to complete both sends an	
receives.	18
	10
Advice to users. The use of the nonblocking MPI_TEST call allows the user t	to 20
schedule alternative activities within a single thread of execution. An event-drive	en 20 21
thread scheduler can be emulated with periodic calls to MPI_TEST. (End of advice	to 22
users.)	23
	24
<b>Example 3.11</b> Simple usage of nonblocking operations and MPI_WAIT.	25
Example 5.11 Shiple usage of honolocking operations and with _wwith.	26
CALL MPI_COMM_RANK(comm, rank, ierr)	27
IF (rank .EQ. 0) THEN	28
CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)	29
**** do some computation to mask latency ****	30
CALL MPI_WAIT(request, status, ierr)	31
ELSE IF (rank .EQ. 1) THEN	32
CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)	33
**** do some computation to mask latency ****	34
CALL MPI_WAIT(request, status, ierr)	35
END IF	36
A request object can be freed using the following MPI procedure.	37
If request object can be need using the following this procedure.	38
	39
MPI_REQUEST_FREE(request)	40
INOUT request communication request (handle)	41
	42
C binding	43
int MPI_Request_free(MPI_Request *request)	44
INT IN I_MOQUEDU_IIEE(IN I_MEQUEDU TEQUEDU)	45
Fortran 2008 binding	46
MPI_Request_free(request, ierror)	47
TYPE(MPI_Request), INTENT(INOUT) :: request	48

**Unofficial Draft for Comment Only** 

1	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2	
3	Fortran binding MPI_REQUEST_FREE(REQUEST, IERROR)
4	
5	INTEGER REQUEST, IERROR
6	MPI_REQUEST_FREE is a local operation. Upon successful return,
7	$MPI_REQUEST_FREE$ sets request to $MPI_REQUEST_NULL$ . For an inactive
8	$request \ representing \ any \ type \ of \ MPI \ operation, \ MPI\_REQUEST\_FREE \ shall \ do \ the \ freeing$
9	stage of the associated operation during its execution.
10	For a request representing a nonblocking point-to-point or a persistent point-to-point
11	operation, it is permitted (although strongly discouraged) to call MPI_REQUEST_FREE
12	when the request is active. In this special case, MPI_REQUEST_FREE will only mark the
13	request for freeing and MPI will actually do the freeing stage of the associated operation
14	later.
15	The use of this routine for generalized requests is described in Section 13.2.
16 17	Calling MPI_REQUEST_FREE with an active request representing any other type of
18	MPI operation (e.g., any partitioned operation (see Chapter 4), any collective operation
19	(see Chapter 6), any I/O operation (see Chapter 14), or any request-based RMA operation
20	(see Chapter 12)) is erroneous.
21	Rationale. For point-to-point operations, the MPI_REQUEST_FREE mechanism is
22	provided for reasons of performance and convenience on the sending side. (End of
23	rationale.)
24	
25	Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not
26	possible to check for the successful completion of the associated communication with
27	calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the
28	communication, an error code cannot be returned to the user—such an error must be
29	treated as fatal. An active receive request should never be freed as the receiver will
30	have no way to verify that the receive has completed and the receive buffer can be
31	reused. (End of advice to users.)
32	
33	<b>Example 3.12</b> An example using MPI_REQUEST_FREE.
34	
35	CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
36	IF (rank .EQ. 0) THEN
37	DO i=1,n
38 39	CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
39 40	CALL MPI_REQUEST_FREE(req, ierr)
40	CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
41	CALL MPI_WAIT(req, status, ierr)
43	END DO ELSE IF (rank .EQ. 1) THEN
44	CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
45	CALL MPI_INLEV(INVAI, I, MPI_NEAL, 0, 0, MPI_COMM_WORLD, Teq, TeTT) CALL MPI_WAIT(req, status, ierr)
46	DO I=1,n-1
47	CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
48	CALL MPI_REQUEST_FREE(req, ierr)

Unofficial Draft for Comment Only

```
CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
CALL MPI_WAIT(req, status, ierr)
END DO
CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
CALL MPI_WAIT(req, status, ierr)
END IF
```

#### 3.7.4 Semantics of Nonblocking Communications

The semantics of nonblocking communication is defined by suitably extending the definitions in Section 3.5.

**Order** Nonblocking communication operations are ordered according to the execution order of the calls that initiate the communication. The non-overtaking requirement of Section 3.5 is extended to nonblocking communication, with this definition of order being used.

**Example 3.13** Message ordering for nonblocking operations.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK .EQ. 0) THEN
CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1, status, ierr)
CALL MPI_WAIT(r2, status, ierr)
```

The first send of process zero will match the first receive of process one, even if both messages are sent before process one executes either receive.

**Progress** A call to MPI\_WAIT that completes a receive will eventually terminate and return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is nonblocking, then the receive should complete even if no call is executed by the sender to complete the send. Similarly, a call to MPI\_WAIT that completes a send will eventually return if a matching receive has been started, unless the receive is satisfied by another send, and even if no call is executed to complete the receive.

**Example 3.14** An illustration of progress semantics.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                    40
                                                                                    41
IF (RANK .EQ. 0) THEN
                                                                                    42
   CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
   CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
                                                                                    43
                                                                                    44
ELSE IF (rank .EQ. 1) THEN
   CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
                                                                                    45
   CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
                                                                                    46
                                                                                    47
   CALL MPI_WAIT(r, status, ierr)
                                                                                    48
END IF
```

2

9

10

11 12

13

14

15 16

17 18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30 31

32

33

34

35

36

37 38

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.

<sup>5</sup> If an MPI\_TEST that completes a receive is repeatedly called with the same arguments, <sup>6</sup> and a matching send has been started, then the call will eventually return flag = true, unless <sup>7</sup> the send is satisfied by another receive. If an MPI\_TEST that completes a send is repeatedly <sup>8</sup> called with the same arguments, and a matching receive has been started, then the call will <sup>9</sup> eventually return flag = true, unless the receive is satisfied by another send.

10 11 12

#### 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.

19 20

21

48

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

21			
22	IN	count	list length (non-negative integer)
23	INOUT	array_of_requests	array of requests (array of handles)
24 25 26	OUT	index	index of handle for operation that completed (integer)
27	OUT	status	status object (Status)
28 29	C binding	-	
30 31	int MPI_W	Altany(int count, M MPI_Status *sta	<pre>PI_Request array_of_requests[], int *index, atus)</pre>
32 33	Fortran 2	008 binding	
34		ny(count, array_of_ ER, INTENT(IN) :: c	requests, index, status, ierror) count
35 36	TYPE(	MPI_Request), INTEN	T(INOUT) :: array_of_requests(count)
37 38		ER, INTENT(OUT) :: MPI_Status) :: stat	
39	INTEG	ER, OPTIONAL, INTEN	T(OUT) :: ierror
40	Fortran b	oinding	
41	MPI_WAITA	NY(COUNT, ARRAY_OF_	REQUESTS, INDEX, STATUS, IERROR)
42	INTEG	ER COUNT, ARRAY_OF_	<pre>REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),</pre>
43		IERROR	
44	Blocks	s until one of the opera	tions associated with the active requests in the array has
45		-	eration is enabled and can terminate, one is arbitrarily
46 47	-	•	ex of that request in the array and returns in status the
41			- v

status of the completing operation. (The array is indexed from zero in C, and from one in

Fortran.) If the request is an active persistent request, it is marked inactive. Any other type of request is deallocated and the request handle is set to MPI\_REQUEST\_NULL.

The array\_of\_requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns immediately with index = MPI\_UNDEFINED, and an empty status.

The execution of MPI\_WAITANY with an array containing multiple entries has the same effect as the execution of MPI\_WAIT with the array entry indicated by the output value of index (unless the output value of index is MPI\_UNDEFINED). MPI\_WAITANY with an array containing one active entry is equivalent to MPI\_WAIT.

MPI_TESTANY(count, array_of_requests, index, flag, status)			
IN	count	list length (non-negative integer)	
INOUT	array_of_requests	array of requests (array of handles)	
OUT	index	index of operation that completed or MPI_UNDEFINED if none completed (integer)	
OUT	flag	true if one of the operations is complete (logical)	
OUT	status	status object (Status)	

#### C binding

```
Fortran 2008 binding
MPI_Testany(count, array_of_requests, index, flag, status, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
INTEGER, INTENT(OUT) :: index
LOGICAL, INTENT(OUT) :: flag
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
Fortran binding
```

```
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
IERROR
LOCICAL FLAG
```

LOGICAL FLAG

Tests for completion of either one or none of the operations associated with active handles. In the former case, it returns flag = true, returns in index the index of this request in the array, and returns in status the status of that operation. If the request is an active 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 flag = true, index = MPI\_UNDEFINED, and an empty status. 48

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

<sup>1</sup> If the array of requests contains active handles then the execution of MPI\_TESTANY <sup>2</sup> has the same effect as the execution of MPI\_TEST with each of the array elements in some <sup>3</sup> arbitrary order, until one call returns flag = true, or all fail. In the former case, index is <sup>4</sup> set to indicate which array element returned flag = true and in the latter case, it is set to <sup>5</sup> MPI\_UNDEFINED. MPI\_TESTANY with an array containing one active entry is equivalent to <sup>6</sup> MPI\_TEST.

MPI\_WAITALL(count, array\_of\_requests, array\_of\_statuses)

```
10
       IN
                 count
                                            lists length (non-negative integer)
11
       INOUT
                array_of_requests
                                            array of requests (array of handles)
12
       OUT
                array_of_statuses
                                            array of status objects (array of Status)
13
14
     C binding
15
     int MPI_Waitall(int count, MPI_Request array_of_requests[],
16
17
                    MPI_Status array_of_statuses[])
18
     Fortran 2008 binding
19
     MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
20
         INTEGER, INTENT(IN) :: count
21
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
22
         TYPE(MPI_Status) :: array_of_statuses(*)
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
     Fortran binding
26
     MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
27
```

28 29 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 has the same effect as the execution of MPI\_WAIT for each of the array elements in some arbitrary order. MPI\_WAITALL with an array of length one is equivalent to MPI\_WAIT.

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

<sup>42</sup> MPI\_WAITALL will return in such case the error code MPI\_ERR\_IN\_STATUS and will set the <sup>43</sup> error field of each status to a specific error code. This code will be MPI\_SUCCESS, if the <sup>44</sup> specific communication completed; it will be another specific error code, if it failed; or it can <sup>45</sup> be MPI\_ERR\_PENDING if it has neither failed nor completed. The function MPI\_WAITALL <sup>46</sup> will return MPI\_SUCCESS if no request had an error, or will return another error code if it <sup>47</sup> failed for other reasons (such as invalid arguments). In such cases, it will not update the <sup>48</sup> error fields of the statuses.

7 8

*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.*)

		-
TALL (count of your	the floor environment of statuses)	6
IALL(count, array_of_reque	sts, flag, array_of_statuses)	7
count	lists length (non-negative integer)	8
array_of_requests	array of requests (array of handles)	9
flag	(logical)	10
0		11
array_of_statuses	array of status objects (array of Status)	12
		13 14
0		15
		16
MP1_Status array_c	DI_statuses[])	17
Fortran 2008 binding		
MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror) 10		
INTEGER, INTENT(IN) :: count		
		21
	-	22
		23
GER, OPTIONAL, INTENT(O	UT) :: ierror	24
ainding		25
0	UESTS, FLAG, ARRAY OF STATUSES, IERROR)	26
		28
CAL FLAG		29
		30
have completed (this includes the case where no handle in the list is active). In this case, each a status entry, that corresponds to an active request is get to the status of the corresponding		
	<pre>count array_of_requests flag array_of_statuses g Testall(int count, MPI_ MPI_Status array_of 2008 binding all(count, array_of_req BER, INTENT(IN) :: count (MPI_Request), INTENT(I CAL, INTENT(OUT) :: flat (MPI_Status) :: array_of GER, OPTIONAL, INTENT(O Dinding ALL(COUNT, ARRAY_OF_REQ ER COUNT, ARRAY_OF_REQ *), IERROR CAL FLAG ns flag = true if all commu- leted (this includes the case</pre>	<pre>array_of_requests array of requests (array of handles) flag (logical) array_of_statuses array of status objects (array of Status) g GEStall(int count, MPI_Request array_of_requests[], int *flag,</pre>

have completed (this includes the case where no handle in the list is active). In this case, each status entry that corresponds to an active request is set to the status of the corresponding operation. Active persistent requests are marked inactive. Requests of any other type are deallocated and the corresponding handles in the array are set to MPI\_REQUEST\_NULL. Each status entry that corresponds to a null or inactive handle is set to empty.

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

Errors that occurred during the execution of MPI\_TESTALL are handled in the same manner as errors in MPI\_WAITALL.

1 2

3

4 5

33

34

35

36

37

38

39

1 2	MPI_WAIT	SOME(incount, array_of_reque	ests, outcount, array_of_indices, array_of_statuses)
3	IN	incount	length of array_of_requests (non-negative integer)
4 5	INOUT	array_of_requests	array of requests (array of handles)
6	OUT	outcount	number of completed requests (integer)
7 8 9	OUT	array_of_indices	array of indices of operations that completed (array of integers)
9 10 11	Ουτ	array_of_statuses	array of status objects for operations that completed (array of Status)
12	~		
13 14 15 16	C binding int MPI_W	-	
17 18 19 20 21 22 23 24 25	MPI_Waits INTEG TYPE( INTEG TYPE(	array_of_statuses, i ER, INTENT(IN) :: incount	T) :: array_of_requests(incount) nt, array_of_indices(*) statuses(*)
26 27 28 29 30		OME(INCOUNT, ARRAY_OF_REC ARRAY_OF_STATUSES, II ER INCOUNT, ARRAY_OF_REQU	QUESTS, OUTCOUNT, ARRAY_OF_INDICES, ERROR) JESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), PI_STATUS_SIZE, *), IERROR
<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> </ol>	completed. have compl indices of t from zero i array <b>array</b> requests ar and the ass	Returns in outcount the num leted. Returns in the first out chese operations (index within in C and from one in Fortran _of_status the status for these e marked as inactive. Any ot sociated handle is set to MPI_ list contains no active handles	tions associated with active handles in the list have ber of requests from the list array_of_requests that atcount locations of the array array_of_indices the in the array array_of_requests; the array is indexed in). Returns in the first outcount locations of the completed operations. Completed active persistent her type or request that completed is deallocated, REQUEST_NULL. s, then the call returns immediately with outcount
41 42 43 44 45 46 47 48	When it is desiral outcount, a all commu MPI_ERR_II success or	one or more of the communi- ble to return specific informa- rray_of_indices and array_of_s nications that have succeede N_STATUS and the error field to indicate the specific error	cations completed by MPI_WAITSOME fails, then tion on each communication. The arguments statuses will be adjusted to indicate completion of d or failed. The call will return the error code d of each status returned will be set to indicate that occurred. The call will return MPI_SUCCESS vill return another error code if it failed for other

reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

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

			6	
IN	incount	length of array_of_requests (non-negative integer)	7	
INOUT	array_of_requests	array of requests (array of handles)	8 9	
OUT	outcount	number of completed requests (integer)	9 10	
OUT	array_of_indices	array of indices of operations that completed (array	11	
		of integers)	12	
OUT	array_of_statuses	array of status objects for operations that completed	13	
	,	(array of Status)	14 15	
			15 16	
C binding			17	
int MPI_7		_Request array_of_requests[],	18	
	int *outcount, int a		19	
	MPI_Status array_of_		20	
	2008 binding		21	
MPI_Tests	-	quests, outcount, array_of_indices,	22 23	
array_of_statuses, ierror) 23 INTEGER, INTENT(IN) :: incount 24				
			25	
	ER, INTENT(OUT) :: outcom		26	
TYPE (	[MPI_Status] :: array_of_s	statuses(*)	27	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			28	
Fortran binding			29 30	
	MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, 3			
	ARRAY_OF_STATUSES, I	ERROR)	32	
INTEC		UESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),	33	
	ARRAY_OF_STATUSES(MI	PI_STATUS_SIZE, *), IERROR	34	
Behav	es like MPI_WAITSOME, exce	ept that it returns immediately. If no operation has	35	
-	completed it returns $outcount = 0$ . If there is no active handle in the list it returns $outcount$			
$=$ MPI_UN			37 38	
	-	on, which returns immediately, whereas	39	
		imunication completes, if it was passed a list that	40	
	contains at least one active handle. Both calls fulfill a <b>fairness</b> requirement: If a request for a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or			
MPI_TESTSOME, and a matching send has been posted, then the receive will eventually <sup>42</sup>				
<sup>43</sup>				

succeed, unless the send is satisfied by another receive; and similarly for send requests. Errors that occur during the execution of MPI\_TESTSOME are handled as for

MPI\_WAITSOME.

Advice to users. The use of MPI\_TESTSOME is likely to be more efficient than the use 47of MPI\_TESTANY. The former returns information on all completed communications,

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

1  $\mathbf{2}$ 

3 4

5

43

44

45

46

```
1
          with the latter, a new call is required for each communication that completes.
\mathbf{2}
          A server with multiple clients can use MPI_WAITSOME so as not to starve any client.
3
          Clients send messages to the server with service requests. The server calls
4
          MPI_WAITSOME with one receive request for each client, and then handles all receives
5
          that completed. If a call to MPI_WAITANY is used instead, then one client could starve
6
          while requests from another client always sneak in first. (End of advice to users.)
7
8
                                    MPI_TESTSOME should complete as many pending com-
          Advice to implementors.
9
          munications as possible. (End of advice to implementors.)
10
11
     Example 3.15 Client-server code (starvation can occur).
12
13
     CALL MPI_COMM_SIZE(comm, size, ierr)
14
     CALL MPI_COMM_RANK(comm, rank, ierr)
15
     IF (rank .GT. 0) THEN
                                       ! client code
16
        DO WHILE(.TRUE.)
17
            CALL MPI_ISEND(a, n, MPI_REAL, 0, tag,
                                                        comm, request, ierr)
18
            CALL MPI_WAIT(request, status, ierr)
19
        END DO
20
     ELSE
                    ! rank=0 -- server code
21
        DO i=1, size-1
22
            CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
23
                             comm, request_list(i), ierr)
24
        END DO
25
        DO WHILE(.TRUE.)
26
            CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
27
            CALL DO_SERVICE(a(1, index)) ! handle one message
28
            CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag, &
29
                             comm, request_list(index), ierr)
30
        END DO
31
     END IF
32
33
34
     Example 3.16 Same code, using MPI_WAITSOME.
35
     CALL MPI_COMM_SIZE(comm, size, ierr)
36
     CALL MPI_COMM_RANK(comm, rank, ierr)
37
     IF (rank .GT. 0) THEN
                                       ! client code
38
        DO WHILE(.TRUE.)
39
            CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
40
            CALL MPI_WAIT(request, status, ierr)
41
        END DO
42
     ELSE
                    ! rank=0 -- server code
43
        DO i=1, size-1
44
            CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
45
                             comm, request_list(i), ierr)
46
        END DO
47
        DO WHILE(.TRUE.)
```

CHAPTER 3. POINT-TO-POINT COMMUNICATION

78

```
CALL MPI_WAITSOME(size, request_list, numdone, &
indices, statuses, ierr)
DO i=1,numdone
CALL DO_SERVICE(a(1, indices(i)))
CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag, &
comm, request_list(indices(i)), ierr)
END DO
END DO
END DO
END IF
```

#### 3.7.6 Non-Destructive Test of status

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

MPI_REQU	JEST_GET_STATUS(request,	flag, status)	18		
IN	request	request (handle)	19 20		
OUT	flag	boolean flag, same as from $MPI\_TEST$ (logical)	21		
OUT	status	status object if flag is true (Status)	22 23		
C binding	<u>r</u>		24 25		
	·	quest request, int *flag,	25 26		
	MPI_Status *status)	1			
			27		
	008 binding		28		
MPI_Request_get_status(request, flag, status, ierror) 29					
TYPE(MPI_Request), INTENT(IN) :: request <sup>30</sup>					
LUGICAL, INTENT(UUT) :: ITag					
	MPI_Status) :: status		32		
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	33		
Fortran b	inding		34		
	ST_GET_STATUS(REQUEST, FI	AG STATUS TERROR)	35		
	ER REQUEST, STATUS(MPI_S		36		
	AL FLAG		37		
			38 39		
	-	complete, and, if so, returns in status the request			
		t does not deallocate or inactivate the request; a	40		
-	subsequent call to test, wait or free should be executed with that request. It sets flag = $\frac{41}{100}$				
false if the operation is not complete.					
One is allowed to call MPI REQUEST GET STATUS with a null or inactive request					

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.

48

44

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10 11

12

13

14

15

1	3.8 Pro	bbe and Cancel	
2 3	The MPI_PROBE, MPI_IPROBE, MPI_MPROBE, and MPI_IMPROBE operations allow in-		
4	coming messages to be checked for, without actually receiving them. The user can then		
5		,	e information returned by the probe (basically, the
6		-	icular, the user may allocate memory for the receive
7		ording to the length of the pr	-
8		-	s pending communications to be cancelled. This is
9	-	· ·	r a receive ties up user resources (send or receive o free these resources gracefully.
10 11	, ,	0	MPI_CANCEL is deprecated. Cancelling a sendrecv
12		calling MPI_CANCEL is not	
13			
14	3.8.1 Pro	obe	
15			
16			· · · · · · · · · · · · · · · · · · ·
17	MPI_IPRO	BE(source, tag, comm, flag, st	catus)
18 19	IN	source	rank of source or MPI_ANY_SOURCE (integer)
20	IN	tag	message tag or $MPI\_ANY\_TAG$ (integer)
21 22	IN	comm	communicator (handle)
23	OUT	flag	(logical)
24	OUT	status	status object (Status)
25			
26	C binding		
27	int MPI_1	-	g, MPI_Comm comm, int *flag,
28 29		MPI_Status *status)	
30	Fortran 2008 binding		
31	MPI_Iprobe(source, tag, comm, flag, status, ierror)		
32	INTEGER, INTENT(IN) :: source, tag		
33		(MPI_Comm), INTENT(IN) ::	comm
34	LOGICAL, INTENT(OUT) :: flag		
35	TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
36 27			,
37 38	Fortran binding		
39	MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)		
40	INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG		
41			
42			ag, status) returns flag = true if there is a message
43			the pattern specified by the arguments source, tag,
44			message that would have been received by a call to s) executed at the same point in the program, and
45 46			ald have been returned by MPI_RECV(). Otherwise,
40 47		turns $flag = false$ , and leaves	
48		<b>J</b>	

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.

MPI\_IPROBE is a local procedure since its return does not depend on MPI calls in other MPI processes, which is marked with the prefix I (for immediate).

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

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

It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.

A probe with MPI\_PROC\_NULL as source returns flag = true, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0; see Section 3.10.

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

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

C binding

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Fortran 2008 binding

MPI\_Probe(source, tag, comm, status, ierror)
 INTEGER, INTENT(IN) :: source, tag
 TYPE(MPI\_Comm), INTENT(IN) :: comm
 TYPE(MPI\_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
```

INTEGER SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR

MPI\_PROBE behaves like MPI\_IPROBE except that it is a non-local call that returns only after a matching message has been found.

The MPI implementation of MPI\_PROBE and MPI\_IPROBE needs to guarantee progress: 43 if a call to MPI\_PROBE has been issued by a process, and a send that matches the probe 44 has been initiated by some process, then the call to MPI\_PROBE will return, unless the 45 message is received by another concurrent receive operation (that is executed by another 46 thread at the probing process). Similarly, if a process busy waits with MPI\_IPROBE and a 47 matching message has been issued, then the call to MPI\_IPROBE will eventually return flag 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

29

30

31

32

33

34

35

36 37

38

39

40

41

```
1
     = true unless the message is received by another concurrent receive operation or matched
\mathbf{2}
     by a concurrent matched probe.
3
     Example 3.17 Use probe to wait for an incoming message.
4
5
          CALL MPI_COMM_RANK(comm, rank, ierr)
6
          IF (rank .EQ. 0) THEN
7
             CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
8
          ELSE IF (rank .EQ. 1) THEN
9
             CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
10
          ELSE IF (rank .EQ. 2) THEN
11
             DO i=1,2
12
                CALL MPI_PROBE(MPI_ANY_SOURCE, 0, &
13
                                 comm, status, ierr)
14
                IF (status(MPI_SOURCE) .EQ. 0) THEN
15
                    CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
     100
16
                ELSE
17
     200
                    CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
18
                END IF
19
             END DO
20
          END IF
21
22
     Each message is received with the right type.
23
^{24}
     Example 3.18 A similar program to the previous example, but now it has a problem.
25
26
          CALL MPI_COMM_RANK(comm, rank, ierr)
27
          IF (rank .EQ. 0) THEN
28
             CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
29
          ELSE IF (rank .EQ. 1) THEN
30
             CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
31
          ELSE IF (rank .EQ. 2) THEN
32
             DO i=1,2
33
                CALL MPI_PROBE(MPI_ANY_SOURCE, 0, &
34
                                 comm, status, ierr)
35
                IF (status(MPI_SOURCE) .EQ. 0) THEN
36
     100
                    CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, &
37
                                    0, comm, status, ierr)
38
                ELSE
39
     200
                    CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, &
40
                                    0, comm, status, ierr)
41
                END IF
42
             END DO
43
          END IF
44
45
         In Example 3.18, the two receive calls in statements labeled 100 and 200 in Example 3.17
46
     are slightly modified, using MPI_ANY_SOURCE as the source argument. The program is now
47
     incorrect: the receive operation may receive a message that is distinct from the message
```

48

#### Unofficial Draft for Comment Only

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 [33]. MPI\_MPROBE and MPI\_IMPROBE solve this problem by matching the incoming message so that it may only be received with MPI\_MRECV or MPI\_IMRECV on the corresponding message handle. (*End of advice to users.*)

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

#### 3.8.2 Matching Probe

The function MPI\_PROBE checks for incoming messages without receiving them. Since the list of incoming messages is global among the threads of each MPI process, it can be hard to use this functionality in threaded environments [33, 30].

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

MPI\_IMPROBE(source, tag, comm, flag, message, status) 36 37 IN rank of source or MPI\_ANY\_SOURCE (integer) source 38 IN message tag or MPI\_ANY\_TAG (integer) tag 39 IN comm communicator (handle) 40 41 OUT flag flag (logical) 42OUT returned message (handle) message 43 OUT status status object (Status) 444546C binding 47int MPI\_Improbe(int source, int tag, MPI\_Comm comm, int \*flag,

MPI\_Message \*message, MPI\_Status \*status)

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23 24

25

26

27

28

29

30

 $^{31}$ 

32

33

34 35

1 Fortran 2008 binding  $\mathbf{2}$ MPI\_Improbe(source, tag, comm, flag, message, status, ierror) 3 INTEGER, INTENT(IN) :: source, tag 4 TYPE(MPI\_Comm), INTENT(IN) :: comm 5LOGICAL, INTENT(OUT) :: flag 6 TYPE(MPI\_Message), INTENT(OUT) :: message 7 TYPE(MPI\_Status) :: status 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 Fortran binding 10 MPI\_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR) 11 INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 12LOGICAL FLAG 13 14MPI\_IMPROBE(source, tag, comm, flag, message, status) returns flag = true if there is 15a message that can be received and that matches the pattern specified by the arguments 16source, tag, and comm. The call matches the same message that would have been received 17by a call to MPI\_RECV(..., source, tag, comm, status) executed at the same point in the 18 program and returns in status the same value that would have been returned by MPI\_RECV. 19In addition, it returns in message a handle to the matched message. Otherwise, the call 20returns flag = false, and leaves status and message undefined. 21MPI\_IMPROBE is a local procedure. According to the definitions in Section 2.4.2 and 22in contrast to MPI\_IPROBE, it is a nonblocking procedure because it is the initialization of 23a matched receive operation.  $^{24}$ A matched receive (MPI\_MRECV or MPI\_IMRECV) executed with the message han-25dle will receive the message that was matched by the probe. Unlike MPI\_IPROBE, no 26other probe or receive operation may match the message returned by MPI\_IMPROBE. 27Each message returned by MPI\_IMPROBE must be received with either MPI\_MRECV or 28MPI\_IMRECV. 29The source argument of MPI\_IMPROBE can be MPI\_ANY\_SOURCE, and the tag argu-30 ment can be MPI\_ANY\_TAG, so that one can probe for messages from an arbitrary source  $^{31}$ and/or with an arbitrary tag. However, a specific communication context must be provided 32 with the comm argument. 33 A synchronous send operation that is matched with MPI\_IMPROBE or MPI\_MPROBE 34 will complete successfully only if both a matching receive is posted with MPI\_MRECV or 35 MPI\_IMRECV, and the receive operation has started to receive the message sent by the 36 synchronous send. 37 There is a special predefined message: MPI\_MESSAGE\_NO\_PROC, which is a message 38which has MPI\_PROC\_NULL as its source process. The predefined constant 39 MPI\_MESSAGE\_NULL is the value used for invalid message handles. 40A matching probe with source = MPI\_PROC\_NULL returns flag = true, message = 41 MPI\_MESSAGE\_NO\_PROC, and the status object returns source = MPI\_PROC\_NULL, tag = 42MPI\_ANY\_TAG, and count = 0; see Section 3.10. It is not necessary to call MPI\_MRECV or 43MPI\_IMRECV with MPI\_MESSAGE\_NO\_PROC, but it is not erroneous to do so. 44 45Rationale. MPI\_MESSAGE\_NO\_PROC was chosen instead of #-update8 46 MPI\_MESSAGE\_PROC\_NULL to avoid possible confusion as another null handle con-47stant. (End of rationale.) MPI\_MESSAGE\_PROC\_NULL must use \constskip and not 48 constmain becaue it is an invalid name, i.e., not an MPI constant **Unofficial Draft for Comment Only** 

#-405

#-PR449

Done

in PR449

MPI_MPR	OBE(source, tag, comm, messa	ge, status)
IN	source	${\rm rank} ~{\rm of} ~{\rm source} ~{\rm or} ~{\sf MPI\_ANY\_SOURCE} ~({\rm integer})$
IN	tag	message tag or $MPI\_ANY\_TAG\xspace$ (integer)
IN	comm	communicator (handle)
OUT	message	returned message (handle)
OUT	status	status object (Status)
Fortran 2 MPI_Mprob INTEG TYPE( TYPE( TYPE(		tag comm :: message

#### Fortran binding

MPI\_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR) INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR  $^{24}$ 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 2.4.2, MPI\_MPROBE is incomplete. It is also a non-local procedure.  $^{31}$ Advice to users. This is one of the exceptions in which incomplete procedures are non-local. (End of advice to users.) 3.8.3 Matched Receives 

The functions MPI\_MRECV and MPI\_IMRECV receive messages that have been previously matched by a matching probe (Section 3.8.2).

1 MPI\_MRECV(buf, count, datatype, message, status) 2 OUT buf initial address of receive buffer (choice) 3 IN count number of elements in receive buffer (non-negative 4 integer) 56 IN datatype of each receive buffer element (handle) datatype 7 INOUT message message (handle) 8 OUT status status object (Status) 9 10 11 C binding int MPI\_Mrecv(void \*buf, int count, MPI\_Datatype datatype, 12MPI\_Message \*message, MPI\_Status \*status) 13 14int MPI\_Mrecv\_c(void \*buf, MPI\_Count count, MPI\_Datatype datatype, 15MPI\_Message \*message, MPI\_Status \*status) 1617Fortran 2008 binding MPI\_Mrecv(buf, count, datatype, message, status, ierror) 1819TYPE(\*), DIMENSION(..) :: buf INTEGER, INTENT(IN) :: count 20TYPE(MPI\_Datatype), INTENT(IN) :: datatype 21TYPE(MPI\_Message), INTENT(INOUT) :: message 22 23TYPE(MPI\_Status) :: status  $^{24}$ INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25MPI\_Mrecv(buf, count, datatype, message, status, ierror) 26TYPE(\*), DIMENSION(..) :: buf 27INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 28TYPE(MPI\_Datatype), INTENT(IN) :: datatype 29 TYPE(MPI\_Message), INTENT(INOUT) :: message 30 TYPE(MPI\_Status) :: status 31INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32 33 Fortran binding 34MPI\_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR) 35 <type> BUF(\*) 36 INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 37 This call receives a message matched by a matching probe operation (Section 3.8.2). 38 The receive buffer consists of the storage containing **count** consecutive elements of the 39 type specified by datatype, starting at address buf. The length of the received message must 40 be less than or equal to the length of the receive buffer. An overflow error occurs if all 41 incoming data does not fit, without truncation, into the receive buffer. 42If the message is shorter than the receive buffer, then only those locations corresponding 43 to the (shorter) message are modified. 44 On return from this function, the message handle is set to MPI\_MESSAGE\_NULL. All 45

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. This is consistent with the status object produced by a call to MPI\_RECV or to MPI\_PROBE with source = MPI\_PROC\_NULL (see Section 3.10). A call to MPI\_MRECV with MPI\_MESSAGE\_NULL is erroneous.

MPI\_IMRECV(buf, count, datatype, message, request)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
INOUT	message	message (handle)
OUT	request	communication request (handle)

#### (

C binding	17
int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,	18 19
MPI_Message *message, MPI_Request *request)	
	20
<pre>int MPI_Imrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	21 22
MPI_Message *message, MPI_Request *request)	22
Fortran 2008 binding	23 24
MPI_Imrecv(buf, count, datatype, message, request, ierror)	24 25
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	26
INTEGER, INTENT(IN) :: count	20
TYPE(MPI_Datatype), INTENT(IN) :: datatype	28
TYPE(MPI_Message), INTENT(INOUT) :: message	29
TYPE(MPI_Request), INTENT(OUT) :: request	30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31
NDT Transmither and the transmither to make the transmither	32
<pre>MPI_Imrecv(buf, count, datatype, message, request, ierror)     TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf</pre>	33
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	34
TYPE(MPI_Datatype), INTENT(IN) :: datatype	35
TYPE(MPI_Message), INTENT(INOUT) :: message	36
TYPE(MPI_Request), INTENT(OUT) :: request	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
INTEGER, OFFICIARE, INTENI(UOI) TETTOT	39
Fortran binding	40
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)	41
<type> BUF(*)</type>	42
INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR	43
MPI_IMRECV is the nonblocking variant of MPI_MRECV and starts a nonblocking	44
	45

45receive of a matched message. Completion semantics are similar to MPI\_IRECV as described 46in Section 3.7.2. On return from this function, the message handle is set to 47MPI\_MESSAGE\_NULL. 48

1  $\mathbf{2}$ 

3

4

56  $\overline{7}$ 

1	If MPI_IMRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the
2	call returns immediately with a request object which, when completed, will yield a status
3	object set to source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0, as if a receive
4	from MPI_PROC_NULL was issued (see Section 3.10). A call to MPI_IMRECV with
5	MPI_MESSAGE_NULL is erroneous.
6	
7	Advice to implementors. If reception of a matched message is started with
8	MPI_IMRECV, then it is possible to cancel the returned request with MPI_CANCEL. If
9	MPI_CANCEL succeeds, the matched message must be found by a subsequent message
10	probe (MPI_PROBE, MPI_IPROBE, MPI_MPROBE, or MPI_IMPROBE), received by
11	a subsequent receive operation or cancelled by the sender. See Section 3.8.4 for details
12	about MPI_CANCEL. The cancellation of operations initiated with MPI_IMRECV may
13	fail. (End of advice to implementors.)
14	
15	3.8.4 Cancel
16	
17	
18	MPI_CANCEL(request)
19	
20	IN request communication request (handle)
21	
22	C binding
23	<pre>int MPI_Cancel(MPI_Request *request)</pre>
24	Fortran 2008 binding
25	MPI_Cancel(request, ierror)
26	TYPE(MPI_Request), INTENT(IN) :: request
27	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28	
29	Fortran binding
30	MPI_CANCEL(REQUEST, IERROR)
31	INTEGER REQUEST, IERROR
32	A call to MPI_CANCEL marks for cancellation a pending, nonblocking communica-
33	tion operation (send or receive). Cancelling a send request by calling MPI_CANCEL is
34	

is ng a sena equest by ıg. 34deprecated. The cancel call is local. It returns immediately, possibly before the communi-35 cation is actually cancelled. It is still necessary to call MPI\_REQUEST\_FREE, MPI\_WAIT or 36 MPI\_TEST (or any of the derived operations) with the cancelled request as argument after 37 the call to MPI\_CANCEL. If a communication is marked for cancellation, then a MPI\_WAIT 38 call for that communication is guaranteed to return, irrespective of the activities of other 39 processes (i.e., MPI\_WAIT behaves as a local function); similarly if MPI\_TEST is repeatedly 40 called in a busy wait loop for a cancelled communication, then MPI\_TEST will eventually 41 be successful. 42

<sup>43</sup> MPI\_CANCEL can be used to cancel a communication that uses a persistent request (see <sup>44</sup> Section 3.9), in the same way it is used for nonpersistent requests. Cancelling a persistent <sup>45</sup> send request by calling MPI\_CANCEL is deprecated. A successful cancellation cancels the <sup>46</sup> active communication, but not the request itself. After the call to MPI\_CANCEL and the <sup>47</sup> subsequent call to MPI\_WAIT or MPI\_TEST, the request becomes inactive and can be <sup>48</sup> 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.*)

			19	
			20	
MPI_TEST	CANCELLED(status, fl	ag)	21	
IN	status	status object (Status)	22	
OUT	flag	(logical)	23	
	<b>U</b>		24	
C binding	m		25	
	•	MDI Status tetatus int tflag)	26	
INC MPI_I		MPI_Status *status, int *flag)	27	
Fortran 2	2008 binding		28	
MPI_Test_cancelled(status, flag, ierror) 26			29	
TYPE(MPI_Status), INTENT(IN) :: status 3			30	
LOGICAL, INTENT(OUT) :: flag				
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 32				
<b>D</b> ( )			33	
			34	
			35	
INTEGER STATUS(MPI_STATUS_SIZE), IERROR 36				
LOGIC	CAL FLAG		37	
Retur	ns flag = true if the comm	nunication associated with the status object was cancelled	38	
	-	her fields of status (such as count or tag) are undefined.	39	

Returns flag = true if the communication associated with the status object was cancelledsuccessfully. 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  $_{47}$  transferred to the receiver before a matching receive is posted), then the cancellation  $_{48}$ 

#### 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

40 41

42 43

44

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

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

#### 3.9 Persistent Communication Requests

11 Often a communication with the same argument list (with the exception of the buffer con-12tents) is repeatedly executed within the inner loop of a parallel computation. In such a 13 situation, it may be possible to optimize the communication by binding the list of com-14munication arguments to a **persistent** communication request once and, then, repeatedly 15using the request to initiate and complete operations. In the case of point-to-point commu-16nication, the persistent request thus created can be thought of as a communication port or 17a "half-channel." It does not provide the full functionality of a conventional channel, since 18 there is no binding of the send port to the receive port. This construct allows reduction 19 of the overhead for communication between the process and communication controller, but 20not of the overhead for communication between one communication controller and another. 21It is not necessary that messages sent with a persistent point-to-point request be received 22 by a receive operation using a persistent point-to-point request, or vice versa. 23

There are also collective communication persistent operations defined in Section 6.13 $^{24}$ and Section 8.8. The remainder of this section covers the point-to-point persistent initializa-25tion operations and the start routines, which are used for both point-to-point and collective 26persistent communication. 27

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.

29 30  $^{31}$ 

43

28

#### MPI\_SEND\_INIT(buf, count, datatype, dest, tag, comm, request)

			<b>c</b> , ,
32 33	IN	buf	initial address of send buffer (choice)
34	IN	count	number of elements sent (non-negative integer)
35	IN	datatype	type of each element (handle)
36 37	IN	dest	rank of destination (integer)
38	IN	tag	message tag (integer)
39	IN	comm	communicator (handle)
40	OUT	request	communication request (handle)
41			(initial)
42			

#### C binding

44	<pre>int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype,</pre>
45	<pre>int dest, int tag, MPI_Comm comm, MPI_Request *request)</pre>
46	<pre>int MPI_Send_init_c(const void *buf, MPI_Count count,</pre>
47	MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
48	MPI_Request *request)

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

1

2

3

4

5

6

7

8 9

```
1
Fortran 2008 binding
                                                                                       \mathbf{2}
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       5
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                       10
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                       11
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                       12
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       13
    INTEGER, INTENT(IN) :: dest, tag
                                                                                       14
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                       15
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                       16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       17
                                                                                       18
Fortran binding
                                                                                       19
MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
                                                                                       20
    <type> BUF(*)
                                                                                       21
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
                                                                                       22
    Creates a persistent communication request for a standard mode send operation, and
                                                                                       23
binds to it all the arguments of a send operation.
                                                                                       24
                                                                                       25
                                                                                       26
MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)
                                                                                       27
  IN
                                      initial address of send buffer (choice)
           buf
                                                                                       28
  IN
                                      number of elements sent (non-negative integer)
                                                                                       29
           count
                                                                                       30
  IN
           datatype
                                      type of each element (handle)
                                                                                       31
  IN
           dest
                                      rank of destination (integer)
                                                                                       32
                                                                                       33
  IN
           tag
                                      message tag (integer)
                                                                                       34
  IN-
                                      communicator (handle)
           comm
                                                                                       35
  OUT
                                      communication request (handle)
           request
                                                                                       36
                                                                                       37
C binding
                                                                                       38
int MPI_Bsend_init(const void *buf, int count, MPI_Datatype datatype,
                                                                                       39
              int dest, int tag, MPI_Comm comm, MPI_Request *request)
                                                                                       40
                                                                                       41
int MPI_Bsend_init_c(const void *buf, MPI_Count count,
                                                                                       42
              MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
                                                                                       43
              MPI_Request *request)
                                                                                       44
Fortran 2008 binding
                                                                                       45
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                       46
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                       47
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                       48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         TYPE(MPI_Request), INTENT(OUT) :: request
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
6
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
7
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         INTEGER, INTENT(IN) :: dest, tag
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
16
         <type> BUF(*)
17
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
18
         Creates a persistent communication request for a buffered mode send.
19
20
21
     MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)
22
       IN
                buf
                                           initial address of send buffer (choice)
23
^{24}
       IN
                                           number of elements sent (non-negative integer)
                count
25
       IN
                datatype
                                           type of each element (handle)
26
       IN
                dest
                                           rank of destination (integer)
27
       IN
28
                tag
                                           message tag (integer)
29
       IN
                                           communicator (handle)
                comm
30
       OUT
                request
                                           communication request (handle)
^{31}
32
     C binding
33
     int MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype,
34
                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
35
36
     int MPI_Ssend_init_c(const void *buf, MPI_Count count,
37
                    MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
38
                    MPI_Request *request)
39
     Fortran 2008 binding
40
     MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
41
42
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
         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
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

MPI\_Ssend\_init(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: dest, tag TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI_	_SSEND_	INIT(BUF,	, COUNT, I	DATATYPE	, DEST	, TAG,	COMM,	REQUEST,	IERROR)
	<type></type>	BUF(*)							*
	INTEGE	R COUNT,	DATATYPE,	, DEST,	TAG, C	OMM, R	EQUEST	, IERROR	

Creates a persistent communication object for a synchronous mode send operation.

MPI\_RSEND\_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

C binding

```
int MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Rsend_init_c(const void *buf, MPI_Count count,
             MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
             MPI_Request *request)
Fortran 2008 binding
MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
```

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

INTEGER, INTENT(IN) :: dest, tag

1

2

 $\mathbf{4}$ 

5 6

7

9

1516

29

30

31

32

33

34 35

36

37

38

39

40

41

42

43

44

45

46

47

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
\mathbf{2}
         TYPE(MPI_Request), INTENT(OUT) :: request
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     Fortran binding
5
     MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
6
         <type> BUF(*)
7
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
8
9
         Creates a persistent communication object for a ready mode send operation.
10
11
     MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)
12
13
                                           initial address of receive buffer (choice)
       OUT
                buf
14
       IN
                count
                                           number of elements received (non-negative integer)
15
                                           type of each element (handle)
       IN
                datatype
16
17
       IN
                source
                                           rank of source or MPI_ANY_SOURCE (integer)
18
                                            message tag or MPI_ANY_TAG (integer)
       IN
                tag
19
       IN
                comm
                                            communicator (handle)
20
21
       OUT
                request
                                           communication request (handle)
22
23
     C binding
^{24}
     int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source,
25
                    int tag, MPI_Comm comm, MPI_Request *request)
26
     int MPI_Recv_init_c(void *buf, MPI_Count count, MPI_Datatype datatype,
27
                    int source, int tag, MPI_Comm comm, MPI_Request *request)
28
29
     Fortran 2008 binding
30
     MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
^{31}
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
32
         INTEGER, INTENT(IN) :: count, source, tag
33
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(MPI_Request), INTENT(OUT) :: request
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
38
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
39
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
40
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         INTEGER, INTENT(IN) :: source, tag
         TYPE(MPI_Comm), INTENT(IN) :: comm
43
         TYPE(MPI_Request), INTENT(OUT) :: request
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
48
```

• 1	> BUF(*) ER COUNT, DATATYPE, SOURG	CE, TAG, COMM, REQUEST, IERROR	1 $2$		
Create	Creates a persistent communication request for a receive operation. The argument buf				
	-	permission to write on the receive buffer by passing	4		
	ent to MPI_RECV_INIT.		5		
0		is inactive after it was created—no active commu-	6 7		
_	attached to the request.		8		
A com	munication that uses a persi	stent request is initiated by the function	9		
MPI_STAR	Т.		10		
			11		
MPI_STAR	T(request)		12		
	( · · )		13		
INOUT	request	communication request (handle)	14		
			15		
C binding	-		16		
int MPI_S	tart(MPI_Request *request	;)	17		
Fortran 2	008 binding		18		
	(request, ierror)		19		
TYPE(	MPI_Request), INTENT(INOU	JT) :: request	20		
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	21		
Fortron b					
Fortran binding23MPI_START(REQUEST, IERROR)24					
			25		
	26				
	The argument, request, is a handle returned by one of the previous five calls. The				
associated request should be inactive. The request becomes active once the call is made.					
If the request is for a send with ready mode, then a matching receive should be posted					
before the call is made. The communication buffer should not be modified after the call, and until the operation completes.					
	and until the operation completes. The call is local, with similar semantics to the nonblocking communication operations				
		to MPI_START with a request created by	32		
		n in the same manner as a call to MPI_ISEND; a	33		
		ted by MPI_BSEND_INIT starts a communication	34		
	e manner as a call to MPI_IBS		35		
		,	36		
		、 、	37 38		
MPI_STAR	TALL(count, array_of_requests	5)	39		
IN	count	list length (non-negative integer)	40		
INOUT	array_of_requests	array of requests (array of handles)	41		
			42		
C binding	5		43		
-	-	equest array_of_requests[])	44		
	45				
	008 binding	ata iomon)	46		
	all(count, array_of_reque	esus, terror)	47		
INIEG.	INTEGER, INTENT(IN) :: count 48				

Unofficial Draft for Comment Only

1 TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count)  $\mathbf{2}$ INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3 Fortran binding 4 MPI\_STARTALL(COUNT, ARRAY\_OF\_REQUESTS, IERROR) 5INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), IERROR 6 7 Start all communications associated with requests in array\_of\_requests. A call to 8 MPI\_STARTALL(count, array\_of\_requests) has the same effect as calls to 9 MPI\_START (&array\_of\_requests[i]), executed for  $i=0, \ldots, count-1$ , in some arbitrary order. 10 A communication started with a call to MPI\_START or MPI\_STARTALL is completed 11by a call to MPI\_WAIT, MPI\_TEST, or one of the derived functions described in Sec-12tion 3.7.5. The request becomes inactive after successful completion of such call. The re-

quest is not deallocated and it can be activated anew by an MPI\_START or MPI\_STARTALL
 call.

15A persistent request is deallocated by a call to MPI\_REQUEST\_FREE (Section 3.7.3). 16The call to MPI\_REQUEST\_FREE can occur at any point in the program after the per-17sistent request was created. However, the request will be deallocated only after it becomes 18 inactive. Active receive requests should not be freed. Otherwise, it will not be possible to 19 check that the receive has completed. Collective operation requests (defined in Section 6.12) 20and Section 8.7 for nonblocking collective operations, and Section 6.13 and Section 8.8 for 21persistent collective operations) must not be freed while active. It is preferable, in general, 22to free requests when they are inactive. If this rule is followed, then the functions described 23in this section will be invoked in a sequence of the form, 24

25 26

27

28

29

30

31

32 33

34

35

36 37

38 39

40

41

42

#### Create (Start Complete)\* Free

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

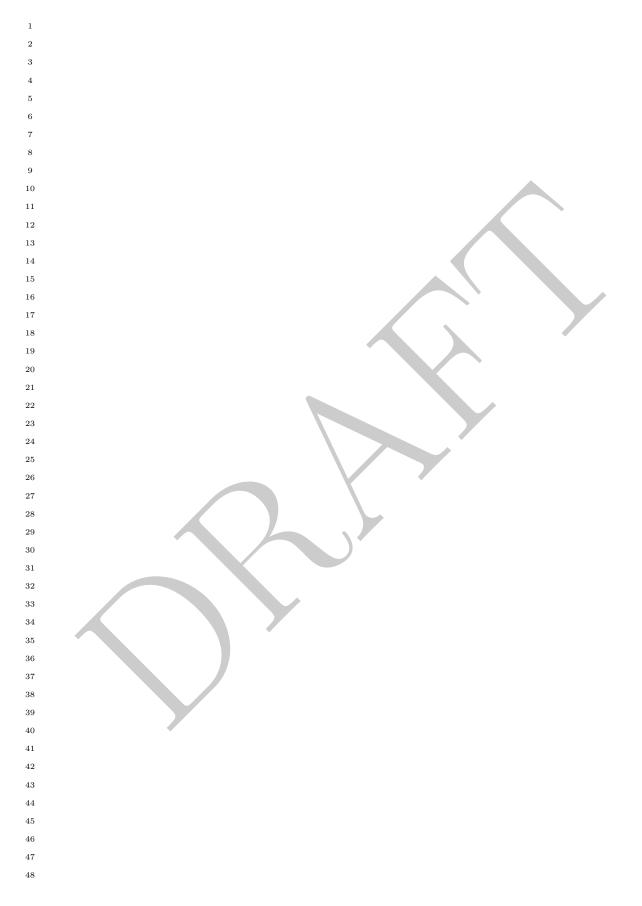
A send operation initiated with MPI\_START can be matched with any receive operation and, likewise, a receive operation initiated with MPI\_START can receive messages generated by any send operation.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10–19.1.20. (End of advice to users.)

#### 3.10 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, tag = MPI\_ANY\_TAG and count = 0. A matching probe (cf. Section 3.8.2) with source = MPI\_PROC\_NULL returns flag = true, message = MPI\_MESSAGE\_NO\_PROC, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0.



## Chapter 4

# Partitioned Point-to-Point Communication

#### 4.1 Introduction

Partitioned communication extends persistent point-to-point communication as defined in Chapter 3. Partitioned communication operations are matched based on the order in which the local initialization calls are performed. Partitioned communication is "partitioned" because it allows for multiple contributions of data to be made, potentially, from multiple actors (e.g., threads or tasks) in an MPI process to a single communication operation.

Advice to users. The techniques of partitioned communication were known as "finepoints" before their adoption into the MPI standard. We refer the interested reader to the original literature describing the design goals, functioning, initial implementation and performance improvements [28, 29]. (End of advice to users.)

Partitioned communication operations use a persistent communication style that involves a sequence of start and test or wait operations. For this sequence partitioned communications use MPI\_START or MPI\_STARTALL calls and completion mechanisms (MPI\_TEST or MPI\_WAIT). Partitioned communication is different in three fundamental ways from persistent point-to-point operations in MPI. First, partitioned communication allows additional partitioned test function calls that can expose partial completion of the operation. Second, partitioned communication may perform all of the initialization required to enable data transfer as early as its initialization phase. Third, partitioned communication allows for MPI to be independently notified of multiple contributions from the send-side to a single data buffer of a single MPI message.

39 The rationale behind having different initialization behavior allowed Rationale. for partitioned communication as opposed to persistent point-to-point is to enable 40 41 flexibility and optimization possibilities in implementations. Buffer setup can occur in 42the partitioned communication initialization functions (see Section 4.2.1). However, such negotiation can be deferred until data is to be moved between two processes. 4344This means that partitioned communication can lazily negotiate as late as testing for completion of the operation on the first iteration of a partitioned communication 4546start and test or wait operations. Matching still occurs as if matching happened 47at the partitioned communication initialization functions as noted in the function 48 descriptions. (End of rationale.)

#### Unofficial Draft for Comment Only

9

10 11

17 18

19

20

21

22

23

 $^{24}$ 

25 26

27 28

29

30

31

32

33

34

35

36

37

CHAPTER 4. PARTITIONED POINT-TO-POINT COMMUNICATION

1 2

### 4.2 Semantics of Partitioned Point-to-Point Communication

MPI guarantees certain general properties of partitioned point-to-point communication
 <sup>4</sup> progress, which are described in this section.

Persistent communications use opaque MPI\_REQUEST objects as described in Sec tion 3. Partitioned communication uses these same semantics for MPI\_REQUEST objects.
 Partitioned communication provides fine-grained transfers on either or both sides of a

Partitioned communication provides fine-grained transfers on either or both sides of a
send-receive operation described by requests. Persistent communication semantics are ideal
for partitioned communication: they provide MPI\_PSEND\_INIT and MPI\_PRECV\_INIT
functions that allow partitioned communication setup to occur prior to message transfers.
Partitioned communication initialization functions are local. The partitioned communication initialization includes inputs on the number of user-visible partitions on the send-side
and receive-sides, which may differ. Valid partitioned communication operations must have
one or more partitions specified.

Once an MPI\_PSEND\_INIT call has been made, the user may start the operation with a call to a starting procedure and complete the operation with a number of MPI\_PREADY calls equal to the requested number of send partitions followed by a call to a completing procedure. A call to MPI\_PREADY notifies the MPI library that a specified portion of the data buffer (a specific partition) is ready to be sent. Notification of partial completion can be done via fine-grained MPI\_PARRIVED calls at the receiver before a final MPI\_TEST/

MPI\_WAIT on the request itself; the latter represents overall operation completion upon success. A full set of methods for starting and completing partitioned communication is given in the following sections.

 $\frac{24}{25}$ 

26

27

 $^{31}$ 

32

33 34

35

36

37

38 39

40

41 42 Advice to users. Having a large number of receiver-side partitions can increase overheads as the completion mechanism may need to work with finer-grained notifications. Using a small number of receiver-side partitions may provide higher performance.

A large number of sender-side partitions may be aggregated by an MPI implementa tion, making performance concerns of a large number of sender-side partitions poten tially less impactful than receiver-side granularity. (*End of advice to users.*)

Advice to implementors. It is expected that an MPI implementation will attempt to balance latency and aggregation for data transfers for the requested partition counts on the sender-side and receiver-side to allow optimization for different hardware. A high quality implementation may perform significant optimizations to enhance performance in this way; they may, for example, resize the data transfers of the partitions to combine partitions in fractional partition sizes (e.g., 2.5 partitions in a single data transfer). (End of advice to implementors.)

Example 4.1 shows a simple partitioned transfer in which the sender-side and receiverside partitioning is identical in partition count.

#### Example 4.1

- 43 44
- 45
- 46
- 47

ſ

```
1
#include "mpi.h"
                                                                                       \mathbf{2}
#define PARTITIONS 8
                                                                                       3
#define COUNT 5
int main(int argc, char *argv[])
                                                                                       4
                                                                                       5
  double message[PARTITIONS*COUNT];
                                                                                       6
  MPI_Count partitions = PARTITIONS;
  int source = 0, dest = 1, tag = 1, flag = 0;
                                                                                       9
  int myrank, i;
                                                                                       10
  int provided;
                                                                                       11
  MPI_Request request;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
                                                                                       12
  if (provided < MPI_THREAD_SERIALIZED)
                                                                                       13
                                                                                       14
     MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
                                                                                       15
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                                                                                       16
  if (myrank == 0)
                                                                                       17
  {
     MPI_Psend_init(message, partitions, COUNT, MPI_DOUBLE, dest, tag,
                                                                                       18
                                                                                       19
                MPI_COMM_WORLD, MPI_INFO_NULL, &request);
     MPI_Start(&request);
                                                                                       20
                                                                                      21
     for(i = 0; i < partitions; ++i)</pre>
                                                                                       22
     {
                                                                                       23
        /* compute and fill partition #i, then mark ready: */
                                                                                       ^{24}
        MPI_Pready(i, &request);
                                                                                       25
     }
                                                                                       26
     while(!flag)
                                                                                       27
     {
        /* do useful work #1 */
                                                                                       28
                                                                                       29
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                       30
        /* do useful work #2 */
                                                                                       31
     }
     MPI_Request_free(&request);
                                                                                       32
                                                                                       33
  }
                                                                                      34
  else if (myrank == 1)
  ſ
                                                                                      35
     MPI_Precv_init(message, partitions, COUNT, MPI_DOUBLE, source, tag,
                                                                                      36
                                                                                      37
                MPI_COMM_WORLD, MPI_INFO_NULL, &request);
                                                                                       38
     MPI_Start(&request);
                                                                                       39
     while(!flag)
                                                                                       40
     {
                                                                                       41
        /* do useful work #1 */
                                                                                       42
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* do useful work #2 */
                                                                                       43
                                                                                       44
     }
                                                                                       45
     MPI_Request_free(&request);
  }
                                                                                       46
                                                                                       47
  MPI_Finalize();
                                                                                       48
  return 0;
```

Rationale. Partitioned communication is designed to provide opportunities for MPI implementations to optimize data transfers. MPI is free to choose how many transfers to do within a partitioned communication send independent of how many partitions are reported as ready to MPI through MPI\_PREADY calls. Aggregation of partitions is permitted but not required. Ordering of partitions is permitted but not required. A naive implementation can simply wait for the entire message buffer to be marked ready before any transfer(s) occur and could wait until the completion function is called on a request before transferring data. However, this modality of communication gives MPI implementations far more flexibility in data movement than non-partitioned communications. (End of rationale.)

#### 4.2.1 Communication Initialization and Starting with Partitioning

Initialization of partitioned communication operations use the initialization calls described below. Subsequent to initialization, MPI\_START/MPI\_STARTALL are used as the first indication to MPI that a message transfer will occur. For send-side operations, neither initializing nor starting the operation enables transfer of any part of the user buffer. Freeing or canceling a partitioned communication request that is active (i.e., initialized and started) and not completed is erroneous. After the partitioned communication operation is started, individual partitions of a message are indicated as ready to be sent by MPI via the MPI\_PREADY function, described below.

MPI\_PSEND\_INIT(buf, partitions, count, datatype, dest, tag, comm, info, request)

		( =,,,,,, _			
27 28	IN	buf	initial address of send buffer (choice) (choice)		
29	IN	partitions	number of partitions (non-negative integer)		
30 31	IN	count	number of elements send per partition (non-negative integer)		
32 33	IN	datatype	type of each element (handle)		
33 34	IN	dest	rank of destination (integer)		
35	IN	tag	message tag (integer)		
36	IN	comm	communicator (handle)		
37 38	IN	info	info argument (handle)		
39	OUT	request	communication request (handle)		
40					
41	<sup>11</sup> C binding				
42	int MPI_Psend_init(void *buf, int partitions, MPI_Count count,				
43	MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,				
44	MPI_Info info, MPI_Request *request)				
45					
46	Fontan 2008 hinding				
47	MPI_Psend	l_init(buf, partitions, co	ount, datatype, dest, tag, comm, info,		
48		request, ierror)			

}

1 2

5

6

10

11

12

13

14

15 16

17

18

19

20

21

22

23 24

25

26

27

28

29 30 31

32

48

TYPE(\*), DIMENSION(..), INTENT(IN) :: buf INTEGER, INTENT(IN) :: partitions, dest, tag INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: datatype TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_PSEND\_INIT creates a partitioned communication request and binds to it all the arguments of a partitioned send operation. Matching follows the same MPI matching rules as for point-to-point communication (see Chapter 3) with communicator, tag and source dictating message matching. In the event that the communicator, tag and source do not uniquely identify a message, the order in which partitioned communication *initialization* calls are made is the order in which they will eventually match. This operation can only match with partitioned communication initialization operations, therefore it is required to be matched with a corresponding MPI\_PRECV\_INIT call. Partitioned communication initialization calls are local. It is erroneous to provide a partitions value  $\leq 0$ . Send-side and receive-side buffers must be identical in size.

Advice to implementors. Unlike MPI\_SEND\_INIT, MPI\_PSEND\_INIT can be matched as early as the initialization call. Also, unlike MPI\_SEND\_INIT, MPI\_PSEND\_INIT takes an info argument. (*End of advice to implementors.*)

MPI_F	MPI_PRECV_INIT(buf, partitions, count, datatype, dest, tag, comm, info, request) 33				
IN	buf	initial address of recv buffer (choice) (choice)	34		
IN	partitions	number of partitions (non-negative integer)	35		
IN	count	number of elements send per partition (non-negative	36 37		
		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		
IN	info	info argument (handle)	44		
OUT	request	communication request (handle)	45		
001	τεγμεσι	communication request (nandle)	46		
			47		

## MPI\_PRECV\_INIT(buf, partitions, count, datatype, dest, tag, comm, info, request

#### C binding

1	int MPI_Pr	<pre>recv_init(void *buf, int</pre>	partitions, MPI_Count count,		
2		MPI_Datatype datatype	e, int dest, int tag, MPI_Comm comm,		
3		MPI_Info info, MPI_R	equest *request)		
4	Fortran 20	008 binding			
5		0	ount, datatype, dest, tag, comm, info,		
6	III 1_1 100V_	request, ierror)			
7	TYPE(*	<pre>(), DIMENSION(), INTEN]</pre>	C(IN) :: buf		
8		ER, INTENT(IN) :: partiti			
9		ER(KIND=MPI_COUNT_KIND),			
10		<pre>//PI_Datatype), INTENT(IN)</pre>			
11		<pre>IPI_Comm), INTENT(IN) ::</pre>			
12		<pre>IPI_Info), INTENT(IN) ::</pre>			
13		(PI_Request), INTENT(OUT)			
14		ER, OPTIONAL, INTENT(OUT)			
15					
16	Fortran bi	5			
17	MPI_PRECV_		DUNT, DATATYPE, DEST, TAG, COMM, INFO,		
18		REQUEST, IERROR)			
19	• -	> BUF(*)			
20			DEST, TAG, COMM, INFO, REQUEST, IERROR		
21 22	INTEGE	ER(KIND=MPI_COUNT_KIND) (	COUNT		
22					
23	Ratio	_	provided in order to support per-operation imple-		
25	menta	tion-defined info keys. (End	of rationale.)		
26	MPI_P	RECV_INIT creates a partitio	ned communication receive request and binds to it		
27	all the argu	ments of a partitioned receiv	ve operation. This operation can only match with		
28	partitioned	communication initialization	operations, therefore the MPI library is required to		
29	match $MPI_{-}$	PRECV_INIT calls only with	a corresponding MPI_PSEND_INIT call. Matching		
30	follows the same MPI matching rules as for point-to-point communication (see Chapter 3)				
31	with communicator, tag and source dictating message matching. In the event that the				
32	communicator, tag and source do not uniquely identify a message, the order in which				
33			n calls are made is the order in which they will		
34			cation initialization calls are local. That is,		
35		e e	e operation completes. It is erroneous to provide a		
36	partitions va	$100 \leq 0$ . Wildcards for source	e and tag are not allowed.		
37	Advice	e to implementors Unlike M	IPI_RECV_INIT, MPI_PRECV_INIT may communi-		
38		-	, MPI_PRECV_INIT takes an info argument. ( <i>End</i>		
39		vice to implementors.)			
40	- ,	·····			
41					
42	MPI PREAI	DY(partition, request)			
43					
44 45	IN	partition	partition to mark ready for transfer (non-negative integer)		
46	INOUT	request	partitioned communication request (handle)		
40			Particular communication request (number)		
48	C binding				

int MPI\_Pready(int partition, MPI\_Request \*request)

## Fortran 2008 binding

MPI\_Pready(partition, request, ierror)
 INTEGER, INTENT(IN) :: partition
 TYPE(MPI\_Request), INTENT(INOUT) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_PREADY(PARTITION, REQUEST, IERROR) INTEGER PARTITION, REQUEST, IERROR

MPI\_PREADY is a send-side call that indicates that a given partition is ready to be transferred. It is erroneous to use MPI\_PREADY on any request object that does not correspond to a partitioned send operation. The partitioning is defined by the MPI\_PSEND\_INIT call. Partition numbering starts at zero and ranges to one less than the number of partitions declared in the MPI\_PSEND\_INIT call. Specifying a partition number that is equal to or larger than the number of partitions is erroneous. After a call to MPI\_START/MPI\_STARTALL, all partitions associated with that operation are inactive. A call to MPI\_PREADY marks the indicated partition as active. Calling MPI\_PREADY on an active partition is erroneous.

MPI_PRE	ADY_RANGE(partition_	_low, partition_high, request)	23
IN	partition_low	partition to mark lowest partition ready for transfer	24
		(non-negative integer)	25
IN	partition_high	partition to mark highest partition ready for transfer	26
		(non-negative integer)	27
INOUT	request	partitioned communication request (handle)	28
	request	partitioned communication request (namalo)	29
C bindir	ur i		30
	0	rtition_low, int partition_high,	31
IIIC MPI_			32
	MPI_Request *r	request)	33
Fortran	2008 binding		34
MPI_Prea	dy_range(partition_	low, partition_high, request, ierror)	35
INTE	GER, INTENT(IN) ::	partition_low, partition_high	36
TYPE	(MPI_Request), INTE	NT(INOUT) :: request	37
INTE	GER, OPTIONAL, INTE	NT(OUT) :: ierror	38
<b>D</b>	L J		39
Fortran	0		40
		LOW, PARTITION_HIGH, REQUEST, IERROR)	41
INTE	GER PARTITION_LUW,	PARTITION_HIGH, REQUEST, IERROR	42
A ca	ll to MPI_PREADY_R	ANGE has the same effect as calls to	43
MPI_PRE	ADY, executed for $i=p$	partition_low ,, partition_high, in some arbitrary order.	44
Calls to N	IPI_PREADY_RANGE	follow the same rules as those for MPI_PREADY calls.	45
			46
			47

	106	CHAPTER 4.	PARTIT	IONED POINT-TO-POINT COMMUNICATION	
1	MPI PRFA	.DY_LIST(length, ar	ray of par	titions request)	
2	IN	length	a)_opa.	list length (integer)	
3	INOUT	array_of_partitions			
4				array of partitions (array of non-negative integers)	
5 6	INOUT	request		partitioned communication request (handle)	
7	C binding	r			
8			ength, ir	t array_of_partitions[],	
9	-	MPI_Request	-		
10 11	Fortran 2	008 binding			
12			rray_of_p	partitions, request, ierror)	
13		ER, INTENT(IN) :			
14				hy_of_partitions(length)	
15		MPI_Request), INT ER, OPTIONAL, INT		-	
16 17				161101	
18	Fortran b				
19		-		PARTITIONS, REQUEST, IERROR) TIONS(*), REQUEST, IERROR	
20					
21 22				the same effect as calls to one specified in the range <i>array_of_partitions</i> [0]	
22				the array_of_partitions, executed in some arbitrary	
24	order. Calls to MPI_PREADY_LIST follow the same rules as those for MPI_PREADY calls.				
25					
26	4.2.2 Cor	nmunication Comp	letion und	er Partitioning	
27 28	The function	ons MPI_WAIT and	MPI_TES	$\Gamma$ (and variants) are used to complete a partitioned	
29		•		ion of a partitioned send operation indicates that	
30				RT/MPI_STARTALL to restart the operation and	
31				DY_RANGE or MPI_PREADY_LIST. Alternatively, communication request after the completion of the	
32				ocess, completion of the partitioned send operation	
33 34	-	-		he message have all been received.	
35				evive operation through MPI_WAIT or MPI_TEST	
36				s all of the partitions. A function for probing the	
37				provided by MPI_PARRIVED. The MPI_PARRIVED message data for the indicated partition has been	
38				success, the receiver becomes free to access the	
39 40			-	s that previously completed for that operation).	
40	-		0		
42					
43					
44					
45					
46 $47$					
48					

1.2. 01111			
MPI_PARR	RIVED(request, partition, flag)		1
INOUT	request	partitioned communication request (handle)	2
	•	, ,	3
IN	partition	partition to be tested (non-negative integer)	4
OUT	flag	true if operation completed on the specified partition,	5
		false if not (boolean)	6
			7 8
C binding			9
int MPI_P	arrived(MPI_Request *req	uest, int partition, int *flag)	10
Fortran 2	008 binding		11
	ved(request, partition, :	flag, ierror)	12
TYPE(	MPI_Request), INTENT(INO	UT) :: request	13
INTEG	ER, INTENT(IN) :: partit:	ion	14
	AL, INTENT(OUT) :: flag		15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			16
Fortran b	inding		17
	VED(REQUEST, PARTITION, 1	FLAG, IERROR)	18
	ER REQUEST, PARTITION, I		19
LOGIC	AL FLAG		20
Tho f	unction MPI PARRIVED con	be used to test partial completion of partitioned	21 22
		RRIVED on an active partitioned communications	22
-		tion identified by request for the specified	20
*	0	ot marked as complete/inactive by this operation.	25
		peration is required to complete the message, as	26
-		ED may be called multiple times for a partition.	27
MPI_PARR	RIVED may be called with a	null or inactive request argument. In either case,	28
the operati	ion returns with $flag = true$ .	Calling MPI_PARRIVED on a request that does not	29
correspond	to a partitioned receive oper	ation is erroneous.	30
			31
4.2.3 Ser	nantics of Communications i	n Partitioned Mode	32
The seman	tics of nonblocking partitione	d communication are defined by suitably extending	33
	ons in Section $3.5$ .	a communication are defined by suitably extending	34
ane demili			35

Interpretation of count and datatype for partitioned communication Partitioned communica-37 tion uses the count and datatype arguments in the partitioned communication initialization 38 functions to describe a single partition. The argument partitions specifies how many equal 39 partitions of a number (count) of datatypes make up the entire buffer to be transferred in the partitioned communication. As partitioned communication describes many partitions, 41 using absolute displacements in datatypes (e.g., MPI\_BOTTOM) is not supported. Partitions 42are contiguous in memory, there is no padding in between partitions. Once a partitioned 43 send operation is started, each partition must be marked as ready using MPI\_PREADY and the operation must be completed using a completion function, such as MPI\_TEST or MPI\_WAIT.

4647

44

45

35 36

40

Order Matching follows the same MPI matching rules as for point-to-point communication (see Chapter 3) with communicator, tag and source dictating message matching. In the event that the communicator, tag and source do not uniquely identify the message, the order in which partitioned communication initialization calls are made is the order in which they will eventually match.

## 4.3 Partitioned Communication Examples

This section provides concrete examples of the utility of partitioned communication in realistic settings.

```
4.3.1 Partition Communication with Threads/Tasks Using OpenMP 4.0 and greater
```

The equal partitioning on send-side and receive-side in Example 4.1 is shown using threads. In this case, the receive-side uses the same number of partitions as the sending-side like the previous example, but this example uses multiple threads on the sending-side. Note that the MPI\_PSEND\_INIT and MPI\_PRECV\_INIT functions match each other like in the previous example.

```
<sup>20</sup> Example 4.2
```

```
21
     #include "mpi.h"
22
     #define NUM_THREADS 8
23
     #define PARTITIONS 8
24
     #define PARTLENGTH 16
25
     int main( int argc, char *argv[]) /* same send/recv partitioning */
26
     {
27
       double message[PARTITIONS*PARTLENGTH];
28
       int partitions = PARTITIONS;
29
       int partlength = PARTLENGTH;
30
       int count = 1, source = 0, dest = 1, tag = 1, flag = 0;
31
       int myrank;
32
       int provided;
33
       MPI_Request request;
34
       MPI_Info info = MPI_INFO_NULL;
35
       MPI_Datatype xfer_type;
36
       MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
37
       if (provided < MPI_THREAD_SERIALIZED)</pre>
38
          MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
39
       MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
40
       MPI_Type_contiguous(partlength, MPI_DOUBLE, &xfer_type);
41
       MPI_Type_commit(&xfer_type);
42
       if (myrank == 0)
                            /* code for process zero */
43
       {
44
          MPI_Psend_init(message, partitions, count, xfer_type, dest, tag,
45
                info, MPI_COMM_WORLD, &request);
46
          MPI_Start(&request);
47
48
```

6 7

8 9

10

11 12

13

```
#pragma omp parallel for shared(request) num_threads(NUM_THREADS)
     for (int i=0; i<partitions; i++)</pre>
     {
        /* compute and fill partition #i, then mark ready: */
        MPI_Pready(i, &request);
     }
     while(!flag)
     {
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* Do useful work */
     }
     MPI_Request_free(&request);
  }
  else if (myrank == 1) /* code for process one */
  {
     MPI_Precv_init(message, partitions, count, xfer_type, source, tag,
            info, MPI_COMM_WORLD, &request);
     MPI_Start(&request);
     while(!flag)
                                                                                      21
     {
                                                                                      22
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                      23
        /* Do useful work */
     }
     MPI_Request_free(&request);
  }
  MPI_Finalize();
                                                                                      29
  return 0;
                                                                                      30
}
      Send-only Partitioning Example with Tasks and OpenMP version 4.0 and greater
4.3.2
                                                                                      33
The previous example is tailored specifically for send-side partitioning using threads. This
                                                                                      34
is an example where parallel task producers produce input to part of an overall buffer; they
                                                                                      35
complete in any order and contribute to the overall buffer.
                                                                                      36
                                                                                      37
Example 4.3
#include "mpi.h"
#define NUM_THREADS 8
#define NUM_TASKS 64
```

```
#define PARTITIONS NUM_TASKS
#define PARTLENGTH 16
```

```
44
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
                                                                                    45
int main( int argc, char *argv[]) /* send-side partitioning */
                                                                                    46
ſ
                                                                                    47
 double message[MESSAGE_LENGTH];
                                                                                    48
 int send_partitions = PARTITIONS,
```

1

2

5

6

9

10

11

12

13 14

15

16

17

18

19

20

 $^{24}$ 

25

26

27

28

3132

38 39

40

41

42

```
1
           send_partlength = PARTLENGTH,
2
           recv_partitions = 1,
3
           recv_partlength = PARTITIONS*PARTLENGTH;
4
       int count = 1, source = 0, dest = 1, tag = 1, flag = 0;
5
       int myrank;
6
       int provided;
7
       MPI_Request request;
8
       MPI_Info info = MPI_INFO_NULL;
9
       MPI_Datatype send_type;
10
       MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
11
       if (provided < MPI_THREAD_SERIALIZED)
12
          MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
13
       MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
14
       MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);
15
       MPI_Type_commit(&send_type);
16
17
       if (myrank == 0)
                            /* code for process zero */
18
       {
19
          MPI_Psend_init(message, send_partitions, count, send_type, dest, tag,
20
                     info, MPI_COMM_WORLD, &request);
21
          MPI_Start(&request);
22
23
          #pragma omp parallel shared(request) num_threads(NUM_THREADS)
24
          {
25
             #pragma omp single
26
             Ł
27
               /* single thread creates 64 tasks to be executed by 8 threads */
28
               for (int partition_num=0; partition_num<NUM_TASKS; partition_num++)
29
               ſ
30
                   #pragma omp task firstprivate(partition_num)
31
                   {
32
                    /* compute and fill partition #partition_num, then mark
33
                       ready: */
34
                    /* buffer is filled in arbitrary order from each task */
35
                    MPI_Pready(partition_num, &request);
36
                   } /*end task*/
37
               } /* end for */
38
             } /* end single */
39
          } /* end parallel */
40
          while(!flag)
41
          {
42
             /* Do useful work */
43
             MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
44
             /* Do useful work */
45
          }
46
          MPI_Request_free(&request);
47
       }
48
       else if (myrank == 1) /* code for process one */
```

```
{
    MPI_Precv_init(message, recv_partitions, recv_partlength, MPI_DOUBLE,
        source, tag, info, MPI_COMM_WORLD, &request);

    MPI_Start(&request);
    while(!flag)
    {
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* Do useful work */
     }
     MPI_Request_free(&request);
}
MPI_Finalize();
return 0;
}
```

### 4.3.3 Send and Receive Partitioning Example with OpenMP version 4.0 and greater

This example demonstrates receive-side partial completion notification using more than one partition per receive-side thread. It uses a naive flag based method to test for multiple completed partitions per thread. Note that this means that some threads may be busy polling for completion of assigned partitions when partitions are available to work on that were not assigned to the polling threads in this example. More advanced work stealing methods could be employed for greater efficiency. Like previous examples, it also demonstrates send-side production of input to part of an overall buffer. This example also uses different send-side and receive-side partitioning.

### Example 4.4

```
#include "mpi.h"
                                                                                     30
                                                                                     31
#define NUM_THREADS 64
                                                                                     32
#define PARTITIONS NUM_THREADS
                                                                                     33
#define PARTLENGTH 16
                                                                                     34
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
int main( int argc, char *argv[]) /* send-side partitioning */
                                                                                     35
                                                                                     36
{
                                                                                     37
  double message[MESSAGE_LENGTH];
                                                                                     38
  int send_partitions = PARTITIONS,
                                                                                     39
      send_partlength = PARTLENGTH,
                                                                                     40
      recv_partitions = PARTITIONS*2,
                                                                                     41
      recv_partlength = PARTLENGTH/2;
                                                                                     42
  int source = 0, dest = 1, tag = 1, flag = 0;
  int myrank;
                                                                                     43
                                                                                     44
  int provided;
                                                                                     45
 MPI_Request request;
                                                                                     46
  MPI_Info info = MPI_INFO_NULL;
                                                                                     47
  MPI_Datatype send_type;
                                                                                     48
  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
```

### CHAPTER 4. PARTITIONED POINT-TO-POINT COMMUNICATION

112

```
1
       if (provided < MPI_THREAD_SERIALIZED)
2
          MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
3
       MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
4
       MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);
5
       MPI_Type_commit(&send_type);
6
7
       if (myrank == 0)
                            /* code for process zero */
8
       {
9
          MPI_Psend_init(message, send_partitions, 1, send_type, dest, tag,
10
                     info, MPI_COMM_WORLD, &request);
11
          MPI_Start(&request);
12
          #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
13
          for (int i=0; i<send_partitions; i++)</pre>
14
          ſ
15
              /* compute and fill partition #i, then mark ready: */
16
             MPI_Pready(i, &request);
17
          }
18
          while(!flag)
19
          ſ
20
             /* Do useful work */
21
             MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
22
             /* Do useful work */
23
          }
24
          MPI_Request_free(&request);
25
       }
26
       else if (myrank == 1) /* code for process one */
27
       {
28
          MPI_Precv_init(message, recv_partitions, recv_partlength, MPI_DOUBLE,
29
                     source, tag, info, MPI_COMM_WORLD, &request);
30
          MPI_Start(&request);
31
          #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
32
          for (int j=0; j<recv_partitions; j+=2)</pre>
33
          Ł
34
              int part1_complete = 0;
35
              int part2_complete = 0;
36
              while(part1_complete == 0 || part2_complete == 0)
37
              {
38
                 /* test partition #j and #j+1 */
39
                 MPI_Parrived(&request, j, &flag);
40
                 if(flag && part1_complete == 0)
41
                 {
42
                    part1_complete++;
43
                    /* Do work using partition j data */
44
                 }
45
                 if (j+1 < recv_partitions) {</pre>
46
                   MPI_Parrived(&request, j+1, &flag);
47
                   if(flag && part2_complete == 0)
48
                   ſ
```

```
part2_complete++;
                                                                                                    1
                                                                                                    \mathbf{2}
                    /* Do work using partition j+1 */
                }
                                                                                                    3
             }
                                                                                                    4
                                                                                                    5
             else {
                  part2_complete++;
                                                                                                    6
             }
                                                                                                    7
         }
                                                                                                    8
      }
                                                                                                    9
                                                                                                    10
      while(!flag)
                                                                                                    11
      {
                                                                                                    12
         /* Do useful work */
         MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
                                                                                                    13
         /* Do useful work */
                                                                                                    14
      }
                                                                                                    15
                                                                                                    16
      MPI_Request_free(&request);
  }
                                                                                                    17
                                                                                                    18
  MPI_Finalize();
  return 0;
                                                                                                    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 5

## 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.

## 5.1 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.

A general datatype is an opaque object that specifies two things:	45
• A sequence of basic datatypes	46
• A sequence of integer (byte) displacements	47 48

 $^{24}$ 

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

11 12

13

14

15

16

17

18

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

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

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

be the associated type signature. This type map, together with a base address **buf**, specifies a communication buffer: the communication buffer that consists of n entries, where the *i*-th entry is at address **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 Typesig.

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.

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

General datatypes can be used in all send and receive operations. We discuss, in Section 5.1.11, the case where the second argument count has value > 1.

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

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

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

then

11 (m

41 42

32

33

34 35 36

37 38

39 40

<sup>43</sup> If  $type_j$  requires alignment to a byte address that is a multiple of  $k_j$ , then  $\epsilon$  is the least <sup>44</sup> non-negative increment needed to round extent(Typemap) to the next multiple of  $\max_j k_j$ . <sup>45</sup> In Fortran, it is implementation dependent whether the MPI implementation computes <sup>46</sup> the alignments  $k_j$  according to the alignments used by the compiler in common blocks, <sup>47</sup> SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE <sup>48</sup> nor BIND(C). The complete definition of **extent** is given by Equation 5.1 Section 5.1.

### Unofficial Draft for Comment Only

Let

**Example 5.1** 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 5.1.6. 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 5.1.6 and in Section 19.1.15. (End of rationale.)

### 5.1.1 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. For Fortran compilers 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 (assuming the Fortran compiler accepts the common extension of INTEGER\*8 for eight-byte integers).

For the large count versions of three datatype constructors with explicit addresses, MPI\_TYPE\_CREATE\_HINDEXED, MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK, and MPI\_TYPE\_CREATE\_STRUCT, absolute addresses shall not be used to specify byte displacements since the parameter is of type MPI\_COUNT instead of type MPI\_AINT.

### 5.1.2 Datatype Constructors

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

			37
MPI_TYP	E_CONTIGUOUS(count, oldty	vpe, newtype)	38
IN	count	replication count (non-negative integer)	39
INI	aldtura		40
IN	oldtype	old datatype (handle)	41
OUT	newtype	new datatype (handle)	42
			43
C bindir	ទេ		44
	-S Type_contiguous(int count	t. MPI Datatype oldtype.	45
<b></b>	MPI_Datatype *newty		46
		r - ,	47

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

```
1
            int MPI_Type_contiguous_c(MPI_Count count, MPI_Datatype oldtype,
\mathbf{2}
                                             MPI_Datatype *newtype)
 3
            Fortran 2008 binding
 4
            MPI_Type_contiguous(count, oldtype, newtype, ierror)
5
                      INTEGER, INTENT(IN) :: count
6
                      TYPE(MPI_Datatype), INTENT(IN) :: oldtype
 7
                      TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 8
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
            MPI_Type_contiguous(count, oldtype, newtype, ierror)
11
                      INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
12
                      TYPE(MPI_Datatype), INTENT(IN) :: oldtype
13
                      TYPE(MPI_Datatype), INTENT(OUT) :: newtype
14
                      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
            Fortran binding
16
            MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
17
                      INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
18
19
                      newtype is the datatype obtained by concatenating count copies of
20
            oldtype. Concatenation is defined using extent as the size of the concatenated copies.
21
            Example 5.2 Let oldtype have type map \{(double, 0), (char, 8)\}, with extent 16, and let
22
            count = 3. The type map of the datatype returned by newtype is
23
24
                        \{(double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40)\};
25
26
            i.e., alternating double and char elements, with displacements 0, 8, 16, 24, 32, 40.
27
                      In general, assume that the type map of oldtype is
28
29
                        \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\
30
            with extent ex. Then newtype has a type map with count \cdot n entries defined by:
31
32
                   \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots
33
34
                 \ldots, (type_0, disp_0 + ex \cdot (\mathsf{count} - 1)), \ldots, (type_{n-1}, disp_{n-1} + ex \cdot (\mathsf{count} - 1))\}.
35
36
            Vector The function MPI_TYPE_VECTOR is a more general constructor that allows repli-
37
            cation of a datatype into locations that consist of equally spaced blocks. Each block is
38
            obtained by concatenating the same number of copies of the old datatype. The spacing
39
            between blocks is a multiple of the extent of the old datatype.
40
41
42
43
44
45
46
47
48
```

MPI_TYP	E_VECTOR(count, blocklength,	stride, oldtype, newtype)	1
IN	count	number of blocks (non-negative integer)	2
IN	blocklength	number of elements in each block (non-negative integer)	3 4 5
IN	stride	number of elements between start of each block (integer)	6 7
IN	oldtype	old datatype (handle)	8
OUT	newtype	new datatype (handle)	9 10
		in an and the second seco	11
C bindin	g		12
	Gype_vector(int count, int	; blocklength, int stride, , MPI_Datatype *newtype)	13 14
int MPT '	Fune vector c(MPT Count co	ount, MPI_Count blocklength,	15
1110 III 1		I_Datatype oldtype, MPI_Datatype *newtype)	16
<b>T</b> (			17 18
	2008 binding	stride eldture reuture ienner)	18 19
	GER, INTENT(IN) :: count,	n, stride, oldtype, newtype, ierror)	20
	(MPI_Datatype), INTENT(IN)	5	21
	(MPI_Datatype), INTENT(OUT		22
	GER, OPTIONAL, INTENT(OUT)		23
MDT Turne	wester(count blocklength	a, stride, oldtype, newtype, ierror)	24
	÷	INTENT(IN) :: count, blocklength, stride	25
	(MPI_Datatype), INTENT(IN)	e e e e e e e e e e e e e e e e e e e	26
	(MPI_Datatype), INTENT(OUT		27 28
INTEG	GER, OPTIONAL, INTENT(OUT)	:: ierror	20
Fortran l	pinding		30
	J	I, STRIDE, OLDTYPE, NEWTYPE, IERROR)	31
		RIDE, OLDTYPE, NEWTYPE, IERROR	32
			33
Example	5.3 Assume, again, that old	Stype has type map $\{(\texttt{double}, 0), (\texttt{char}, 8)\}$ , with	34
		R(2, 3, 4, oldtype, newtype) will create the datatype	35
with type			36 37
[(da	(abar 8) (double 16)	(abar 24) $(dauble 22)$ $(abar 40)$	38
{(αο	(double, 0), (char, 8), (double, 10)	), (char, 24), (double, 32), (char, 40),	39
(dou	ble, 64), (char, 72), (double, 80)	$0), (char, 88), (double, 96), (char, 104)\}.$	40
× ×			41
	_	h of the old type, with a stride of 4 elements $(4 \cdot 16)$	42
bytes) bet	ween the the start of each bloc	ek.	43
Example	5.4 A call to MPI TYPF V	ECTOR(3, 1, -2, oldtype, newtype) will create the	44
datatype,	· · · · · · · · · · · · · · · · · · ·		45 46
			40
{(do	uble, 0), (char, 8), (double, -3)	$(char, -24), (double, -64), (char, -56)\}.$	48

1	In ge	neral, assume that <b>oldtype</b> has	type map,	
2 3	$\{(ty$	$pe_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$_{n-1})\},$	
4 5 6		nt $ex$ . Let bl be the blocklength n entries:	a. The newly created datatype has a type map with	
7	$\{(ty$	$pe_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$_{n-1}),$	
9	$(type_0, disp_0 + ex), \ldots, (type_{n-1}, disp_{n-1} + ex), \ldots,$			
10 11	$(type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),$			
12 13	(typ	$be_0, disp_0 + stride \cdot ex), \dots, (typ_0)$	$pe_{n-1}, disp_{n-1} + stride \cdot ex), \ldots,$	
14 15	(typ	$e_0, disp_0 + (stride + bl - 1) \cdot e_0$	$(x), \ldots, (type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex), \ldots,$	
16 17	(typ	$e_0, disp_0 + stride \cdot (count - 1) \cdot $	$ex),\ldots,$	
18 19	(typ	$e_{n-1}, disp_{n-1} + stride \cdot (count + stride)$	$(-1) \cdot ex), \ldots,$	
20 21	(typ	$e_0, disp_0 + (stride \cdot (count - 1))$	$+ bl - 1) \cdot ex), \dots,$	
22 23	$(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}.$			
24 25 26	A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1,			
27 28 29 30 31 32	MPI_TYP	E_VECTOR, except that strid oth types of vector constructor	EATE_HVECTOR is identical to e is given in bytes, rather than in elements. The ors is illustrated in Section 5.1.14. (H stands for	
33 34	MPI_TYP	E_CREATE_HVECTOR(count,	blocklength, stride, oldtype, newtype)	
35	IN	count	number of blocks (non-negative integer)	
36 37	IN	blocklength	number of elements in each block (non-negative integer)	
38 39	IN	stride	number of bytes between start of each block (integer)	
40	IN	oldtype	old datatype (handle)	
41 42	OUT	newtype	new datatype (handle)	
43 44 45 46	C bindin int MPI_	Type_create_hvector(int c	ount, int blocklength, MPI_Aint stride, , MPI_Datatype *newtype)	
47 48	int MPI_	• 1	_Count count, MPI_Count blocklength, YI_Datatype oldtype, MPI_Datatype *newtype)	

```
1
Fortran 2008 binding
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
                                                                                                        \mathbf{2}
                 ierror)
     INTEGER, INTENT(IN) :: count, blocklength
     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                                        6
     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
                                                                                                        10
                 ierror)
                                                                                                        11
     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, stride
                                                                                                        12
     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                                        13
     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                                        14
     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                                        15
                                                                                                        16
Fortran binding
                                                                                                        17
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
                                                                                                        18
                 IERROR)
                                                                                                        19
     INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
     INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
                                                                                                        20
                                                                                                       21
     Assume that oldtype has type map,
                                                                                                        22
                                                                                                        23
      \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\
                                                                                                        ^{24}
with extent ex. Let bl be the blocklength. The newly created datatype has a type map with
                                                                                                        25
count \cdot bl \cdot n entries:
                                                                                                        26
                                                                                                        27
      \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), \}
                                                                                                        28
                                                                                                        29
      (type_0, disp_0 + ex), \ldots, (type_{n-1}, disp_{n-1} + ex), \ldots,
                                                                                                        30
                                                                                                        31
      (type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),
                                                                                                        32
                                                                                                        33
      (type_0, disp_0 + \mathsf{stride}), \dots, (type_{n-1}, disp_{n-1} + \mathsf{stride}), \dots,
                                                                                                       34
                                                                                                       35
      (type_0, disp_0 + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), \ldots,
                                                                                                        36
      (type_{n-1}, disp_{n-1} + stride + (bl - 1) \cdot ex), \ldots,
                                                                                                        37
                                                                                                        38
      (type_0, disp_0 + stride \cdot (count - 1)), \dots, (type_{n-1}, disp_{n-1} + stride \cdot (count - 1)), \dots,
                                                                                                        39
                                                                                                        40
      (type_0, disp_0 + stride \cdot (count - 1) + (bl - 1) \cdot ex), \ldots,
                                                                                                        41
                                                                                                        42
      (type_{n-1}, disp_{n-1} + stride \cdot (count - 1) + (bl - 1) \cdot ex)\}.
                                                                                                        43
                                                                                                        44
                                                                                                        45
                                                                                                        46
                                                                                                        47
                                                                                                        48
```

```
122
                                                                          CHAPTER 5. DATATYPES
           1
                Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a
           \mathbf{2}
                sequence of blocks (each block is a concatenation of the old datatype), where each block
           3
                can contain a different number of copies and have a different displacement. All block
           4
                displacements are multiples of the old type extent.
           5
           6
                MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype,
           7
                               newtype)
           8
                                              Substitute in Latex source " --- " by "---"
#-update1
                  IN
                                                        number of blocks – also number of entries in
                            count
           10
                                                        array_of_displacements and array_of_blocklengths
#-405
           11
                                                        (non-negative integer)
#-PR427
           12
                  IN
                            array_of_blocklengths
                                                        number of elements per block (array of non-negative
           13
Done
                                                        integers)
           14
                            array_of_displacements
                                                        displacement for each block, in multiples of oldtype
                  IN
           15
                                                        (array of integers)
           16
           17
                  IN
                            oldtype
                                                        old datatype (handle)
           18
                  OUT
                            newtype
                                                        new datatype (handle)
           19
           20
                C binding
           21
                int MPI_Type_indexed(int count, const int array_of_blocklengths[],
           22
                               const int array_of_displacements[], MPI_Datatype oldtype,
           23
                               MPI_Datatype *newtype)
           24
           25
                int MPI_Type_indexed_c(MPI_Count count,
           26
                               const MPI_Count array_of_blocklengths[],
           27
                               const MPI_Count array_of_displacements[],
           28
                               MPI_Datatype oldtype, MPI_Datatype *newtype)
           29
                Fortran 2008 binding
           30
                MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
           ^{31}
                               oldtype, newtype, ierror)
           32
                     INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
           33
                                array_of_displacements(count)
           34
                     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
           35
                     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
           36
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           37
           38
                MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
           39
                               oldtype, newtype, ierror)
           40
                     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
           41
                                array_of_blocklengths(count), array_of_displacements(count)
           42
                     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
           43
                     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
           44
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           45
                Fortran binding
           46
                MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
           47
                               OLDTYPE, NEWTYPE, IERROR)
           48
```

```
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
          OLDTYPE, NEWTYPE, IERROR
```

**Example 5.5** Let oldtype have type map  $\{(double, 0), (char, 8)\}$ , with extent 16. Let B =(3, 1) and let D = (4, 0). A call to MPI\_TYPE\_INDEXED(2, B, D, oldtype, newtype) returns a datatype with type map,

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

(double, 0), (char, 8).

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

In general, assume that oldtype has type map,

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

with extent *ex*. Let B be the array\_of\_blocklengths argument and D be the array\_of\_displacements argument. The newly created datatype has  $n \cdot \sum_{i=0}^{\text{count}-1} B[i]$  entries:

$$\{(type_0, disp_0 + \mathsf{D}[0] \cdot ex), \dots, (type_{n-1}, disp_{n-1} + \mathsf{D}[0] \cdot ex), \dots,$$

 $(type_0, disp_0 + (D[0] + B[0] - 1) \cdot ex), \dots,$  $(type_{n-1}, disp_{n-1} + (D[0] + B[0] - 1) \cdot ex), \dots,$  $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}] \cdot ex), \dots, (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}] \cdot ex), \dots,$ 

$$(type_0, dsp_0 + (\mathsf{D}[\mathsf{count-1}] + \mathsf{B}[\mathsf{count-1}] - 1) \cdot ex), \dots,$$

$$(type_{n-1}, disp_{n-1} + (\mathsf{D}[\mathsf{count-1}] + \mathsf{B}[\mathsf{count-1}] - 1) \cdot ex)\}.$$

A call to MPI\_TYPE\_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent to a call to MPI\_TYPE\_INDEXED(count, B, D, oldtype, newtype) where

$$\mathsf{D}[\mathsf{j}] = j \cdot \mathsf{stride}, \ j = 0, \dots, \mathsf{count} - 1,$$

and

 $B[j] = blocklength, j = 0, \dots, count - 1.$ 

Hindexed The function MPI\_TYPE\_CREATE\_HINDEXED is identical to MPI\_TYPE\_INDEXED, except that block displacements in array\_of\_displacements are specified in bytes, rather than in multiples of the oldtype extent.

 $^{24}$ 

```
MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklengths, array_of_displacements,
           2
                               oldtype, newtype)
                                                 Substitute in Latex source " --- " by "---"
                                                       number of blocks – also number of entries in
#-update1
                  IN
                            count
                                                       array_of_displacements and array_of_blocklengths
#-405
           5
                                                       (non-negative integer)
#-PR427
           6
Done
                  IN
                            array_of_blocklengths
                                                       number of elements in each block (array of
           \overline{7}
                                                       non-negative integers)
           8
           9
                  IN
                            array_of_displacements
                                                       byte displacement of each block (array of integers)
           10
                  IN
                            oldtype
                                                       old datatype (handle)
           11
                  OUT
                                                       new datatype (handle)
                            newtype
           12
           13
           14
                C binding
                int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],
           15
           16
                               const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
           17
                               MPI_Datatype *newtype)
           18
                int MPI_Type_create_hindexed_c(MPI_Count count,
           19
                               const MPI_Count array_of_blocklengths[],
           20
                               const MPI_Count array_of_displacements[],
           21
                               MPI_Datatype oldtype, MPI_Datatype *newtype)
           22
           23
                Fortran 2008 binding
           24
                MPI_Type_create_hindexed(count, array_of_blocklengths,
           25
                               array_of_displacements, oldtype, newtype, ierror)
           26
                     INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
           27
                     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                array_of_displacements(count)
           28
                    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
           29
                    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
           30
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           31
           32
                MPI_Type_create_hindexed(count, array_of_blocklengths,
           33
                               array_of_displacements, oldtype, newtype, ierror)
           34
                     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
           35
                                array_of_blocklengths(count), array_of_displacements(count)
           36
                    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
           37
                    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
           38
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           39
           40
                Fortran binding
           41
                MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
           42
                               ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
           43
                     INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
           44
                     INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
           45
                    Assume that oldtype has type map,
           46
           47
                     \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\
           48
```

with extent ex. Let B be the array\_of\_blocklengths argument and D be the array\_of\_displacements argument. The newly created datatype has a type map with  $n \cdot \sum_{i=0}^{count-1} B[i]$  entries:

$$\{(type_0, disp_0 + D[0]), \dots, (type_{n-1}, disp_{n-1} + D[0]), \dots, \\ (type_0, disp_0 + D[0] + (B[0] - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1) \cdot ex), \dots, \\ (type_0, disp_0 + D[count-1]), \dots, (type_{n-1}, disp_{n-1} + D[count-1]), \dots, \\ (type_0, disp_0 + D[count-1] + (B[count-1] - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + D[count-1] + (B[count-1] - 1) \cdot ex)\}.$$

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

			21
MPI_TYF		K(count, blocklength, array_of_displacements,	22
		s already modified into: number of blocks also number of	23
IN	count	ries in array_of_displacements length of array of displacements (non-negative	<sup>24</sup> #-update1
	count	integer) Substitute in Latex source " " by ""	<sup>25</sup> #-405
INI			
IN	blocklength	size of block (non-negative integer)	<sup>27</sup> <b>Done</b>
IN	array_of_displacements	array of displacements (array of integers)	28
IN	oldtype	old datatype (handle)	29
OUT	newtype	new datatype (handle)	30
001	newtype	new datatype (nandle)	31
			32
C bindi	3		33
int MPI_	<i>v</i> <b>1</b>	k(int count, int blocklength,	34
		<pre>displacements[], MPI_Datatype oldtype,</pre>	35
	MPI_Datatype *newty	pe)	36
int MPI_	Type_create_indexed_bloc	k_c(MPI_Count count, MPI_Count blocklength,	37
		ay_of_displacements[],	38
	MPI_Datatype oldtyp	e, MPI_Datatype *newtype)	39
<b>T</b> (			40
	2008 binding		41
MP1_Type		unt, blocklength, array_of_displacements,	42
T 11001	oldtype, newtype, i		43
INIE	EGER, INTENT(IN) :: count	5	44
	array_of_displaceme		45
	E(MPI_Datatype), INTENT(I)	<i>v</i> 1	46
	E(MPI_Datatype), INTENT(O	V1	47
TNIF	EGER, OPTIONAL, INTENT(OU	1) :: lerror	48

Unofficial Draft for Comment Only

 $1 \\ 2$ 

 $13 \\ 14$ 

```
1
                MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
           2
                               oldtype, newtype, ierror)
           3
                    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength,
           4
                                array_of_displacements(count)
           5
                    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
           6
                    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
           7
                    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           8
                Fortran binding
           9
                MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
           10
                               OLDTYPE, NEWTYPE, IERROR)
           11
                    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
           12
                                NEWTYPE, IERROR
           13
           14
           15
                Hindexed_block The function MPI_TYPE_CREATE_HINDEXED_BLOCK is identical to
           16
                MPI_TYPE_CREATE_INDEXED_BLOCK, except that block displacements in
           17
                array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.
           18
           19
                MPI_TYPE_CREATE_HINDEXED_BLOCK(count, blocklength, array_of_displacements,
           20
                                                 Was already modified into: number of blocks --- also number of
           21
                               oldtype, newtype)
                                                 entries in array_of_displacements
#-update1 22
                                                       length of array of displacements (non-negative
                  IN
                           count
           23
                                                       integer) Substitute in Latex source " --- " by "---"
           ^{24}
#-PR427
                  IN
                           blocklength
                                                       size of block (non-negative integer)
           25
           26
                  IN
                           array_of_displacements
                                                       byte displacement of each block (array of integers)
           27
                  IN
                           oldtype
                                                       old datatype (handle)
           28
                  OUT
                                                       new datatype (handle)
                           newtype
           29
           30
                C binding
           ^{31}
                int MPI_Type_create_hindexed_block(int count, int blocklength,
           32
                               const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
           33
                               MPI_Datatype *newtype)
           34
           35
                int MPI_Type_create_hindexed_block_c(MPI_Count count,
           36
                               MPI_Count blocklength,
           37
                               const MPI_Count array_of_displacements[],
           38
                               MPI_Datatype oldtype, MPI_Datatype *newtype)
           39
                Fortran 2008 binding
           40
                MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
           41
                               oldtype, newtype, ierror)
           42
                    INTEGER, INTENT(IN) :: count, blocklength
           43
                    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
           44
                                array_of_displacements(count)
           45
                    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
           46
                    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
           47
                     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
           48
```

#-405

Done

MPI_Type	e_create_hindexed_block(c	ount, blocklength, array_of_displacements,	1	
	oldtype, newtype, i		2	
INTE		, INTENT(IN) :: count, blocklength,	3	
	array_of_displacem		4	
	E(MPI_Datatype), INTENT(I	• -	5	
	E(MPI_Datatype), INTENT(O		6 7	
TNLE	EGER, OPTIONAL, INTENT(OU	T) :: lerror	8	
Fortran	binding		9	
MPI_TYPE	E_CREATE_HINDEXED_BLOCK(C	OUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	10	
	OLDTYPE, NEWTYPE, 1		11	
		OLDTYPE, NEWTYPE, IERROR	12	
INTH	EGER(KIND=MPI_ADDRESS_KIN	D) ARRAY_OF_DISPLACEMENTS(*)	13	
			14	
Struct N	MPL TYPE CREATE STRUC	<b>Γ</b> is the most general type constructor. It further	15	
		DEXED in that it allows each block to consist of repli-	16	
0	f different datatypes.		17	
	0 1		18	
			19	#-update1
MPI_IYI	PE_CREATE_STRUCT(count,	array_of_blocklengths, array_of_displacements, as already modified into: number of blocks also number of	20	#-405
	array_of_types, newt <sup>we</sup>	tries in arrays	21 22	
IN	count	number of blocks also number of entries in arrays	22	#-PR427
		array_of_types, array_of_displacements, and	23 24	Done
		array_of_blocklengths (non-negative integer)		
IN	array_of_blocklengths	Substitute in Latex source " " by	26	
		non-negative integers)	27	
IN	array_of_displacements	byte displacement of each block (array of integers)	28	
IN	array_of_types	types of elements in each block (array of handles)	29	
OUT	newtype	new datatype (handle)	30	
			31 32	
C bindi	ng		33	
		<pre>ount, const int array_of_blocklengths[],</pre>	34	
		ay_of_displacements[],	35	
		array_of_types[], MPI_Datatype *newtype)	36	
			37	
int MPI_	_Type_create_struct_c(MPI	_count count, ray_of_blocklengths[],	38	
		ray_of_displacements[],	39	
	· · · · · · · · · · · · · · · · · · ·	array_of_types[], MPI_Datatype *newtype)	40	
		array_or_oypes(), in r_bababype inewoype)	41	
	2008 binding		42	
MPI_Type	e_create_struct(count, ar		43	
7 1100		ents, array_of_types, newtype, ierror)	44	
		, array_of_blocklengths(count)	45	
11/11	EGER(KIND=MPI_ADDRESS_KIN array_of_displacem		46 47	
ͲϒϽΪ		N) :: array_of_types(count)	47 48	
1 1 1 1	L I_Datatype, INIDAI(I	., array_or_oypob(count)	20	

```
1
            TYPE(MPI_Datatype), INTENT(OUT) :: newtype
\mathbf{2}
            INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
       MPI_Type_create_struct(count, array_of_blocklengths,
4
                          array_of_displacements, array_of_types, newtype, ierror)
5
            INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
6
                           array_of_blocklengths(count), array_of_displacements(count)
7
            TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
8
            TYPE(MPI_Datatype), INTENT(OUT) :: newtype
9
            INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
       Fortran binding
12
       MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
13
                          ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
14
            INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
15
                           IERROR
16
             INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
17
18
       Example 5.6 Let type1 have type map.
19
20
              \{(double, 0), (char, 8)\},\
21
       with extent 16. Let B = (2, 1, 3), D = (0, 16, 26), and T = (MPI_FLOAT, type1, MPI_CHAR).
22
       Then a call to MPI_TYPE_CREATE_STRUCT(3, B, D, T, newtype) returns a datatype with
23
       type map,
24
25
              \{(\texttt{float}, 0), (\texttt{float}, 4), (\texttt{double}, 16), (\texttt{char}, 24), (\texttt{char}, 26), (\texttt{char}, 27), (\texttt{char}, 28)\}.
26
       That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at
27
       16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies
28
       four bytes.)
29
            In general, let T be the array_of_types argument, where T[i] is a handle to,
30
^{31}
             typemap_{i} = \{(type_{0}^{i}, disp_{0}^{i}), \dots, (type_{n_{i}-1}^{i}, disp_{n_{i}-1}^{i})\},\
32
33
       with extent ex_i. Let B be the array_of_blocklength argument and D be the
34
       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:
35
36
             \{(type_0^0, disp_0^0 + \mathsf{D}[0]), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0]), \dots, \}
37
38
             (type_0^0, disp_0^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots,
39
             (type_0^{\mathsf{C}-1}, disp_0^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1]), \dots, (type_{n\mathsf{C}-1}^{\mathsf{C}-1}, disp_{n\mathsf{C}-1}^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1]), \dots,
40
41
42
             (type_0^{\mathsf{C}-1}, disp_0^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1] + (\mathsf{B}[\mathsf{c}-1] - 1) \cdot ex_{\mathsf{C}-1}), \dots,
43
             (type_{nc}^{\mathsf{C}-1}, disp_{nc}^{\mathsf{C}-1}, -1 + \mathsf{D}[\mathsf{c}-1] + (\mathsf{B}[\mathsf{c}-1]-1) \cdot ex_{\mathsf{C}-1})\}.
44
45
46
            A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent
47
       to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T
48
       is equal to oldtype.
```

### 5.1. DERIVED DATATYPES

### 5.1.3 Subarray Datatype Constructor

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

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

### C binding

const MP1_Col	int array_or_	starts[],	int order,
MPI Datatype	oldtype, MPI	Datatvpe	*newtvpe)

### Fortran 2008 binding

### Fortran binding

MPI\_TYPE\_CREATE\_SUBARRAY(NDIMS, ARRAY\_OF\_SIZES, ARRAY\_OF\_SUBSIZES, ARRAY\_OF\_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)

1 2	<pre>INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),</pre>
3 4 5 6 7 8 9 10	The subarray type constructor creates an MPI datatype describing an <i>n</i> -dimensional subarray of an <i>n</i> -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 14.1.1. 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
11 12 13 14 15 16 17 18 19	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 <i>n</i> -dimensional ar- ray and the requested subarray are specified by array_of_sizes and array_of_subsizes, re- spectively. 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.
20 21	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]).
22 23 24 25	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. ( <i>End of advice to users.</i> )
26 27 28	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:
29	<b>MPI_ORDER_C</b> The ordering used by C arrays, (i.e., row-major order)
30 31	<b>MPI_ORDER_FORTRAN</b> The ordering used by Fortran arrays, (i.e., column-major order)
32 33 34	A ndims-dimensional subarray (newtype) with no extra padding can be defined by the function Subarray() as follows:
35 36 37 38	$\begin{array}{llllllllllllllllllllllllllllllllllll$
39 40	Let the typemap of <b>oldtype</b> have the form:
41	$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}$
42 43 44 45 46 47 48	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 5.2 defines the base step. Equation 5.3 defines the recursion step when order = MPI_ORDER_FORTRAN, and Equation 5.4 defines the recursion step when order = MPI_ORDER_C. These equations use the conceptual datatypes lb_marker and ub_marker; see Section 5.1.6 for details.

		1
		2
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\},\$	(5.2)	3
$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\})$		4
$= \{(lb_marker, 0),$		5
$(type_0, disp_0 + start_0 \times ex), \ldots, (type_{n-1}, disp_{n-1} + start_0 \times ex),$		6 7
$(type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1},$		8
$disp_{n-1} + (start_0 + 1) \times ex), \dots$		9
$(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \dots,$		10
$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$		11
$(ub\_marker, size_0 \times ex)$ }		12
		13 14
Subamar(adima (sina sina sina )	$(\mathbf{F}, 2)$	14
Subarray( $ndims$ , { $size_0, size_1, \ldots, size_{ndims-1}$ },	(5.3)	16
$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$	r	17
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$		18
$= \text{Subarray}(ndims - 1, \{size_1, size_2, \dots, size_{ndims - 1}\},\$		19
$\{subsize_1, subsize_2, \dots, subsize_{ndims-1}\},\$		20
$\{start_1, start_2, \dots, start_{ndims-1}\},\$		21
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$		22 23
		23 24
Subarray( $ndims$ , { $size_0, size_1, \ldots, size_{ndims-1}$ },	(5.4)	25
$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$	( )	26
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$		27
$= \text{Subarray}(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\}, $		28
		29
$\{subsize_0, subsize_1, \dots, subsize_{ndims-2}\},$		30
$\{start_0, start_1, \dots, start_{ndims-2}\},\$		31
$Subarray (1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldty)$	γpe))	32
		33

For an example use of MPI\_TYPE\_CREATE\_SUBARRAY in the context of I/O see Section 14.9.2.

#### 5.1.4Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [47] data distributions. However, unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call this constructor with identical arguments (with the exception of rank which should be set appropriately). These filetypes (along with identical disp and etype) are then used to define the view (via MPI\_FILE\_SET\_VIEW), see MPI I/O, especially Section 14.1.1 and Section 14.3. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (End of advice to users.)

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

```
1
     MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs,
\mathbf{2}
                     array_of_dargs, array_of_psizes, order, oldtype, newtype)
3
       IN
                 size
                                             size of process group (positive integer)
4
                                             rank in process group (non-negative integer)
       IN
                 rank
5
6
       IN
                 ndims
                                             number of array dimensions as well as process grid
7
                                             dimensions (positive integer)
8
       IN
                 array_of_gsizes
                                             number of elements of type oldtype in each dimension
9
                                             of global array (array of positive integers)
10
       IN
                 array_of_distribs
                                             distribution of array in each dimension (array of
11
                                             states)
12
13
       IN
                 array_of_dargs
                                             distribution argument in each dimension (array of
14
                                             positive integers)
15
       IN
                 array_of_psizes
                                             size of process grid in each dimension (array of
16
                                             positive integers)
17
                 order
                                             array storage order flag (state)
       IN
18
19
       IN
                 oldtype
                                             old datatype (handle)
20
       OUT
                 newtype
                                             new datatype (handle)
21
22
     C binding
23
     int MPI_Type_create_darray(int size, int rank, int ndims,
24
                     const int array_of_gsizes[], const int array_of_distribs[],
25
                     const int array_of_dargs[], const int array_of_psizes[],
26
                     int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
27
     int MPI_Type_create_darray_c(int size, int rank, int ndims,
28
                     const MPI_Count array_of_gsizes[],
29
30
                     const int array_of_distribs[], const int array_of_dargs[],
^{31}
                     const int array_of_psizes[], int order, MPI_Datatype oldtype,
32
                    MPI_Datatype *newtype)
33
     Fortran 2008 binding
34
     MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
35
                     array_of_distribs, array_of_dargs, array_of_psizes, order,
36
                    oldtype, newtype, ierror)
37
          INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
38
                     array_of_distribs(ndims), array_of_dargs(ndims),
39
                     array_of_psizes(ndims), order
40
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
41
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
42
          INTEGER, OPTIONAL, INTENT(OUT) :: 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, INTENT(IN) :: size, rank, ndims, array_of_distribs(ndims),
48
                     array_of_dargs(ndims), array_of_psizes(ndims), order
```

```
1
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_gsizes(ndims)
                                                                                            2
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                            3
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                            4
                                                                                            5
Fortran binding
                                                                                            6
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
               ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
               OLDTYPE, NEWTYPE, IERROR)
                                                                                            9
    INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
                                                                                           10
                ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE,
                                                                                           11
                NEWTYPE, IERROR
                                                                                           12
                                                                                           13
    MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding
                                                                                           14
to the distribution of an ndims-dimensional array of oldtype elements onto an
                                                                                           15
ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be
                                                                                           16
set to 1. (See Example 5.7.) 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
                                                                                           17
in the process grid is assumed to be row-major, as in the case of virtual Cartesian process
                                                                                           18
                                                                                           19
topologies.
                                                                                           20
                        For both Fortran and C arrays, the ordering of processes in the
     Advice to users.
                                                                                           21
     process grid is assumed to be row-major. This is consistent with the ordering used in
                                                                                           22
     virtual Cartesian process topologies in MPI. To create such virtual process topologies,
                                                                                           23
     or to find the coordinates of a process in the process grid, etc., users may use the
                                                                                           ^{24}
     corresponding process topology functions, see Chapter 8. (End of advice to users.)
                                                                                           25
                                                                                           26
    Each dimension of the array can be distributed in one of three ways:
                                                                                           27
   • MPI_DISTRIBUTE_BLOCK - Block distribution
                                                                                           28
                                                                                           29
   • MPI_DISTRIBUTE_CYCLIC - Cyclic distribution
                                                                                           30
                                                                                           31
   • MPI_DISTRIBUTE_NONE - Dimension not distributed.
                                                                                           32
                                                                                           33
    The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument.
                                                                                           34
The distribution argument for a dimension that is not distributed is ignored. For any
dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify
                                                                                           35
                                                                                           36
array_of_dargs[i] * array_of_psizes[i] < array_of_gsizes[i].
                                                                                           37
    For example, the HPF layout ARRAY(CYCLIC(15)) corresponds to
                                                                                           38
MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout AR-
                                                                                           39
RAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of
MPI_DISTRIBUTE_DFLT_DARG.
                                                                                           40
                                                                                           41
    The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the stor-
                                                                                           42
age order. Therefore, arrays described by this type constructor may be stored in Fortran
(column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN
                                                                                           43
                                                                                           44
and MPI_ORDER_C.
                                                                                           45
    This routine creates a new MPI datatype with a typemap defined in terms of a function
called "cyclic()" (see below).
                                                                                           46
                                                                                           47
                                                                                           48
```

```
1
          Without loss of generality, it suffices to define the typemap for the
\mathbf{2}
     MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.
3
          MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the
4
     MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.
          MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG
5
6
     is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to
7
           (array_of_gsizes[i] + array_of_psizes[i] - 1)/array_of_psizes[i].
8
9
     If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and
10
     MPI_DISTRIBUTE_CYCLIC are equivalent.
11
          MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with
12
     array_of_dargs[i] set to array_of_gsizes[i].
13
          Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to
14
     MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with
15
     array_of_dargs[i] set to 1.
16
          For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined
17
     by the following code fragment:
18
19
          oldtypes[0] = oldtype;
20
          for (i = 0; i < ndims; i++) {</pre>
21
               oldtypes[i+1] = cyclic(array_of_dargs[i],
22
                                         array_of_gsizes[i],
23
                                         r[i],
24
                                         array_of_psizes[i],
25
                                         oldtypes[i]);
26
          }
27
          newtype = oldtypes[ndims];
28
29
          For MPI_ORDER_C, the code is:
30
31
          oldtypes[0] = oldtype;
32
          for (i = 0; i < ndims; i++) {</pre>
               oldtypes[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
33
34
                                            array_of_gsizes[ndims - i - 1],
35
                                            r[ndims - i - 1],
36
                                            array_of_psizes[ndims - i - 1],
37
                                            oldtypes[i]);
38
          }
39
          newtype = oldtypes[ndims];
40
41
     where r[i] is the position of the process (with rank rank) in the process grid at dimension
42
     i. The values of r[i] are given by the following code fragment:
43
44
          t_rank = rank;
45
          t_size = 1;
46
          for (i = 0; i < ndims; i++)</pre>
47
              t_size *= array_of_psizes[i];
48
```

```
for (i = 0; i < ndims; i++) {
                                                                                                                           2
            t_size = t_size / array_of_psizes[i];
            r[i] = t_rank / t_size;
            t_rank = t_rank % t_size;
      }
Let the typemap of oldtype have the form:
       \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}
                                                                                                                           9
where type_i is a predefined MPI datatype, and let ex be the extent of
                                                                                                                          10
oldtype. The following function uses the conceptual datatypes lb_marker and ub_marker, see
                                                                                                                          11
Section 5.1.6 for details.
                                                                                                                          12
      Given the above, the function cyclic() is defined as follows:
                                                                                                                          13
       cyclic(darg, gsize, r, psize, oldtype)
                                                                                                                          14
         = {(lb_marker, 0),
                                                                                                                          15
               (type_0, disp_0 + r \times darq \times ex), \ldots,
                                                                                                                          16
                                                                                                                          17
                        (type_{n-1}, disp_{n-1} + r \times darg \times ex),
                                                                                                                          18
               (type_0, disp_0 + (r \times darq + 1) \times ex), \ldots,
                                                                                                                          19
                        (type_{n-1}, disp_{n-1} + (r \times darq + 1) \times ex).
                                                                                                                          20
               . . .
                                                                                                                          21
               (type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots,
                                                                                                                          22
                        (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex),
                                                                                                                          23
                                                                                                                          ^{24}
                                                                                                                          25
               (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots,
                                                                                                                          26
                        (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex),
                                                                                                                          27
                                                                                                                          28
               (type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \dots,
                                                                                                                          29
                        (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),
                                                                                                                          30
                                                                                                                          31
               (type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \dots,
                                                                                                                          32
                        (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),
                                                                                                                          33
                                                                                                                          34
                                                                                                                          35
               (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots,
                                                                                                                          36
                        (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),
                                                                                                                          37
                                                                                                                          38
               (type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \dots,
                                                                                                                          39
                     (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex
                                                                                                                          40
                                  +psize \times darg \times ex \times (count - 1)),
                                                                                                                          41
                                                                                                                          42
               (type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex
                                                                                                                          43
                                                                                                                          44
                                  +psize \times darg \times ex \times (count - 1)), \ldots,
                                                                                                                          45
                        (type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex
                                                                                                                          46
                                  +psize \times darg \times ex \times (count - 1)),
                                                                                                                          47
                                                                                                                          48
               (ub_marker, gsize * ex)
```

```
1
     where count is defined by this code fragment:
2
          nblocks = (gsize + (darg - 1)) / darg;
3
          count = nblocks / psize;
4
          left_over = nblocks - count * psize;
5
          if (r < left_over)</pre>
6
              count = count + 1;
7
8
     Here, nblocks is the number of blocks that must be distributed among the processors.
9
     Finally, darg_{last} is defined by this code fragment:
10
11
          if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
12
              darg_last = darg;
13
          else {
14
              darg_last = num_in_last_cyclic - darg * r;
15
              if (darg_last > darg)
16
                   darg_last = darg;
17
              if (darg_last <= 0)</pre>
18
                   darg_last = darg;
19
              }
20
21
     Example 5.7 Consider generating the filetypes corresponding to the HPF distribution:
22
23
            <oldtype> FILEARRAY(100, 200, 300)
24
     !HPF$ PROCESSORS PROCESSES(2, 3)
25
     !HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
26
     This can be achieved by the following Fortran code, assuming there will be six processes
27
     attached to the run:
28
29
     ndims = 3
30
     array_of_gsizes(1) = 100
^{31}
     array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
32
     \operatorname{array_of}_{\operatorname{dargs}}(1) = 10
33
     array_of_gsizes(2) = 200
34
     array_of_distribs(2) = MPI_DISTRIBUTE_NONE
35
     \operatorname{array_of_dargs}(2) = 0
36
     array_of_gsizes(3) = 300
37
     array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
38
     array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
39
     array_of_psizes(1) = 2
40
     array_of_psizes(2) = 1
41
     array_of_psizes(3) = 3
42
     call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
43
     call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
44
     call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
45
           array_of_distribs, array_of_dargs, array_of_psizes,
                                                                              &
46
           MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
47
48
```

### 5.1.5 Address and Size Functions

The displacements in a general datatype are relative to some initial buffer address. Absolute addresses can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI\_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI\_BOTTOM. 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 5.1.12 on pages 20 and 152.

Rationale. Address sized integer values, i.e., MPL\_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.)

### MPI\_GET\_ADDRESS(location, address)

	, , ,					
IN	location	location in caller memory (choice)				
OUT	address	address of location (integer)				
C bindi int MPI	3	ocation, MPI_Aint *address)				
Fortran 2008 binding						
MPI Get	_address(location, address	. ierror)				
	TYPE(*), DIMENSION(), ASYNCHRONOUS :: location					
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address						
TNT	EGER, OPTIONAL, INTENT(OUT	) :: lerror				
Fortran	binding					
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)						
<ty< td=""><td>pe&gt; LOCATION(*)</td><td></td></ty<>	pe> LOCATION(*)					
INT	- EGER(KIND=MPI_ADDRESS_KIND	) ADDRESS				
TNT	EGER IERROR					
Ret	urns the (byte) address of $locat$	ion.				

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

### Unofficial Draft for Comment Only

 $\frac{44}{45}$ 

```
1
     Example 5.8 Using MPI_GET_ADDRESS for an array.
\mathbf{2}
3
     REAL A(100,100)
     INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
4
     CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
5
     CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
6
     DIFF = MPI_AINT_DIFF(I2, I1)
7
8
     ! The value of DIFF is 909*SIZEOF(REAL); the values of I1 and I2 are
     ! implementation dependent.
9
10
           Advice to users.
                              C users may be tempted to avoid the usage of
11
           MPI_GET_ADDRESS and rely on the availability of the address operator &. Note.
12
           however, that & cast-expression is a pointer, not an address. ISO C does not require
13
           that the value of a pointer (or the pointer cast to int) be the absolute address of
14
           the object pointed at—although this is commonly the case. Furthermore, referencing
15
           may not have a unique definition on machines with a segmented address space. The
16
           use of MPI_GET_ADDRESS to "reference" C variables guarantees portability to such
17
           machines as well. (End of advice to users.)
18
19
                               To prevent problems with the argument copying and register
           Advice to users.
20
           optimization done by Fortran compilers, please note the hints in Sections 19.1.10-
21
           19.1.20. (End of advice to users.)
22
23
          To ensure portability, arithmetic on MPI addresses must be performed using the
^{24}
     MPI_AINT_ADD and MPI_AINT_DIFF functions.
25
26
27
     MPI_AINT_ADD(base, disp)
28
       IN
                 base
                                              base address (integer)
29
       IN
30
                 disp
                                              displacement (integer)
^{31}
32
     C binding
33
     MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
34
     Fortran 2008 binding
35
     INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)
36
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp
37
38
     Fortran binding
39
     INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP)
40
          INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP
41
          MPI_AINT_ADD produces a new MPI_Aint value that is equivalent to the sum of
42
     the base and disp arguments, where base represents a base address returned by a call to
43
     MPI_GET_ADDRESS and disp represents a signed integer displacement. The resulting ad-
44
     dress is valid only at the process that generated base, and it must correspond to a location
45
     in the same object referenced by base, as described in Section 5.1.12. The addition is per-
46
     formed in a manner that results in the correct MPI_Aint representation of the output address,
47
     as if the process that originally produced base had called:
48
```

MPI\_Get\_address((char \*) base + disp, &result);

```
MPI_AINT_DIFF(addr1, addr2)
```

IN	addr1	minuend address (integer)
IN	addr2	subtrahend address (integer)

### C binding

MPI\_Aint MPI\_Aint\_diff(MPI\_Aint addr1, MPI\_Aint addr2)

### Fortran 2008 binding

```
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2
```

### Fortran binding

```
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)
INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2
```

MPI\_AINT\_DIFF produces a new MPI\_Aint value that is equivalent to the difference between addr1 and addr2 arguments, where addr1 and addr2 represent addresses returned by calls to MPI\_GET\_ADDRESS. The resulting address is valid only at the process that generated addr1 and addr2, and addr1 and addr2 must correspond to locations in the same object in the same process, as described in Section 5.1.12. The difference is calculated in a manner that results in the signed difference from addr1 to addr2, as if the process that originally produced the addresses had called (char \*) addr1 - (char \*) addr2 on the addresses initially passed to MPI\_GET\_ADDRESS.

The following auxiliary functions provide useful information on derived datatypes.

MPI\_TYPE\_SIZE(datatype, size)

IN	datatype	datatype (handle)	31
IIN	uatatype	datatype (nandle)	
OUT	size	datatype size (integer)	32
			33
C bindi			34
	U U U U U U U U U U U U U U U U U U U		35
int MP1	_Type_size(MPI_Datatype da	tatype, int *size)	36
int MPI_	Type_size_c(MPI_Datatype	datatype, MPI_Count *size)	37
-			38
	2008 binding		39
MPI_Type	e_size(datatype, size, ier:	ror)	40
TYPE	E(MPI_Datatype), INTENT(IN	) :: datatype	41
INTE	EGER, INTENT(OUT) :: size		42
INTE	EGER, OPTIONAL, INTENT(OUT)	) :: ierror	
	,,,,	,	43
MPI_Type	e_size(datatype, size, ier:	ror)	44
TYPE	E(MPI_Datatype), INTENT(IN	) :: datatype	45
INTE	EGER(KIND=MPI_COUNT_KIND),	INTENT(OUT) :: size	46
INTE	EGER, OPTIONAL, INTENT(OUT	) :: ierror	47
			48

1

2 3 4

5 6

9

10

11

12

13

14 15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27 28 29

```
1
     Fortran binding
\mathbf{2}
     MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
3
          INTEGER DATATYPE, SIZE, IERROR
4
5
6
     MPI_TYPE_SIZE_X(datatype, size)
7
                datatype
       IN
                                            datatype (handle)
8
       OUT
9
                size
                                            datatype size (integer)
10
11
     C binding
12
     int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)
13
     Fortran 2008 binding
14
     MPI_Type_size_x(datatype, size, ierror)
15
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
17
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     Fortran binding
20
     MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
21
         INTEGER DATATYPE, IERROR
22
         INTEGER(KIND=MPI_COUNT_KIND) SIZE
23
         MPI_TYPE_SIZE and MPI_TYPE_SIZE_X set the value of size to the total size, in
24
```

<sup>24</sup> MPI\_TYPE\_SIZE and MPI\_TYPE\_SIZE\_X set the value of size to the total size, in <sup>25</sup> bytes, of the entries in the type signature associated with datatype; i.e., the total size of the <sup>26</sup> data in a message that would be created with this datatype. Entries that occur multiple <sup>27</sup> times in the datatype are counted with their multiplicity. For both functions, if the OUT <sup>28</sup> parameter cannot express the value to be returned (e.g., if the parameter is too small to <sup>29</sup> hold the output value), it is set to MPI\_UNDEFINED.

 $30 \\ 31$ 

## 5.1.6 Lower-Bound and Upper-Bound Markers

32 It is often convenient to define explicitly the lower bound and upper bound of a type map, 33 and override the definition given on page 141. This allows one to define a datatype that has 34 "holes" at its beginning or its end, or a datatype with entries that extend above the upper 35 bound or below the lower bound. Examples of such usage are provided in Section 5.1.14. 36 Also, the user may want to overide the alignment rules that are used to compute upper 37 bounds and extents. E.g., a C compiler may allow the user to overide default alignment 38 rules for some of the structures within a program. The user has to specify explicitly the 39 bounds of the datatypes that match these structures. 40

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.

<sup>47</sup> Example 5.9 A call to MPI\_TYPE\_CREATE\_RESIZED(MPI\_INT, -3, 9, type1) creates a

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

In general, if

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

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

	$\min_{j} disp_{j}$	if no entry has type
$lb(Typemap) = \langle$		lb_marker
	$\min_{j} \{ disp_j \text{ such that } type_j = lb\_marker \}$	otherwise

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

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

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

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

Advice to users. Structures combining different basic datatypes should be defined 44 so that there will be no gaps based on alignment rules. If such a datatype is used 45 to create an array of structures, users should also avoid an alignment-gap at the 46 end of the structure. In MPI communication, the content of such gaps would not 47 be communicated into the receiver's buffer. For example, such an alignment-gap 48

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9 10 11

12

25

26

27

28

29

30

 $^{31}$ 

32 33

34

35

36

37

38 39

40

41

42

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 END TYPE

one should define

TYPE, BIND(C) :: my\_data REAL, DIMENSION(3) :: x REAL :: gap1 DOUBLE PRECISION :: p END TYPE

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

In principle, a compiler may define an additional alignment rule for structures, e.g., to use at least 4 or 8 byte alignment, although the content may have a  $max_ik_i$  alignment less than this structure alignment. To maintain portability, users should always resize structure derived datatype handles if used in an array of structures, see the Example in Section 19.1.15. (End of advice to users.)

36 37

5.1.7Extent and Bounds of Datatypes

MPI\_TYPE\_GET\_EXTENT(datatype, lb, extent)

IN	datatype	datatype to get information on (handle)
OUT	lb	lower bound of datatype (integer)
OUT	extent	extent of datatype (integer)
C bindin int MPI_	0	type datatype, MPI_Aint *1b,

1

2

3

4

5

6

7 8

9

10

11

12

13

14

1516

17 18 19

20

2122

23

2425

26

27

28

29

30

31

32

33

34

35

38 39

4041

...

# 5.1. DERIVED DATATYPES

int MPI_7	<pre>'ype_get_extent_c(MPI_Data MPI_Count *extent)</pre>	atype datatype, MPI_Count *1b,	12
MPI_Type_ TYPE( INTEC	2008 binding get_extent(datatype, lb, [MPI_Datatype), INTENT(IN) ER(KIND=MPI_ADDRESS_KIND) ER, OPTIONAL, INTENT(OUT)	:: datatype , INTENT(OUT) :: lb, extent	3 4 5 6 7
MPI_Type_ TYPE( INTEC	get_extent(datatype, lb, MPI_Datatype), INTENT(IN)	extent, ierror) :: datatype INTENT(OUT) :: lb, extent	8 9 10 11 12
INTEC	Dinding _GET_EXTENT(DATATYPE, LB, ER DATATYPE, IERROR ER(KIND=MPI_ADDRESS_KIND)		13 14 15 16 17 18
	E_GET_EXTENT_X(datatype,	lb_extent)	19 20
IN	datatype	datatype to get information on (handle)	20
			22
OUT	lb	lower bound of datatype (integer)	23
OUT	extent	extent of datatype (integer)	24
<u></u>			25
C binding	5		26 27
int MPI_I	<pre>ype_get_extent_x(MP1_Data MP1_Count *extent)</pre>	atype datatype, MPI_Count *1b,	28
	MF1_Count *extent)		29
	2008 binding		30
• -	get_extent_x(datatype, 1		31
	(MPI_Datatype), INTENT(IN)	• -	32
	ER(KIND=MP1_COUNT_KIND), ER, OPTIONAL, INTENT(OUT)	INTENT(OUT) :: 1b, extent	33
INIEC	ER, UFIIONAL, INIENI(UUI)		34
Fortran k	pinding		35
	_GET_EXTENT_X(DATATYPE, LE	3, EXTENT, IERROR)	36 27
	ER DATATYPE, IERROR		37 38
INTEG	ER(KIND=MPI_COUNT_KIND) I	B, EXTENT	39
Retur	ns the lower bound and the ex	stent of datatype (as defined in Equation $5.1$ ).	40
	· · · · · · · · · · · · · · · · · · ·	arameter cannot express the value to be returned	41
· - ·	-	ld the output value), it is set to MPI_UNDEFINED.	42
		of a datatype, using lower bound and upper bound	43
		stride of successive datatypes that are replicated	44
by datatyp	be constructors, or are replicat	ed by the <b>count</b> argument in a send or receive call.	45

 $46 \\ 47$ 

	144		CH	IAPTER 5. DATATYPES
<sup>1</sup> #-other	MPI_TYP IN	E_CREATE_RESIZED(oldtype, oldtype	lb, extent, newtype) input datatype (handle)	"input" datatype is correct. After the release of this RC 40 document, somebody changed it to "old" datatype, which is incorrect and must be reverted.
#-error!	IN	lb	new lower bound of data	atype (integer)
Done 5	IN	extent	new extent of datatype	(integer)
6	OUT	newtype	output datatype (handle	9)
7		ithub.com/mpi-forum/mpi-standard/commit/06d0e8b00 ap-datatypes/datatypes.tex, line 1800	0679dae3af201a1594f67bfb1db1899e#diff-	23137ee8f54746beeeb2ec2383c079a5bf30723d60fc080fc861ff8b8dbe70d2
9	C bindin	g		
10	int MPI_	Type_create_resized(MPI_Dates in the set of	atatype oldtype, MPI	_Aint lb,
11		MPI_Aint extent, MPI	_Datatype *newtype)	
12	int MPI_	Type_create_resized_c(MPI	_Datatype oldtype, M	PI Count 1b,
13	-	MPI_Count extent, MP		
14				
15		2008 binding		
16		_create_resized(oldtype, )		lerror)
17		(MPI_Datatype), INTENT(IN	• •	
18		GER(KIND=MPI_ADDRESS_KIND) (MPI_Datatype), INTENT(OU)		extent
19		GER, OPTIONAL, INTENT(OUT)		
20	111111	dent, of flower, infent(001)	) ielioi	
21	• -	_create_resized(oldtype,		ierror)
22 23		(MPI_Datatype), INTENT(IN		
23		GER(KIND=MPI_COUNT_KIND),		xtent
25		(MPI_Datatype), INTENT(OU		
26	INTE(	GER, OPTIONAL, INTENT(OUT	) :: ierror	
27	Fortran l	binding		
28	MPI_TYPE	_CREATE_RESIZED(OLDTYPE, I	LB, EXTENT, NEWTYPE,	IERROR)
29	INTE	GER OLDTYPE, NEWTYPE, IER	ROR	
30	INTE	GER(KIND=MPI_ADDRESS_KIND	) LB, EXTENT	
31	Retur	rns in <b>newtype</b> a handle to a ne	ew datatype that is ident	ical to oldtype except that
32		bound of this new datatype i		
33		Any previous <b>lb</b> and <b>ub</b> mark	·	
34		J F T T T T T T T T T T T T T T T T T T	and a me	r

<sup>33</sup> + extent. Any previous **lb** and **ub** markers are erased, and a new pair of lower bound and <sup>34</sup> upper bound markers are put in the positions indicated by the **lb** and extent arguments. <sup>35</sup> This affects the behavior of the datatype when used in communication operations, with <sup>36</sup> count > 1, and when used in the construction of new derived datatypes.

37 38

39

# 5.1.8 True Extent of Datatypes

40Suppose we implement gather (see also Section 6.5) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root pro-4142cess, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount 43of space that needs to be allocated, if the user has modified the extent, for example 44by using MPI\_TYPE\_CREATE\_RESIZED. The functions MPI\_TYPE\_GET\_TRUE\_EXTENT 45and MPI\_TYPE\_GET\_TRUE\_EXTENT\_X are provided which return the true extent of the 4647datatype.

MPI_TYPE	E_GET_TRUE_EXTENT(datat	ype, true_lb, true_extent)	1
IN	datatype	datatype to get information on (handle)	2 3
OUT	true_lb	true lower bound of datatype (integer)	4
OUT	true_extent	true size of datatype (integer)	5
			6
C binding			7 8
int MPI_1		Datatype datatype, MPI_Aint *true_lb,	9
	MPI_Aint *true_exten		10
int MPI_1	<pre>Yype_get_true_extent_c(MP] MPI_Count *true_exte</pre>	I_Datatype datatype, MPI_Count *true_lb,	11
_			12 13
	2008 binding	true lb true extent ierror)	14
• -	(MPI_Datatype), INTENT(IN)	, true_lb, true_extent, ierror) ) :: datatype	15
	• -	), INTENT(OUT) :: true_lb, true_extent	16
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	17 18
MPI_Type_	_get_true_extent(datatype;	, true_lb, true_extent, ierror)	19
	MPI_Datatype), INTENT(IN)		20
		INTENT(OUT) :: true_lb, true_extent	21
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: lerror	22 23
Fortran b	6		23
	GET_TRUE_EXTENT(DATATYPE; ER DATATYPE, IERROR	, TRUE_LB, TRUE_EXTENT, IERROR)	25
	ER(KIND=MPI_ADDRESS_KIND)	) TRUE I.B. TRUE EXTENT	26
	·····	,	27
			28 29
MPI_TYPE	E_GET_TRUE_EXTENT_X(da	tatype, true_lb, true_extent)	30
IN	datatype	datatype to get information on (handle)	31
OUT	true_lb	true lower bound of datatype (integer)	32 33
OUT	true_extent	true size of datatype (integer)	34
			35
C binding			36
int MPI_1		I_Datatype datatype, MPI_Count *true_lb,	37
	MPI_Count *true_exte	nt)	38 39
	2008 binding		40
• -		pe, true_lb, true_extent, ierror)	41
	MPI_Datatype), INTENT(IN) ER(KIND=MPI_COUNT_KIND)	) :: datatype INTENT(OUT) :: true_lb, true_extent	42
	ER, OPTIONAL, INTENT(OUT)	-	43
			44
Fortran k	6	PE, TRUE_LB, TRUE_EXTENT, IERROR)	45 46
	ER DATATYPE, IERROR	, 1100_00, 1100_0A1001, 1000000/	40
	ER(KIND=MPI_COUNT_KIND)	TRUE_LB, TRUE_EXTENT	48

true\_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring explicit lower bound markers. true\_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring explicit lower bound and upper bound markers, and performing no rounding for alignment. If the typemap associated with datatype is  $Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$ Then  $true_{lb}(Typemap) = min_{i} \{ disp_{i} : type_{i} \neq \mathsf{lb}_{marker}, \mathsf{ub}_{marker} \},$  $true\_ub(Typemap) = max_i \{ disp_i + sizeof(type_i) : type_i \neq lb\_marker, ub\_marker \},\$ and  $true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).$ (Readers should compare this with the definitions in Section 5.1.6 and Section 5.1.7, which describe the function MPI\_TYPE\_GET\_EXTENT.) The true\_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed. For both functions, if either OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI\_UNDEFINED. 5.1.9 Commit and Free A datatype object has to be **committed** before it can be used in a communication. As an argument in datatype constructors, uncommitted and also committed datatypes can be used. There is no need to commit basic datatypes. They are "pre-committed." MPI\_TYPE\_COMMIT(datatype) INOUT datatype datatype that is committed (handle) C binding int MPI\_Type\_commit(MPI\_Datatype \*datatype) Fortran 2008 binding MPI\_Type\_commit(datatype, ierror) TYPE(MPI\_Datatype), INTENT(INOUT) :: datatype INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI\_TYPE\_COMMIT(DATATYPE, IERROR) INTEGER DATATYPE, IERROR The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed,

the content of different buffers, with different starting addresses.

1

 $\mathbf{2}$ 

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

 $\mathbf{2}$ 

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

The system may "compile" at commit time an internal Advice to implementors. representation for the datatype that facilitates communication, e.g., change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (End of advice to implementors.) MPI\_TYPE\_COMMIT will accept a committed datatype; in this case, it is equivalent to a no-op. **Example 5.10** The following code fragment gives examples of using MPI\_TYPE\_COMMIT. INTEGER type1, type2 CALL MPI\_TYPE\_CONTIGUOUS(5, MPI\_REAL, type1, ierr) ! new type object created CALL MPI\_TYPE\_COMMIT(type1, ierr) ! now type1 can be used for communication type2 = type1! type2 can be used for communication ! (it is a handle to same object as type1) CALL MPI\_TYPE\_VECTOR(3, 5, 4, MPI\_REAL, type1, ierr) ! new uncommitted type object created CALL MPI\_TYPE\_COMMIT(type1, ierr) ! now type1 can be used anew for communication MPI\_TYPE\_FREE(datatype) datatype that is freed (handle) INOUT datatype C binding int MPI\_Type\_free(MPI\_Datatype \*datatype) Fortran 2008 binding MPI\_Type\_free(datatype, ierror) TYPE(MPI\_Datatype), INTENT(INOUT) :: datatype INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding 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

 $43 \\ 44$ 

Advice to implementors. The implementation may keep a reference count of active 45 communications that use the datatype, in order to decide when to free it. Also, one 46 may implement constructors of derived datatypes so that they keep pointers to their 47 datatype arguments, rather then copying them. In this case, one needs to keep track 48

# Unofficial Draft for Comment Only

from the freed datatype. The system behaves as if input datatype arguments to derived

datatype constructors are passed by value.

1 of active datatype definition references in order to know when a datatype object can  $\mathbf{2}$ be freed. (End of advice to implementors.) 3 4 5.1.10 Duplicating a Datatype 56  $\overline{7}$ MPI\_TYPE\_DUP(oldtype, newtype) 8 IN oldtype datatype (handle) 9 10 OUT newtype copy of oldtype (handle) 11 12C binding 13 int MPI\_Type\_dup(MPI\_Datatype oldtype, MPI\_Datatype \*newtype) 14Fortran 2008 binding 15MPI\_Type\_dup(oldtype, newtype, ierror) 16TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 17TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 20Fortran binding 21MPI\_TYPE\_DUP(OLDTYPE, NEWTYPE, IERROR) 22 INTEGER OLDTYPE, NEWTYPE, IERROR 23MPI\_TYPE\_DUP is a type constructor which duplicates the existing oldtype with as-24sociated key values. For each key value, the respective copy callback function determines 25the attribute value associated with this key in the new communicator; one particular action 26that a copy callback may take is to delete the attribute from the new datatype. Returns 27in newtype a new datatype with exactly the same properties as oldtype and any copied 28cached information, see Section 7.7.4. The new datatype has identical upper bound and 29 lower bound and yields the same net result when fully decoded with the functions in Sec-30 tion 5.1.13. The newtype has the same committed state as the old oldtype.  $^{31}$ 32 5.1.11 Use of General Datatypes in Communication 33

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

```
<sup>39</sup> MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
```

```
40 MPI_TYPE_COMMIT(newtype)
```

<sup>41</sup> MPI\_SEND(buf, 1, newtype, dest, tag, comm)

<sup>42</sup> MPI\_TYPE\_FREE(newtype).

48

- 43
   44
   45
   46
   47
   48
   49
   49
   40
   41
   41
   41
   41
   42
   42
   43
   44
   45
   44
   45
   44
   45
   45
   46
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   47
   4
- $_{46}$  Suppose that a send operation MPI\_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

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

 $\mathbf{2}$ 

 $^{24}$ 

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

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

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

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

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

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

**Example 5.11** This example shows that type matching is defined in terms of the basic types that a derived type consists of.

	26
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2,)	27
CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4,)	28
CALL MPI_TYPE_CONTIGUOUS(2, type2, type22,)	29
	30
CALL MPI_SEND(a, 4, MPI_REAL,)	31
CALL MPI_SEND(a, 2, type2,)	32
CALL MPI_SEND(a, 1, type22,)	33
CALL MPI_SEND(a, 1, type4,)	34
	35
CALL MPI_RECV(a, 4, MPI_REAL,)	36
CALL MPI_RECV(a, 2, type2,)	37
CALL MPI_RECV(a, 1, type22,)	38
CALL MPI_RECV(a, 1, type4,)	39

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,

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

```
1
     The received message need not fill all the receive buffer, nor does it need to fill a number of
\mathbf{2}
     locations which is a multiple of n. Any number, k, of basic elements can be received, where
3
     0 \le k \le \text{count} \cdot n. The number of basic elements received can be retrieved from status using
4
     the query functions MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X.
5
6
     MPI_GET_ELEMENTS(status, datatype, count)
7
8
       IN
                                             return status of receive operation (Status)
                 status
9
       IN
                                             datatype used by receive operation (handle)
                 datatype
10
                                             number of received basic elements (integer)
       OUT
                 count
11
12
13
     C binding
     int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,
14
15
                    int *count)
16
     int MPI_Get_elements_c(const MPI_Status *status, MPI_Datatype datatype,
17
                    MPI_Count *count)
18
19
     Fortran 2008 binding
     MPI_Get_elements(status, datatype, count, ierror)
20
21
          TYPE(MPI_Status), INTENT(IN) :: status
22
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
          INTEGER, INTENT(OUT) :: count
24
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Get_elements(status, datatype, count, ierror)
26
          TYPE(MPI_Status), INTENT(IN) :: status
27
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
29
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     Fortran binding
32
     MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
33
          INTEGER STATUS (MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
34
35
36
     MPI_GET_ELEMENTS_X(status, datatype, count)
37
       IN
                 status
                                             return status of receive operation (Status)
38
39
       IN
                 datatype
                                             datatype used by receive operation (handle)
40
       OUT
                 count
                                             number of received basic elements (integer)
41
42
     C binding
43
     int MPI_Get_elements_x(const MPI_Status *status, MPI_Datatype datatype,
44
                    MPI_Count *count)
45
46
     Fortran 2008 binding
47
     MPI_Get_elements_x(status, datatype, count, ierror)
48
          TYPE(MPI_Status), INTENT(IN) :: status
```

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### Fortran binding

```
MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
INTEGER(KIND=MPI_COUNT_KIND) COUNT
```

The datatype argument should match the argument provided by the receive call that set the status variable. For both functions, if the OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI\_UNDEFINED.

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

```
Example 5.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.
```

```
. . .
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
   CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
   CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
   CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                 ! returns i=1
  CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=2
   CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
   CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                 ! returns i=MPI_UNDEFINED
   CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=3
END IF
```

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

Rationale. The extension given to the definition of MPI\_GET\_COUNT seems natural: <sup>46</sup> one would expect this function to return the value of the count argument, when the <sup>47</sup> receive buffer is filled. Sometimes datatype represents a basic unit of data one wants <sup>48</sup>

# Unofficial Draft for Comment Only

1

2

5

6

7

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22 23

 $^{24}$ 

25

26

27 28 29

30

31

32

33

34

35

36

37

38

39 40

41

42

43

44

to transfer, for example, a record in an array of records (structures). One should be able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI\_GET\_ELEMENTS or MPI\_GET\_ELEMENTS\_X. (End of rationale.)

Advice to implementors. The definition implies that a receive cannot change the value of storage outside the entries defined to compose the communication buffer. In particular, the definition implies that padding space in a structure should not be modified when such a structure is copied from one process to another. This would prevent the obvious optimization of copying the structure, together with the padding, as one contiguous block. The implementation is free to do this optimization when it does not impact the outcome of the computation. The user can "force" this optimization by explicitly including padding as part of the message. (End of advice to implementors.)

# 5.1.12 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,  $^{24}$ to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are 25defined recursively as follows: 26

- 1. The function MPL 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 Aexecuted on addresses.

39 The rules above impose no constraints on the use of derived datatypes, as long as 40 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 42variables that are not within the same sequential storage must obey certain restrictions. 43 Basically, a communication buffer with variables that are not within the same sequential 44storage can be used only by specifying in the communication call  $buf = MPI_BOTTOM$ , count 45= 1, and using a datatype argument where all displacements are valid (absolute) addresses. 46

47It is not expected that MPI implementations will be able to de-Advice to users. 48 tect erroneous, "out of bound" displacements—unless those overflow the user address

# **Unofficial Draft for Comment Only**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

151617

18

19

20

21

22

23

27

28

29

30

3132

33

34 35

36

37

38

space—since the MPI call may not know the extent of the arrays and records in the host program. (End of advice to users.)

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

#### Decoding a Datatype 5.1.13

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

	num_datatypes, combine	r)	22
IN	datatype	datatype to access (handle)	23 24
OUT	num_integers	number of input integers used in call constructing	25
001	num_integers	combiner (non-negative integer)	26
OUT	num_addresses		27
001	num_addresses	number of input addresses used in call constructing combiner (non-negative integer)	28
			29
OUT	num_large_counts	number of input large counts used in call	30
		constructing combiner (non-negative integer, only	31
		$\mathbf{present} \ \mathbf{for} \ \mathbf{large} \ \mathbf{count} \ \mathbf{variants})$	32
OUT	num_datatypes	number of input datatypes used in call constructing	33
		combiner (non-negative integer)	34
OUT	combiner	combiner (state)	35
001	combine.		36
C bindin	a di seconda		37
	-	atype datatype, int *num_integers,	38
IIIC FIFI		int *num_datatypes, int *combiner)	39
	the *num_addresses,	int *num_datatypes, int *combiner)	40
int MPI_7	Type_get_envelope_c(MPI_Da	atatype datatype, MPI_Count *num_integers,	41
	MPI_Count *num_addre	<pre>sses, MPI_Count *num_large_counts,</pre>	42
	MPI_Count *num_datat	ypes, int *combiner)	43
Fortran (	2008 binding		44
	0	um_integers, num_addresses, num_datatypes,	45
	combiner, ierror)	m_integers, num_addresses, num_datatypes,	46
ТАЬЕ	(MPI_Datatype), INTENT(IN)	) :: datatype	47
111 []		· ·· aavavjpo	48

MPI\_TYPE\_GET\_ENVELOPE(datatype, num\_integers, num\_addresses, num\_large\_counts,

1

 $\mathbf{2}$ 

3

4

5

6

7

8 9

10

11

12

13

14

15

16

17

18

192021

22

1 2	<pre>INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,</pre>
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4 5 6 7	<pre>MPI_Type_get_envelope(datatype, num_integers, num_addresses,</pre>
8 9 10	<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: num_integers,</pre>
11	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12 13 14 15 16	<pre>Fortran binding MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,</pre>
17 18 19 20 21 22	For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-of-arguments values returned can be used to provide sufficiently large arrays in the decoding routine MPI_TYPE_GET_CONTENTS. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.
23 24 25 26 27 28 29 30 31	<ul> <li>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.</li> <li>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</li> </ul>
32 33	datatype. The former is a constant object that cannot be freed, while the latter is a derived datatype that can be freed. ( <i>End of rationale.</i> )
34 35 36 37 38 39 40	The list in Table 5.1 has the values that can be returned in combiner on the left and the call associated with them on the right. If combiner is MPI_COMBINER_NAMED then datatype is a named predefined datatype. If the MPI_TYPE_GET_ENVELOPE variant without num_large_counts is invoked with a datatype that requires an output value of num_large_counts > 0, then an error of class MPI_ERR_TYPE is raised.
41 42 43 44 45	Rationale. The large count variant of this MPI procedure was added in MPI-4. It contains a new num_large_counts parameter. The other variant—the variant that existed before MPI-4—was not changed in order to preserve backwards compatibility. (End of rationale.)
46 47 48	The actual arguments used in the creation call for a datatype can be obtained using MPI_TYPE_GET_CONTENTS. MPI_TYPE_GET_ENVELOPE and MPI_TYPE_GET_CONTENTS also support large

# 5.1. DERIVED DATATYPES

MPI_COMBINER_NAMED	a named predefined datatype	1
MPI_COMBINER_DUP	MPI_TYPE_DUP	2
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS	3
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR	4
MPI_COMBINER_HVECTOR	MPI_TYPE_CREATE_HVECTOR	5
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED	6
MPI_COMBINER_HINDEXED	MPI_TYPE_CREATE_HINDEXED	7
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK	8
MPI_COMBINER_HINDEXED_BLOCK	MPI_TYPE_CREATE_HINDEXED_BLOCK	9
MPI_COMBINER_STRUCT	MPI_TYPE_CREATE_STRUCT	10
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY	11
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY	12
MPI_COMBINER_F90_REAL MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_REAL MPI_TYPE_CREATE_F90_COMPLEX	13 14
MPI_COMBINER_F90_COMPLEX MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_COMPLEX MPI_TYPE_CREATE_F90_INTEGER	14
MPI_COMBINER_P90_INTEGER	MPI_TYPE_CREATE_RESIZED	16
		17
		18
Table 5.1: combiner values return	ned from MPI_TYPE_GET_ENVELOPE	19
		20
count types in separate additional MPI pro-	cedures in C (suffixed with the "_c") and interface	21
polymorphism in Fortran when using USE	mpi_f08.	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	MPI_TYPI		e, max_integers, max_addresses, max_large_counts,
2 3		max_datatypes, array_c array_of_large_counts,	of_integers, array_of_addresses, array_of_datatypes)
4 5	IN	datatype	datatype to access (handle)
6 7	IN	max_integers	number of elements in array_of_integers (non-negative integer)
8 9	IN	max_addresses	number of elements in array_of_addresses (non-negative integer)
10 11 12 13	IN	max_large_counts	number of elements in array_of_large_counts (non-negative integer, only present for large count variants)
13 14 15	IN	max_datatypes	number of elements in array_of_datatypes (non-negative integer)
16 17	OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)
18 19 20	OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)
21 22 23	OUT	array_of_large_counts	contains large count arguments used in constructing datatype (array of integers, only present for large count variants)
24 25 26	OUT	array_of_datatypes	contains datatype arguments used in constructing datatype (array of handles)
27 28 29 30 31	C bindin		
32 33 34 35 36 37 38	int MPI_7	MPI_Count max_addre	large_counts[],
<ol> <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>	MPI_Type TYPE INTEC INTEC	array_of_integers, ierror) (MPI_Datatype), INTENT(I GER, INTENT(IN) :: max_i	ntegers, max_addresses, max_datatypes y_of_integers(max_integers) D), INTENT(OUT) ::

TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2 3
MPI_Type_get_contents(datatype, max_integers, max_addresses,	3 4
<pre>max_large_counts, max_datatypes, array_of_integers,</pre>	4 5
array_of_addresses, array_of_large_counts, array_of_datatypes,	6
ierror)	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	8
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: max_integers,	9
max_addresses, max_large_counts, max_datatypes	10
<pre>INTEGER, INTENT(OUT) :: array_of_integers(max_integers)</pre>	11
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::</pre>	12
array_of_addresses(max_addresses)	13
<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::</pre>	14
array_of_large_counts(max_large_counts)	15
<pre>TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)</pre>	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
Fortran binding	18
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	19
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,	20
IERROR)	21
INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	22
ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	23
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)	24
	25
datatype must be a predefined unnamed or a derived datatype; the call is erroneous if	26
datatype is a predefined named datatype.	27
The values given for max_integers, max_addresses, max_large_counts, and	28
max_datatypes must be at least as large as the value returned in num_integers,	29
num_addresses, num_large_counts, and num_datatypes, respectively, in the call	30
MPI_TYPE_GET_ENVELOPE for the same datatype argument.	31
<i>Rationale.</i> The arguments max_integers, max_addresses, max_large_counts, and	32
max_addresses, max_arge_counts, and max_addresses, max_arge_counts, and max_datatypes allow for error checking in the call. ( <i>End of rationale.</i> )	33
max_datatypes anow for error checking in the call. (End of fationale.)	34
If the MPI_TYPE_GET_CONTENTS variant without max_large_counts is invoked with	35
a datatype that requires > 0 values in array_of_large_counts, then an error of class	36
MPI_ERR_TYPE is raised.	37
	38
Rationale. The large count variant of this MPI procedure was added in MPI-4.	39
It contains new max_large_counts and array_of_large_counts parameters. The other	40
variant—the variant that existed before MPI-4—was not changed in order to preserve	41
backwards compatibility. (End of rationale.)	42
	43
The datatypes returned in array_of_datatypes are handles to datatype objects that	44
are equivalent to the datatypes used in the original construction call. If these were derived	45
datatypes, then the returned datatypes are new datatype objects, and the user is responsible	46
	A7

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.

47

The committed state of returned derived datatypes is undefined, i.e., the datatypes may  $\mathbf{2}$ or may not be committed. Furthermore, the content of attributes of returned datatypes is 3 undefined. 4 Note that MPI\_TYPE\_GET\_CONTENTS can be invoked with a 5datatype argument that was constructed using MPI\_TYPE\_CREATE\_F90\_REAL, 6 MPI\_TYPE\_CREATE\_F90\_INTEGER, or MPI\_TYPE\_CREATE\_F90\_COMPLEX (an unnamed  $\overline{7}$ predefined datatype). In such a case, an empty array\_of\_datatypes is returned. 8 *Rationale.* The definition of datatype equivalence implies that equivalent predefined 9 datatypes are equal. By requiring the same handle for named predefined datatypes, 10 it is possible to use the == or .EQ. comparison operator to determine the datatype 11 involved. (End of rationale.) 1213 Advice to implementors. The datatypes returned in array\_of\_datatypes must appear 14to the user as if each is an equivalent copy of the datatype used in the type constructor 15call. Whether this is done by creating a new datatype or via another mechanism such 16as a reference count mechanism is up to the implementation as long as the semantics 17 are preserved. (End of advice to implementors.) 18 19 The committed state and attributes of the returned datatype is delib-Rationale. 20erately left vague. The datatype used in the original construction may have been 21modified since its use in the constructor call. Attributes can be added, removed, or 22 modified as well as having the datatype committed. The semantics given allow for 23a reference count implementation without having to track these changes. (End of 24rationale.) 2526In the deprecated datatype constructor calls, the address arguments in Fortran are 27of type INTEGER. In the preferred calls, the address arguments are of type 28 INTEGER(KIND=MPI\_ADDRESS\_KIND). The call MPI\_TYPE\_GET\_CONTENTS returns all ad-29 dresses in an argument of type INTEGER(KIND=MPI\_ADDRESS\_KIND). This is true even if the 30 deprecated calls were used. Thus, the location of values returned can be thought of as being  $^{31}$ returned by the C bindings. It can also be determined by examining the preferred calls for 32 datatype constructors for the deprecated calls that involve addresses. 33 34By having all address arguments returned in the Rationale. 35array\_of\_addresses argument, the result from a C and Fortran decoding of a datatype 36 gives the result in the same argument. It is assumed that an integer of type 37 INTEGER (KIND=MPI\_ADDRESS\_KIND) will be at least as large as the INTEGER argument 38 used in datatype construction with the old MPI-1 calls so no loss of information will 39 occur. (End of rationale.) 40 41 The following defines what values are placed in each entry of the returned arrays 42depending on the datatype constructor used for datatype. It also specifies the size of the 43 arrays needed which is the values returned by MPI\_TYPE\_GET\_ENVELOPE. In Fortran, 44the following calls were made: 4546PARAMETER (LARGE = 1000) 47INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR 48 INTEGER (KIND=MPI\_ADDRESS\_KIND) A(LARGE)

```
1
! CONSTRUCT DATATYPE TYPE (NOT SHOWN)
                                                                                         \mathbf{2}
CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)
                                                                                         3
IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
   WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, &
   " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
   CALL MPI_ABORT(MPI_COMM_WORLD, 99, IERROR)
ENDIF
CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR)
                                                                                         9
or in C the analogous calls of:
                                                                                        10
                                                                                        11
#define LARGE 1000
                                                                                        12
int ni, na, nd, combiner, i[LARGE];
                                                                                        13
MPI_Aint a[LARGE];
                                                                                        14
MPI_Datatype type, d[LARGE];
                                                                                        15
/* construct datatype type (not shown) */
                                                                                        16
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
                                                                                        17
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
                                                                                        18
    fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
                                                                                        19
    fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
                                                                                        20
             LARGE);
                                                                                        21
    MPI_Abort(MPI_COMM_WORLD, 99);
                                                                                        22
};
                                                                                        23
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
                                                                                        ^{24}
                                                                                        25
    In the descriptions that follow, the lower case name of arguments is used.
                                                                                        26
    If combiner is MPI_COMBINER_NAMED then it is erroneous to call
                                                                                        27
MPI_TYPE_GET_CONTENTS.
                                                                                        28
    If combiner is MPI_COMBINER_DUP then
                                                                                        29
                                                                                        30
                                            С
                                                 Fortran location
                    Constructor argument
                                                                                        31
                    oldtype
                                           d[0]
                                                      D(1)
                                                                                        32
                                                                                        33
and ni = 0, na = 0, nd = 1.
                                                                                        34
  If combiner is MPI_COMBINER_CONTIGUOUS then
                                                                                        35
                                                                                        36
                                            С
                                                 Fortran location
                    Constructor argument
                                                                                        37
                                           i[0]
                    count
                                                       I(1)
                                                                                        38
                    oldtype
                                           d[0]
                                                      D(1)
                                                                                        39
                                                                                        40
and ni = 1, na = 0, nd = 1.
                                                                                        41
    If combiner is MPI_COMBINER_VECTOR then
                                                                                        42
                                                                                        43
                    Constructor argument
                                            С
                                                 Fortran location
                                                                                        44
                    count
                                           i[0]
                                                       I(1)
                                                                                        45
                    blocklength
                                                      I(2)
                                           i[1]
                                                                                        46
                    stride
                                           i[2]
                                                      I(3)
                                                                                        47
                    oldtype
                                           d[0]
                                                      D(1)
                                                                                        48
```

a	nd ni = 2 no =	0  nd = 1		
	nd ni = 3, na = If combiner is	s MPI_COMBINER_HVE	CTOR then	
	II COMDINEL IS		CTOR then	
		Constructor argun	nent C Fo	rtran location
		count	i[0]	I(1)
		blocklength	i[1]	I(2)
		stride	a[0]	A(1)
		oldtype	d[0]	D(1)
a	nd ni $= 2$ , na $=$			
	If combiner is	S MPI_COMBINER_IND	EXED then	
		ctor argument	C	Fortran location
	count		i[0]	I(1)
	•		[1] to $i[i[0]]$	I(2) to $I(I(1)+1)$
		displacements i[i[0]	+1] to i[2*i[0]]	
	oldtype		d[0]	D(1)
ล	nd ni $-2^*$ count	+1, na = 0, nd = 1.		
a		1, 10 = 0, 10 = 1. S MPI_COMBINER_HINI	<b>DEXED</b> then	
	Co	onstructor argument	С	Fortran location
		unt	i[0]	I(1)
	ari	ray_of_blocklengths	i[1] to $i[i[0]]$	
		ray_of_blocklengths ray_of_displacements	$i[1] \text{ to } i[i[0]] \\ a[0] \text{ to } a[i[0]-1]$	I(2) to $I(I(1)+1)$
	ar	ray_of_blocklengths ray_of_displacements dtype	i[1] to i[i[0]] a[0] to a[i[0]-1 d[0]	I(2) to $I(I(1)+1)$
	ar	ray_of_displacements dtype	a[0] to $a[i[0]-1]$	I(2) to $I(I(1)+1)$ A(1) to A(I(1))
a	$arr old \\ old \\ nd ni = count+1$	ray_of_displacements dtype , na = count, nd = $1$ .	a[0] to a[i[0]-1 d[0]	I(2) to $I(I(1)+1)$ A(1) to A(I(1)) D(1)
a	$arr old \\ old \\ nd ni = count+1$	ray_of_displacements dtype	a[0] to a[i[0]-1 d[0]	I(2) to $I(I(1)+1)$ A(1) to A(I(1)) D(1)
a	ari old nd ni = count+1 If combiner is	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE	a[0] to a[i[0]-1 d[0]	I(2) to $I(I(1)+1)$ A(1) to A(I(1)) D(1)
a	arr old nd ni = count+1 If combiner is Cc	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE	a[0] to a[i[0]-1 d[0] EXED_BLOCK the	I(2) to I(I(1)+1) A(1) to A(I(1)) D(1) D(1)
a	ari old nd ni = count+1 If combiner is Cc	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE postructor argument unt	$a[0] \text{ to } a[i[0]-1]$ $d[0]$ EXED_BLOCK therefore the second	I(2)  to  I(I(1)+1) $A(1)  to  A(I(1))$ $D(1)$ nen $Fortran location$ $I(1)$
a	ari old nd ni = count+1 If combiner is $\overline{Cc}$ co blo	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength	$a[0] \text{ to } a[i[0]-1]$ $d[0]$ EXED_BLOCK therefore the second	I(2)  to  I(I(1)+1) $A(1)  to  A(I(1))$ $D(1)$ nen $I(1)$ $I(1)$ $I(2)$
a	arr old nd ni = count+1 If combiner is $\overline{Cc}$ co blo arr	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements	$\begin{array}{c} a[0] \text{ to } a[i[0]-1] \\ d[0] \\ \hline \\ $	$[1(2) \text{ to } I(I(1)+1) \\ A(1) \text{ to } A(I(1)) \\ D(1) \\ \hline \\ $
a	arr old nd ni = count+1 If combiner is $\overline{Cc}$ co blo arr	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength	$a[0] \text{ to } a[i[0]-1]$ $d[0]$ EXED_BLOCK therefore the second	I(2)  to  I(I(1)+1) $A(1)  to  A(I(1))$ $D(1)$ nen $I(1)$ $I(1)$ $I(2)$
	ar: old nd ni = count+1 If combiner is Cc co blo ar: old	ray_of_displacements dtype ., na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype	$\begin{array}{c} a[0] \text{ to } a[i[0]-1] \\ d[0] \\ \hline \\ $	$[1(2) \text{ to } I(I(1)+1) \\ A(1) \text{ to } A(I(1)) \\ D(1) \\ \hline \\ $
	ar: old nd ni = count+1 If combiner is $\overline{Cc}$ blo ar: old nd ni = count+2	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype 2, na = 0, nd = 1.	$a[0] \text{ to } a[i[0]-1] \\ d[0]$ EXED_BLOCK the second sec	I(2)  to  I(I(1)+1) $A(1)  to  A(I(1))$ $D(1)$ nen $I(1)$ $I(2)$ $I(3)  to  I(I(1)+2)$ $D(1)$
	ar: old nd ni = count+1 If combiner is $\overline{Cc}$ blo ar: old nd ni = count+2	ray_of_displacements dtype ., na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype	$a[0] \text{ to } a[i[0]-1] \\ d[0]$ EXED_BLOCK the second sec	I(2)  to  I(I(1)+1) $A(1)  to  A(I(1))$ $D(1)$ nen $I(1)$ $I(2)$ $I(3)  to  I(I(1)+2)$ $D(1)$
	ar: <u>old</u> nd ni = count+1 If combiner is $\overline{Cc}$ co bla ar: <u>old</u> nd ni = count+2 If combiner is	ray_of_displacements dtype ., na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype 2, na = 0, nd = 1. s MPI_COMBINER_HINE	$a[0] \text{ to } a[i[0]-1] \\ d[0]$ EXED_BLOCK the second sec	I(2)  to  I(I(1)+1) $A(1)  to  A(I(1))$ $D(1)$ nen $I(1)$ $I(2)$ $I(3)  to  I(I(1)+2)$ $D(1)$
	ar: old nd ni = count+1 If combiner is $\overline{Cc}$ blo ar: old nd ni = count+2 If combiner is $\overline{Cc}$	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype 2, na = 0, nd = 1.	a[0] to a[i[0]-1 d[0] EXED_BLOCK th <u>C</u> i[0] i[1] i[2] to i[i[0]+1 d[0] DEXED_BLOCK	I(2) to $I(I(1)+1)$ A(1) to $A(I(1))$ D(1) nen Fortran location I(1) I(2) I(3) to $I(I(1)+2)$ D(1) then Fortran location
	ar: old nd ni = count+1 If combiner is $\overline{Cc}$ co blo ar: old nd ni = count+2 If combiner is $\overline{Cc}$ co co blo ar: old nd ni = count+2 Cc co blo ar: old co co blo ar: co co blo ar: co co blo ar: co co blo ar: co co blo ar: co co blo ar: co co blo ar: co co blo co co blo co co blo co co co co blo co co co co co co co co co c	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype 2, na = 0, nd = 1. s MPI_COMBINER_HINE	$a[0] \text{ to } a[i[0]-1]$ $d[0]$ EXED_BLOCK therefore the second	I(2)  to  I(I(1)+1) $A(1)  to  A(I(1))$ $D(1)$ nen $I(1)$ $I(2)$ $I(3)  to  I(I(1)+2)$ $D(1)$ then
	ar: <u>old</u> nd ni = count+1 If combiner is $\frac{Cc}{co}$ blo ar: <u>old</u> nd ni = count+2 If combiner is $\frac{Cc}{co}$ blo ar: <u>old</u> nd ni = count+2 If combiner is <u>Cc</u> <u>co</u> blo ar: <u>old</u>	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype 2, na = 0, nd = 1. s MPI_COMBINER_HINE onstructor argument ount	$\begin{array}{c} a[0] \text{ to } a[i[0]-1] \\ d[0] \\ \hline \\ \hline \\ \\ \\ \hline \\ \\ \\ \hline \\ \\ \\ \\ \hline \\ \\ \\ \\ \\ \hline \\$	I(2) to $I(I(1)+1)$ A(1) to $A(I(1))$ D(1) nen Fortran location I(1) I(2) I(3) to $I(I(1)+2)$ D(1) then Fortran location I(1) I(2)
	ar: old nd ni = count+1 If combiner is $\overline{Cc}$ co bla ar: old nd ni = count+2 If combiner is $\overline{Cc}$ bla ar: old ar: ar: old ar: old ar: ar: ar: ar: ar: ar: ar: ar:	ray_of_displacements dtype ., na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype 2, na = 0, nd = 1. s MPI_COMBINER_HINE onstructor argument ount ocklength	$\begin{array}{c} a[0] \text{ to } a[i[0]-1] \\ d[0] \\ \hline \\ \hline \\ \\ \\ \hline \\ \\ \\ \hline \\ \\ \\ \\ \hline \\ \\ \\ \\ \\ \hline \\$	I(2) to $I(I(1)+1)$ A(1) to $A(I(1))$ D(1) nen Fortran location I(1) I(2) I(3) to $I(I(1)+2)$ D(1) then Fortran location I(1) I(2)
a	ar: old nd ni = count+1 If combiner is $\overline{Cc}$ co ble ar: old nd ni = count+2 If combiner is $\overline{Cc}$ co ble ar: old ar:	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype 2, na = 0, nd = 1. s MPI_COMBINER_HINE onstructor argument ount ocklength ray_of_displacements dtype	$a[0] \text{ to } a[i[0]-1] \\ d[0]$ EXED_BLOCK the set of	$\begin{array}{c c} I(2) \text{ to } I(I(1)+1) \\ A(1) \text{ to } A(I(1)) \\ D(1) \\ \hline \\ $
a	arr old nd ni = count+1 If combiner is $\frac{Cc}{co}$ blo arr old nd ni = count+2 If combiner is $\frac{Cc}{co}$ blo arr old nd ni = count+2 If combiner is arr old nd ni = ni = 2, na = 1	ray_of_displacements dtype ., na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype 2, na = 0, nd = 1. s MPI_COMBINER_HINE onstructor argument ocklength ray_of_displacements dtype count, nd = 1.	$a[0] \text{ to } a[i[0]-1] \\ d[0]$ EXED_BLOCK the set of	$\begin{array}{c c} I(2) \text{ to } I(I(1)+1) \\ A(1) \text{ to } A(I(1)) \\ D(1) \\ \hline \\ $
a	arr old nd ni = count+1 If combiner is $\frac{Cc}{co}$ blo arr old nd ni = count+2 If combiner is $\frac{Cc}{co}$ blo arr old nd ni = count+2 If combiner is arr old nd ni = ni = 2, na = 1	ray_of_displacements dtype , na = count, nd = 1. s MPI_COMBINER_INDE onstructor argument unt ocklength ray_of_displacements dtype 2, na = 0, nd = 1. s MPI_COMBINER_HINE onstructor argument ount ocklength ray_of_displacements dtype	$a[0] \text{ to } a[i[0]-1] \\ d[0]$ EXED_BLOCK the set of	$\begin{array}{c c} I(2) \text{ to } I(I(1)+1) \\ A(1) \text{ to } A(I(1)) \\ D(1) \\ \hline \\ $

	Constructor arg	gument C	Fortran location
-	count	i[0]	I(1)
	array_of_blockl		]] $I(2) \text{ to } I(I(1)+1)$
	array_of_displa	cements $a[0]$ to $a[i[0]$	
	array_of_types	d[0] to $d[i[0]$	[-1] D(1) to D(I(1))
	t+1, $na = counter is MPI_COMBI$	, nd = count. NER_SUBARRAY then	
Constr	uctor argument	С	Fortran location
ndims		i[0]	I(1)
array_	of_sizes	i[1] to $i[i[0]]$	I(2)  to  I(I(1)+1)
array_	of_subsizes	i[i[0]+1] to $i[2*i[0]]$	I(I(1)+2) to $I(2*I(1)+1)$
	of_starts	i[2*i[0]+1] to $i[3*i[0]$	
order	_	i[3*i[0]+1]	I(3*I(1)+2]
oldtype	e	d[0]	D(1)
	ims+2, $na = 0$ , $na =$	NER_DARRAY then	
	ctor argument	С	Fortran location
size		i[0]	I(1)
$\operatorname{rank}$		i[1]	I(2)
ndims		i[2]	I(3)
array_of	_gsizes	i[3] to $i[i[2]+2]$	I(4)  to  I(I(3)+3)
array_of	_distribs	i[i[2]+3] to $i[2*i[2]+2$	
		i[2*i[2]+3] to $i[3*i[2]+3$	2] $I(2*I(3)+4)$ to $I(3*I(3)+3)$
array_of	_dargs	1[2,1[2]+3] to $1[3,1[2]+$	$[2] 1(2 \cdot 1(3) + 4) to 1(3 \cdot 1(3) + 3)$
array_of array_of		$i[3^*i[2]+3]$ to $i[4^*i[2]+$	
e e			
array_of		i[3*i[2]+3] to $i[4*i[2]+3]$	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$
array_of order oldtype and ni = $4^*$ nd	$E_{\text{psizes}}$ ims+4, na = 0, na =	$i[3^*i[2]+3]$ to $i[4^*i[2]+$ $i[4^*i[2]+3]$ d[0]	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4)
array_of order oldtype and ni = $4^*$ nd	$f_{psizes}$ ims+4, na = 0, notematical interval in the second	$i[3^*i[2]+3]$ to $i[4^*i[2]+$ $i[4^*i[2]+3]$ d[0] nd = 1. NER_F90_REAL then	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1)
array_of order oldtype and ni = $4^*$ nd	$E_{psizes}$ $ims+4, na = 0, no er is MPI_COMBIN Construct$	$i[3^*i[2]+3]$ to $i[4^*i[2]+$ $i[4^*i[2]+3]$ d[0] nd = 1. NER_F90_REAL then etor argument C I	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1)
array_of order oldtype and ni = $4^*$ nd	$\frac{1}{2} \text{ psizes}$ $\frac{1}{2} \text{ ims} + 4, \text{ na} = 0, \text{ n}$ $\frac{1}{2} \text{ construct}}{p}$	$i[3^*i[2]+3]$ to $i[4^*i[2]+$ $i[4^*i[2]+3]$ d[0] nd = 1. NER_F90_REAL then ctor argument C I i[0]	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1) Fortran location I(1)
array_of order oldtype and ni = $4$ *nd If combine and ni = 2, na	E_psizes ims+4, na = 0, n er is MPI_COMBI	$i[3^*i[2]+3]$ to $i[4^*i[2]+$ $i[4^*i[2]+3]$ d[0] nd = 1. NER_F90_REAL then etor argument C I	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1) Fortran location I(1) I(2)
array_of order oldtype and ni = $4$ *nd If combine and ni = 2, na	E_psizes ims+4, na = 0, n er is MPI_COMBII r m m r m m r m m r m m r m m m r m m m m m m m m	$i[3*i[2]+3] \text{ to } i[4*i[2]+$ $i[4*i[2]+3]$ $d[0]$ $nd = 1.$ $NER_F90_REAL \text{ then}$ $i[0]$ $i[1]$ $NER_F90_COMPLEX \text{ th}$	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1) Fortran location I(1) I(2) en
array_of order oldtype and ni = $4$ *nd If combine and ni = 2, na	E_psizes ims+4, na = 0, n er is MPI_COMBII r m m r m m r m m r m m r m m m r m m m m m m m m	i[3*i[2]+3]  to  i[4*i[2]+ $i[4*i[2]+3]$ $d[0]$ nd = 1. NER_F90_REAL then $ctor argument C I$ $i[0]$ $i[1]$ NER_F90_COMPLEX then etor argument C I	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1) Fortran location I(1) I(2) en
array_of order oldtype and ni = $4$ *nd If combine and ni = 2, na	E_psizes ims+4, na = 0, n er is MPI_COMBII r m m r m m r m m r m m r m m m r m m m m m m m m	$i[3*i[2]+3] \text{ to } i[4*i[2]+$ $i[4*i[2]+3]$ $d[0]$ $nd = 1.$ $NER_F90_REAL \text{ then}$ $contained C I$ $i[0]$ $i[1]$ $NER_F90_COMPLEX \text{ th}$ $contained C I$ $i[0]$	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1) Fortran location I(1) I(2) en Fortran location I(1)
array_of order oldtype and ni = $4$ *nd If combine and ni = 2, na	E_psizes ims+4, na = 0, n er is MPI_COMBI <u>Construct</u> p r a = 0, nd = 0. er is MPI_COMBI <u>Construct</u>	i[3*i[2]+3]  to  i[4*i[2]+ $i[4*i[2]+3]$ $d[0]$ nd = 1. NER_F90_REAL then $ctor argument C I$ $i[0]$ $i[1]$ NER_F90_COMPLEX then etor argument C I	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1) Fortran location I(1) I(2) en
array_of order oldtype and ni = $4$ *nd If combined and ni = 2, na If combined and ni = 2, na	E_psizes ims+4, na = 0, n er is MPI_COMBII r m = 0, nd = 0. r m = 0, nd = 0. r r m = 0, nd = 0. r r r r r r r r	$i[3*i[2]+3] \text{ to } i[4*i[2]+$ $i[4*i[2]+3]$ $d[0]$ $nd = 1.$ $NER_F90_REAL \text{ then}$ $contained C I$ $i[0]$ $i[1]$ $NER_F90_COMPLEX \text{ th}$ $contained C I$ $i[0]$	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1) Fortran location I(1) I(2) en Fortran location I(1) I(2)
array_of order oldtype and ni = $4$ *nd If combined and ni = 2, na If combined and ni = 2, na	$\frac{1}{2} \text{psizes}$ $\frac{1}{2} \text{psizes}$ $\frac{1}{2} \text{psizes}$ $\frac{1}{2} \text{construct}$ $\frac{1}{2$	i[3*i[2]+3]  to  i[4*i[2]+ $i[4*i[2]+3]$ $d[0]$ nd = 1. NER_F90_REAL then $i[0]$ $i[1]$ NER_F90_COMPLEX th etor argument C I $i[0]$ $i[1]$ NER_F90_INTEGER the	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1) Fortran location I(1) I(2) en Fortran location I(1) I(2) n
array_of order oldtype and ni = $4$ *nd If combined and ni = 2, na If combined and ni = 2, na	$\frac{1}{2} \text{psizes}$ $\frac{1}{2} \text{psizes}$ $\frac{1}{2} \text{psizes}$ $\frac{1}{2} \text{construct}$ $\frac{1}{2$	$i[3*i[2]+3] \text{ to } i[4*i[2]+$ $i[4*i[2]+3]$ $d[0]$ $nd = 1.$ $NER_F90_REAL \text{ then}$ $cor argument C I$ $i[0]$ $i[1]$ $NER_F90_COMPLEX \text{ th}$ $cor argument C I$ $i[0]$ $i[1]$ $NER_F90_INTEGER \text{ the}$	2] $I(3*I(3)+4)$ to $I(4*I(3)+3)$ I(4*I(3)+4) D(1) Fortran location I(1) I(2) en Fortran location I(1) I(2)

```
1
     and ni = 1, na = 0, nd = 0.
\mathbf{2}
         If combiner is MPI_COMBINER_RESIZED then
3
                                                 С
4
                         Constructor argument
                                                      Fortran location
5
                         lb
                                                a[0]
                                                            A(1)
6
                                                            A(2)
                         extent
                                                a[1]
7
                                                d[0]
                         oldtype
                                                            D(1)
8
     and ni = 0, na = 2, nd = 1.
9
10
^{11}
     5.1.14 Examples
12
     The following examples illustrate the use of derived datatypes.
13
14
     Example 5.13 Send and receive a section of a 3D array.
15
16
     REAL a(100, 100, 100), e(9, 9, 9)
17
     INTEGER oneslice, twoslice, threeslice, myrank, ierr
18
     INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
19
     INTEGER status (MPI_STATUS_SIZE)
20
21
     ! extract the section a(1:17:2, 3:11, 2:10)
22
     ! and store it in e(:,:,:).
23
^{24}
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
25
26
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
27
28
     ! create datatype for a 1D section
^{29}
     CALL MPI_TYPE_VECTOR(9, 1, 2, MPI_REAL, oneslice, ierr)
30
31
     ! create datatype for a 2D section
32
     CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*sizeofreal, oneslice, &
33
                                            twoslice, ierr)
34
35
     ! create datatype for the entire section
36
     CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*100*sizeofreal, twoslice, &
37
                                      threeslice, ierr)
38
39
     CALL MPI_TYPE_COMMIT(threeslice, ierr)
40
     CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9, &
41
                         MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
42
43
     Example 5.14 Copy the (strictly) lower triangular part of a matrix.
44
45
46
47
48
```

```
1
REAL a(100,100), b(100,100)
                                                                                    2
INTEGER disp(100), blocklen(100), ltype, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)
! copy lower triangular part of array a
                                                                                    5
! onto lower triangular part of array b
                                                                                    6
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
                                                                                    9
                                                                                    10
! compute start and size of each column
                                                                                    11
DO i=1,100
   disp(i) = 100*(i-1) + i
                                                                                    12
   blocklen(i) = 100-i
                                                                                    13
END DO
                                                                                    14
                                                                                    15
! create datatype for lower triangular part
                                                                                    16
                                                                                    17
CALL MPI_TYPE_INDEXED(100, blocklen, disp, MPI_REAL, ltype, ierr)
                                                                                    18
                                                                                    19
CALL MPI_TYPE_COMMIT(ltype, ierr)
CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1, &
                                                                                    20
                                                                                    21
                   ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
                                                                                    22
                                                                                    23
Example 5.15 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
! 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
! create datatype for one row
                                                                                    36
CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
                                                                                    37
                                                                                    38
! 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
! 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 5.16 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
\mathbf{5}
     INTEGER status(MPI_STATUS_SIZE)
6
7
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
8
9
     ! transpose matrix a onto b
10
11
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
12
13
     ! create datatype for one row
14
     CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
15
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
22
     ! send 100 rows and receive in column major order
23
     CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100, &
^{24}
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
25
26
     Example 5.17 Use of MPI datatypes to manipulate an array of structures.
27
28
     struct Partstruct
29
     ſ
30
                type; /* particle type */
        int
^{31}
        double d[6];
                       /* particle coordinates */
32
        char b[7]; /* some additional information */
33
     };
34
35
     struct Partstruct
                            particle[1000];
36
37
     int
                   i, dest, tag;
38
     MPI_Comm
                   comm;
39
40
41
     /* build datatype describing structure */
42
43
     MPI_Datatype Particlestruct, Particletype;
44
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
45
     int
                   blocklen[3] = \{1, 6, 7\};
46
     MPI_Aint
                   disp[3];
47
     MPI_Aint
                   base, lb, sizeofentry;
48
```

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

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

```
/* an alternative solution to 4.3 */
                                                                                       1
                                                                                       \mathbf{2}
                                                                                       3
MPI_Datatype Twodouble;
                                                                                       4
MPI_Type_contiguous(2, MPI_DOUBLE, &Twodouble);
                                                                                       5
                                                                                       6
MPI_Datatype Onepair;
                          /* datatype for one pair of coordinates, with
                                                                                       7
                            the extent of one particle entry */
                                                                                       9
                                                                                      10
MPI_Type_create_resized(Twodouble, 0, sizeofentry, &Onepair );
                                                                                      11
MPI_Type_commit(&Onepair);
MPI_Send(particle[0].d, 1000, Onepair, dest, tag, comm);
                                                                                      12
                                                                                      13
                                                                                      14
                                                                                      15
Example 5.18 The same manipulations as in the previous example, but use absolute
                                                                                      16
addresses in datatypes.
                                                                                      17
                                                                                      18
struct Partstruct
                                                                                      19
ſ
                                                                                      20
    int
            type;
                                                                                      21
    double d[6];
                                                                                      22
    char
            b[7];
                                                                                      23
};
                                                                                      ^{24}
                                                                                      25
struct Partstruct particle[1000];
                                                                                      26
                                                                                      27
/* build datatype describing first array entry */
                                                                                      28
                                                                                      29
MPI_Datatype Particletype;
                                                                                      30
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
                                                                                      31
              block[3] = \{1, 6, 7\};
int
                                                                                      32
MPI_Aint
              disp[3];
                                                                                      33
                                                                                      34
MPI_Get_address(particle, disp);
                                                                                      35
MPI_Get_address(particle[0].d, disp+1);
                                                                                      36
MPI_Get_address(particle[0].b, disp+2);
                                                                                      37
MPI_Type_create_struct(3, block, disp, type, &Particletype);
                                                                                      38
                                                                                      39
/* Particletype describes first array entry -- using absolute
                                                                                      40
   addresses */
                                                                                      41
                                                                                      42
/* 5.1: send the entire array */
                                                                                      43
                                                                                      44
MPI_Type_commit(&Particletype);
                                                                                      45
MPI_Send(MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
                                                                                      46
                                                                                      47
```

```
1
     /* 5.2: send the entries of type zero,
\mathbf{2}
              preceded by the number of such entries */
3
4
     MPI_Datatype Zparticles, Ztype;
5
6
                   zdisp[1000];
     int
7
     int
                   zblock[1000], i, j, k;
8
                   zzblock[2] = {1,1};
     int
9
     MPI_Datatype zztype[2];
10
     MPI_Aint
                   zzdisp[2];
^{11}
12
     j=0;
13
     for (i=0; i < 1000; i++)
14
         if (particle[i].type == 0)
15
              {
16
                  for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);</pre>
17
                  zdisp[j] = i;
18
                  zblock[j] = k-i;
19
                  j++;
20
                  i = k;
21
              }
22
     MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
23
     /* Zparticles describe particles with type zero, using
^{24}
        their absolute addresses*/
25
26
     /* prepend particle count */
27
     MPI_Get_address(&j, zzdisp);
28
     zzdisp[1] = (MPI_Aint)0;
29
     zztype[0] = MPI_INT;
30
     zztype[1] = Zparticles;
^{31}
     MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
32
33
     MPI_Type_commit(&Ztype);
34
     MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
35
36
37
     Example 5.19 This example shows how datatypes can be used to handle unions.
38
39
     union {
40
        int
                 ival;
41
        float
                 fval;
42
            } u[1000];
43
44
     int
              i, utype;
45
46
     /* All entries of u have identical type; variable
47
        utype keeps track of their current type */
48
```

```
1
                                                                                      2
MPI_Datatype
                mpi_utype[2];
                ubase, extent;
MPI_Aint
/* compute an MPI datatype for each possible union type;
                                                                                      5
                                                                                      6
   assume values are left-aligned in union storage. */
MPI_Get_address(u, &ubase);
MPI_Get_address(u+1, &extent);
                                                                                      a
                                                                                      10
extent = MPI_Aint_diff(extent, ubase);
                                                                                      11
MPI_Type_create_resized(MPI_INT, 0, extent, &mpi_utype[0]);
                                                                                      12
                                                                                      13
                                                                                      14
MPI_Type_create_resized(MPI_FLOAT, 0, extent, &mpi_utype[1]);
                                                                                      15
                                                                                      16
for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);</pre>
                                                                                      17
                                                                                      18
/* actual communication */
                                                                                      19
MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
                                                                                      20
                                                                                      21
Example 5.20 This example shows how a datatype can be decoded. The routine
                                                                                      22
printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for
                                                                                      23
datatypes that are not predefined.
                                                                                      24
                                                                                      25
/*
                                                                                      26
  Example of decoding a datatype.
                                                                                      27
  Returns 0 if the datatype is predefined, 1 otherwise
                                                                                      28
                                                                                      29
 */
                                                                                      30
#include <stdio.h>
                                                                                      31
#include <stdlib.h>
#include "mpi.h"
                                                                                      32
                                                                                      33
int printdatatype(MPI_Datatype datatype)
                                                                                      34
{
    int *array_of_ints;
                                                                                      35
                                                                                      36
    MPI_Aint *array_of_adds;
                                                                                      37
    MPI_Datatype *array_of_dtypes;
                                                                                      38
    int num_ints, num_adds, num_dtypes, combiner;
                                                                                      39
    int i;
                                                                                      40
                                                                                      41
    MPI_Type_get_envelope(datatype,
                                                                                      42
                            &num_ints, &num_adds, &num_dtypes, &combiner);
    switch (combiner) {
                                                                                      43
                                                                                      44
    case MPI_COMBINER_NAMED:
                                                                                      45
        printf("Datatype is named:");
                                                                                      46
        /* To print the specific type, we can match against the
                                                                                      47
            predefined forms. We can NOT use a switch statement here
                                                                                      48
            We could also use MPI_TYPE_GET_NAME if we prefered to use
```

```
1
                 names that the user may have changed.
2
               */
3
                                                 printf("MPI_INT\n");
             if
                      (datatype == MPI_INT)
4
             else if (datatype == MPI_DOUBLE) printf("MPI_DOUBLE\n");
5
              ... else test for other types ...
6
             return 0;
7
             break;
8
         case MPI_COMBINER_STRUCT:
9
         case MPI_COMBINER_STRUCT_INTEGER:
10
             printf("Datatype is struct containing");
11
                               = (int *)malloc(num_ints * sizeof(int));
             array_of_ints
12
             array_of_adds
13
                          (MPI_Aint *) malloc(num_adds * sizeof(MPI_Aint));
14
             array_of_dtypes = (MPI_Datatype *)
15
                  malloc(num_dtypes * sizeof(MPI_Datatype));
16
             MPI_Type_get_contents(datatype, num_ints, num_adds, num_dtypes,
17
                                  array_of_ints, array_of_adds, array_of_dtypes);
18
             printf(" %d datatypes:\n", array_of_ints[0]);
19
             for (i=0; i<array_of_ints[0]; i++) {</pre>
20
                  printf("blocklength %d, displacement %ld, type:\n",
21
                          array_of_ints[i+1], (long)array_of_adds[i]);
22
                  if (printdatatype(array_of_dtypes[i])) {
23
                      /* Note that we free the type ONLY if it
24
                         is not predefined */
25
                      MPI_Type_free(&array_of_dtypes[i]);
26
                  }
27
             }
28
             free(array_of_ints);
29
             free(array_of_adds);
30
             free(array_of_dtypes);
31
             break;
32
             . . .
                  other combiner values ...
33
         default:
34
             printf("Unrecognized combiner type\n");
35
         7
36
         return 1;
37
     }
38
39
           Pack and Unpack
     5.2
```

41 Some existing communication libraries provide pack/unpack functions for sending noncon-42tiguous data. In these, the user explicitly packs data into a contiguous buffer before sending 43 it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are 44 described in Section 5.1, allow one, in most cases, to avoid explicit packing and unpacking. 45 The user specifies the layout of the data to be sent or received, and the communication 46 library directly accesses a noncontiguous buffer. The pack/unpack routines are provided 47 for compatibility with previous libraries. Also, they provide some functionality that is not 48

# **Unofficial Draft for Comment Only**

otherwise available in MPI. For instance, a message can be received in several parts, where the receive operation done on a later part may depend on the content of a former part. Another use is that outgoing messages may be explicitly buffered in user supplied space, thus overriding the system buffering policy. Finally, the availability of pack and unpack operations facilitates the development of additional communication libraries layered on top of MPI.

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

IN	inbuf	input buffer start (choice)	10
IN	incount	number of input data items (non-negative integer)	11
INI		1 0 0 ,	12
IN	datatype	datatype of each input data item (handle)	13
OUT	outbuf	output buffer start (choice)	14
IN	outsize	output buffer size, in bytes (non-negative integer)	15
			16
INOUT	position	current position in buffer, in bytes (integer)	17
IN	comm	communicator for packed message (handle)	18
			19

# C binding

int MPI\_Pack\_c(const void \*inbuf, MPI\_Count incount, MPI\_Datatype datatype, void \*outbuf, MPI\_Count outsize, MPI\_Count \*position, MPI\_Comm comm)

# Fortran 2008 binding

28MPI\_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror) 29 TYPE(\*), DIMENSION(...), INTENT(IN) :: inbuf 30 INTEGER, INTENT(IN) :: incount, outsize 31TYPE(MPI\_Datatype), INTENT(IN) :: datatype 32 TYPE(\*), DIMENSION(..) :: outbuf 33 INTEGER, INTENT(INOUT) :: position 34 TYPE(MPI\_Comm), INTENT(IN) :: comm 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI\_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror) 37 TYPE(\*), DIMENSION(...), INTENT(IN) :: inbuf 38 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: incount, outsize 39 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 40 TYPE(\*), DIMENSION(..) :: outbuf 41 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(INOUT) :: position 42TYPE(MPI\_Comm), INTENT(IN) :: comm 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 45Fortran binding

1

 $\mathbf{2}$ 

3

4

5 6

7 8

9

20

21

22

23

24

25

1	INTEG	ER INCOUNT,	DATATYPE,	OUTSIZE, POSITION, COMM, IERROR			
2 3 4 5 6 7 8 9 10 11 12 13	space spect allowed in bytes, star a communi The in packing. p of position packed met	ified by outbuf MPI_SEND. T ting at the add ication buffer f nput value of osition is incre- is the first loca	and outsize. The output b lress outbuf of or a message position is the emented by t ation in the nm argument	ffer specified by inbuf, incount, datatype into the buffer . The input buffer can be any communication buffer ouffer is a contiguous storage area containing outsize (length is counted in <i>bytes</i> , not elements, as if it were e of type MPI_PACKED). ne first location in the output buffer to be used for he size of the packed message, and the output value output buffer following the locations occupied by the t is the communicator that will be subsequently used			
14	MPI_UNPA	ACK(inbuf, insiz	ze, position, c	outbuf, outcount, datatype, comm)			
15 16	IN	inbuf		input buffer start (choice)			
17	IN	insize		size of input buffer, in bytes (non-negative integer)			
18	INOUT	position		current position in bytes (integer)			
19 20	OUT	outbuf		output buffer start (choice)			
21	IN	outcount		number of items to be unpacked (integer)			
22	IN	datatype		datatype of each output data item (handle)			
23 24	IN	comm		communicator for packed message (handle)			
25 26 27 28	C binding int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf,						
29 30 31 32	int MPI_U	-	itbuf, MPI_	buf, MPI_Count insize, MPI_Count *position, Count outcount, MPI_Datatype datatype,			
33		008 binding					
34							
35 36	<pre>ierror) TYPE(*), DIMENSION(), INTENT(IN) :: inbuf</pre>						
37	INTEGER, INTENT(IN) :: insize, outcount						
38	INTEGER, INTENT(INOUT) :: position						
39	TYPE(*), DIMENSION() :: outbuf						
40	TYPE(MPI_Datatype), INTENT(IN) :: datatype						
41 42	TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror						
43			-				
44	MPI_Unpac	k(inbuf, ins: ierror)	ize, posit:	ion, outbuf, outcount, datatype, comm,			
45	TYPE(			ENT(IN) ·· inbuf			
46	TYPE(*), DIMENSION(), INTENT(IN) :: inbuf INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount						
47 48	INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position						

```
TYPE(*), DIMENSION(..) :: outbuf
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
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.

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

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

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

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

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

1

2

5

6

8

9

10 11

12 13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29 30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

42

43

44

45

1 A message sent with any type (including MPI\_PACKED) can be received using the type  $\mathbf{2}$ MPI\_PACKED. Such a message can then be unpacked by calls to MPI\_UNPACK. 3 A packing unit (or a message created by a regular, "typed" send) can be unpacked into 4 several successive messages. This is effected by several successive related calls to 5MPI\_UNPACK, where the first call provides position = 0, and each successive call inputs the 6 value of position that was output by the previous call, and the same values for inbuf, insize  $\overline{7}$ and comm. 8 The concatenation of two packing units is not necessarily a packing unit; nor is a 9 substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two 10 packing units and then unpack the result as one packing unit; nor can one unpack a substring 11of a packing unit as a separate packing unit. Each packing unit, that was created by a related 12sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of 13related unpack calls. 14The restriction on "atomic" packing and unpacking of packing units Rationale. 1516allows 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 17 heterogeneous environment) (End of rationale.) 18 19The following call allows the user to find out how much space is needed to pack a 20message and, thus, manage space allocation for buffers. 2122 23MPI\_PACK\_SIZE(incount, datatype, comm, size) 24count argument to packing call (non-negative integer) IN incount 2526IN datatype argument to packing call (handle) datatype 27IN comm communicator argument to packing call (handle) 28OUT upper bound on size of packed message, in bytes size 29 (non-negative integer) 30  $^{31}$ 32 C binding 33 int MPI\_Pack\_size(int incount, MPI\_Datatype datatype, MPI\_Comm comm, 34int \*size) 35 int MPI\_Pack\_size\_c(MPI\_Count incount, MPI\_Datatype datatype, 36 MPI\_Comm comm, MPI\_Count \*size) 37 38Fortran 2008 binding 39 MPI\_Pack\_size(incount, datatype, comm, size, ierror) 40INTEGER, INTENT(IN) :: incount 41 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 42TYPE(MPI\_Comm), INTENT(IN) :: comm 43 INTEGER, INTENT(OUT) :: size 44INTEGER, OPTIONAL, INTENT(OUT) :: ierror 45MPI\_Pack\_size(incount, datatype, comm, size, ierror) 46INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: incount 47TYPE(MPI\_Datatype), INTENT(IN) :: datatype

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          1
                                                                                          \mathbf{2}
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
                                                                                          3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          4
Fortran binding
                                                                                          5
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
                                                                                          6
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
                                                                                          7
                                                                                          8
    A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound
                                                                                          9
on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype,
                                                                                          10
outbuf, outcount, position, comm). If the packed size of the datatype cannot be expressed
                                                                                          11
by the size parameter, then MPI_PACK_SIZE sets the value of size to MPI_UNDEFINED.
                                                                                         12
     Rationale. The call returns an upper bound, rather than an exact bound, since the
                                                                                         13
     exact amount of space needed to pack the message may depend on the context (e.g.,
                                                                                         14
     first message packed in a packing unit may take more space). (End of rationale.)
                                                                                          15
                                                                                          16
                                                                                          17
Example 5.21 An example using MPI_PACK.
                                                                                          18
                                                                                          19
int
            position, i, j, a[2];
            buff[1000];
                                                                                          20
char
                                                                                         21
                                                                                         22
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                                                                                         23
if (myrank == 0)
                                                                                          ^{24}
{
                                                                                          25
    /* SENDER CODE */
                                                                                          26
                                                                                         27
    position = 0;
    MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
                                                                                         28
                                                                                         29
    MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
    MPI_Send(buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
                                                                                         30
                                                                                          31
}
else /* RECEIVER CODE */
                                                                                          32
    MPI_Recv(a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                                                                                         33
                                                                                         34
                                                                                         35
Example 5.22 An elaborate example.
                                                                                         36
                                                                                         37
      position, i;
int
                                                                                          38
float a[1000];
                                                                                          39
      buff[1000];
char
                                                                                          40
                                                                                          41
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                                                                                          42
if (myrank == 0)
{
                                                                                          43
                                                                                          44
    /* SENDER CODE */
                                                                                          45
                                                                                          46
    int len[2];
                                                                                          47
    MPI_Aint disp[2];
                                                                                          48
    MPI_Datatype type[2], newtype;
```

```
1
\mathbf{2}
         /* build datatype for i followed by a[0]...a[i-1] */
3
4
         len[0] = 1;
5
         len[1] = i;
6
         MPI_Get_address(&i, disp);
7
         MPI_Get_address(a, disp+1);
8
         type[0] = MPI_INT;
9
         type[1] = MPI_FLOAT;
10
         MPI_Type_create_struct(2, len, disp, type, &newtype);
11
         MPI_Type_commit(&newtype);
12
         /* Pack i followed by a[0]...a[i-1]*/
13
14
15
         position = 0;
16
         MPI_Pack(MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);
17
18
         /* Send */
19
20
         MPI_Send(buff, position, MPI_PACKED, 1, 0,
21
                   MPI_COMM_WORLD);
22
23
     /* ****
^{24}
        One can replace the last three lines with
25
        MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
26
        **** */
27
     }
28
     else if (myrank == 1)
29
     ſ
30
         /* RECEIVER CODE */
^{31}
32
         MPI_Status status;
33
34
          /* Receive */
35
36
         MPI_Recv(buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
37
38
         /* Unpack i */
39
40
         position = 0;
41
         MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
42
         /* Unpack a[0]...a[i-1] */
43
44
         MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
45
     }
46
47
     Example 5.23 Each process sends a count, followed by count characters to the root; the
```

root concatenates all characters into one string.

```
int count, gsize, counts[64], totalcount, k1, k2, k,
     displs[64], position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
      /* allocate local pack buffer */
MPI_Pack_size(1, MPI_INT, comm, &k1);
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
k = k1+k2;
lbuf = (char *)malloc(k);
      /* pack count, followed by count characters
position = 0;
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);
if (myrank != root) {
    /* gather at root sizes of all packed messages */
    MPI_Gather(&position, 1, MPI_INT, NULL, 0,
               MPI_DATATYPE_NULL, root, comm);
    /* gather at root packed messages */
    MPI_Gatherv(lbuf, position, MPI_PACKED, NULL,
                NULL, NULL, MPI_DATATYPE_NULL, root, comm);
} else {
           /* root code */
    /* gather sizes of all packed messages */
    MPI_Gather(&position, 1, MPI_INT, counts, 1,
               MPI_INT, root, comm);
    /* gather all packed messages */
    displs[0] = 0;
    for (i=1; i < gsize; i++)</pre>
        displs[i] = displs[i-1] + counts[i-1];
    totalcount = displs[gsize-1] + counts[gsize-1];
    rbuf = (char *)malloc(totalcount);
    cbuf = (char *)malloc(totalcount);
    MPI_Gatherv(lbuf, position, MPI_PACKED, rbuf,
                counts, displs, MPI_PACKED, root, comm);
    /* unpack all messages and concatenate strings */
    concat_pos = 0;
    for (i=0; i < gsize; i++) {</pre>
        position = 0;
```

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 44

45

46

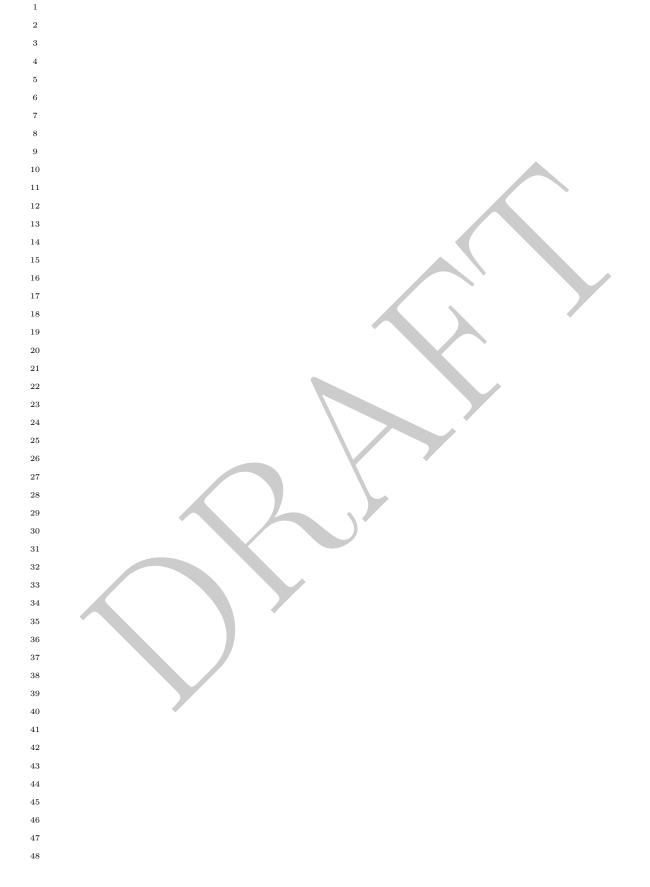
47

```
1
               MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
2
                            &position, &count, 1, MPI_INT, comm);
3
               MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
4
                            &position, cbuf+concat_pos, count, MPI_CHAR, comm);
5
               concat_pos += count;
6
          }
7
          cbuf[concat_pos] = '\0';
8
     }
9
10
     5.3
            Canonical MPI_PACK and MPI_UNPACK
11
12
     These functions read/write data to/from the buffer in the "external32" data format specified
13
     in Section 14.5.2, and calculate the size needed for packing. Their first arguments specify
14
     the data format, for future extensibility, but currently the only valid value of the datarep
15
     argument is "external32".
16
17
           Advice to users. These functions could be used, for example, to send typed data in a
18
           portable format from one MPI implementation to another. (End of advice to users.)
19
          The buffer will contain exactly the packed data, without headers. MPI_BYTE should
20
     be used to send and receive data that is packed using MPI_PACK_EXTERNAL.
21
22
           Rationale. MPI_PACK_EXTERNAL specifies that there is no header on the message
23
           and further specifies the exact format of the data. Since MPI_PACK may (and is
24
           allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed
25
           with MPI_PACK_EXTERNAL. (End of rationale.)
26
27
28
29
     MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)
30
       IN
                 datarep
                                              data representation (string)
^{31}
       IN
                 inbuf
                                              input buffer start (choice)
32
33
       IN
                                              number of input data items (integer)
                 incount
34
       IN
                 datatype
                                              datatype of each input data item (handle)
35
       OUT
                 outbuf
                                              output buffer start (choice)
36
37
       IN
                 outsize
                                              output buffer size, in bytes (integer)
38
       INOUT
                 position
                                              current position in buffer, in bytes (integer)
39
40
     C binding
41
     int MPI_Pack_external(const char datarep[], const void *inbuf, int incount,
42
                     MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
43
                     MPI_Aint *position)
44
45
     int MPI_Pack_external_c(const char datarep[], const void *inbuf,
46
                     MPI_Count incount, MPI_Datatype datatype, void *outbuf,
47
                     MPI_Count outsize, MPI_Count *position)
48
```

Fortran 2008 binding			
			2
	position, ierror)		3
	CTER(LEN=*), INTENT(IN) :	-	4
	*), DIMENSION(), INTENT		5
	ER, INTENT(IN) :: incount		6
	MPI_Datatype), INTENT(IN)		7
	*), DIMENSION() :: outb		° 9
	ER(KIND=MPI_ADDRESS_KIND)		10
	ER(KIND=MP1_ADDRESS_KIND) ER, OPTIONAL, INTENT(OUT)	, INTENT(INOUT) :: position	11
INIEG	ER, OFIIONAL, INIENI(001)	161101	12
MPI_Pack_	<pre>external(datarep, inbuf,</pre>	incount, datatype, outbuf, outsize,	13
	position, ierror)		14
	CTER(LEN=*), INTENT(IN) :	-	15
	*), DIMENSION(), INTENT		16
		INTENT(IN) :: incount, outsize	17
	<pre>MPI_Datatype), INTENT(IN) *), DIMENSION() :: outb</pre>		18
		INTENT(INOUT) :: position	19
	ER, OPTIONAL, INTENT(OUT)	-	20
		161101	21
Fortran b	3		22
MPI_PACK_		INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	23 24
	POSITION, IERROR)		24 25
	CTER*(*) DATAREP		26
• 1	> INBUF(*), OUTBUF(*)		27
	ER INCOUNT, DATATYPE, IER ER(KIND=MPI_ADDRESS_KIND)		28
INIEG	ER(KIND-HFI_ADDRESS_KIND)	UUISIZE, FUSITION	29
			30
	CK EXTERNAL (dataren inhu	f, insize, position, outbuf, outcount, datatype)	31
		,	32
IN	datarep	data representation (string)	33
IN	inbuf	input buffer start (choice)	34
IN	insize	input buffer size, in bytes (integer)	35
INOUT	position	current position in buffer, in bytes (integer)	36
		-	37 38
OUT	outbuf	output buffer start (choice)	39
IN	outcount	number of output data items (integer)	40
IN	datatype	datatype of output data item (handle)	41
			42
C binding	C binding 43		
int MPI_U	npack_external(const char	datarep[], const void *inbuf,	44
	MPI_Aint insize, MPI_	Aint *position, void *outbuf,	45
	int outcount, MPI_Dat	catype datatype)	46
			47

```
1
     int MPI_Unpack_external_c(const char datarep[], const void *inbuf,
\mathbf{2}
                   MPI_Count insize, MPI_Count *position, void *outbuf,
3
                   MPI_Count outcount, MPI_Datatype datatype)
4
     Fortran 2008 binding
5
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
6
                   datatype, ierror)
7
         CHARACTER(LEN=*), INTENT(IN) :: datarep
8
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
9
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
10
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
11
         TYPE(*), DIMENSION(...) :: outbuf
12
         INTEGER, INTENT(IN) :: outcount
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
17
                   datatype, ierror)
18
         CHARACTER(LEN=*), INTENT(IN) :: datarep
19
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
21
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
22
         TYPE(*), DIMENSION(..) :: outbuf
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     Fortran binding
26
     MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
27
                   DATATYPE, IERROR)
28
         CHARACTER*(*) DATAREP
29
         <type> INBUF(*), OUTBUF(*)
30
         INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
31
         INTEGER OUTCOUNT, DATATYPE, IERROR
32
33
34
     MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)
35
36
       IN
                datarep
                                           data representation (string)
37
       IN
                incount
                                           number of input data items (integer)
38
       IN
                datatype
                                           datatype of each input data item (handle)
39
40
       OUT
                                           output buffer size, in bytes (integer)
                size
41
42
     C binding
43
     int MPI_Pack_external_size(const char datarep[], int incount,
44
                   MPI_Datatype datatype, MPI_Aint *size)
45
     int MPI_Pack_external_size_c(const char datarep[], MPI_Count incount,
46
47
                   MPI_Datatype datatype, MPI_Count *size)
48
```

Fortran 2008 binding  $\mathbf{2}$ MPI\_Pack\_external\_size(datarep, incount, datatype, size, ierror) CHARACTER(LEN=\*), INTENT(IN) :: datarep INTEGER, INTENT(IN) :: incount TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_Pack\_external\_size(datarep, incount, datatype, size, ierror) CHARACTER(LEN=\*), INTENT(IN) :: datarep INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: incount TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI\_PACK\_EXTERNAL\_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR) CHARACTER\*(\*) DATAREP INTEGER INCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) SIZE  $^{24}$  $^{31}$ 



## Chapter 6

# **Collective Communication**

#### 6.1 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 6.3 and Section 6.12.1).
- MPI\_BCAST, MPI\_IBCAST: Broadcast from one member to all members of a group (Section 6.4 and Section 6.12.2). This is shown as "broadcast" in Figure 6.1.
- MPI\_GATHER, MPI\_IGATHER, MPI\_GATHERV, MPI\_IGATHERV: Gather data from all members of a group to one member (Section 6.5 and Section 6.12.3). This is shown as "gather" in Figure 6.1.
- MPI\_SCATTER, MPI\_ISCATTER, MPI\_SCATTERV, MPI\_ISCATTERV: Scatter data from one member to all members of a group (Section 6.6 and Section 6.12.4). This is shown as "scatter" in Figure 6.1.
- MPI\_ALLGATHER, MPI\_IALLGATHER, MPI\_ALLGATHERV, MPI\_IALLGATHERV: A variation on Gather where all members of a group receive the result (Section 6.7 and Section 6.12.5). This is shown as "allgather" in Figure 6.1.
- 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 6.8 and Section 6.12.6). This is shown as "complete exchange" in Figure 6.1.
- 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 6.9.6 and Section 6.12.8) and a variation where the result is returned to only one member (Section 6.9 and Section 6.12.7).
- MPI\_REDUCE\_SCATTER\_BLOCK, MPI\_IREDUCE\_SCATTER\_BLOCK, MPI\_REDUCE\_SCATTER, MPI\_IREDUCE\_SCATTER: A combined reduction and scatter operation (Section 6.10, Section 6.12.9, and Section 6.12.10).

4

34

35

36

37

38

• MPI\_SCAN, MPI\_ISCAN, MPI\_EXSCAN, MPI\_IEXSCAN: Scan across all members of a group (also called prefix) (Section 6.11, Section 6.11.2, Section 6.12.11, and Section 6.12.12).

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 6.2. 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 5. 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 5 for 13 information concerning communication buffers, general datatypes and type matching rules, 14and to Chapter 7 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 5.1) 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 participating MPI processes. 28

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 6.14.

*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 implementations 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.

#### Unofficial Draft for Comment Only

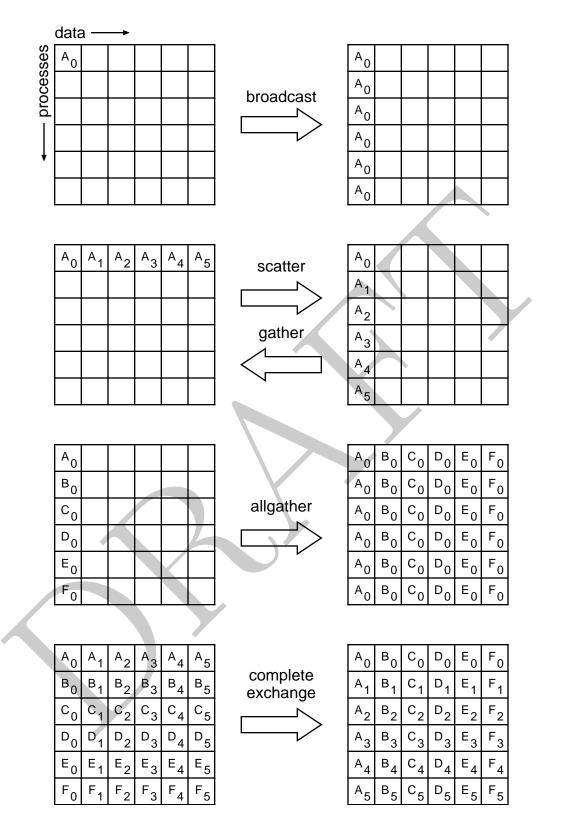


Figure 6.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data  $A_0$ , but after the broadcast all processes contain it.

#### Unofficial Draft for Comment Only

 $^{31}$ 

 $\frac{44}{45}$ 

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 6.14. (*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 6.14. (End of advice to implementors.)

<sup>13</sup> Many of the descriptions of the collective routines provide illustrations in terms of <sup>14</sup> blocking MPI point-to-point routines. These are intended solely to indicate what data is <sup>15</sup> sent or received by what process. Many of these examples are *not* correct MPI programs; <sup>16</sup> for purposes of simplicity, they often assume infinite buffering.

17 18 19

### 6.2 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 7. For the purposes of this chapter, it is sufficient to know that there are two types of communicators: *intra-communicators* and *inter-communicators*. An intra-communicator can be thought of as an identifier for a single group of processes linked with a context. An inter-communicator identifies two distinct groups of processes linked with a context.

### <sup>28</sup> 6.2.1 Specifics for Intra-Communicator Collective Operations

All processes in the group identified by the intra-communicator must call the collective routine.

In many cases, collective communication can occur "in place" for intra-communicators, 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.

36 The "in place" operations are provided to reduce unnecessary memory Rationale. 37 motion by both the MPI implementation and by the user. Note that while the simple 38 check of testing whether the send and receive buffers have the same address will 39 work for some cases (e.g., MPI\_ALLREDUCE), they are inadequate in others (e.g., 40 MPI\_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits 41 aliasing of arguments; the approach of using a special value to denote "in place" 42operation eliminates that difficulty. (End of rationale.) 43

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.

<sup>47</sup> Note that MPI\_IN\_PLACE is a special kind of value; it has the same restrictions on its <sup>48</sup> use that MPI\_BOTTOM has. (*End of advice to users.*)

#### Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

6.2.2 Applying Collective Operations to Inter-Communicators	1
To understand how collective operations apply to inter-communicators, we can view most MPI intra-communicator collective operations as fitting one of the following categories (see,	2 3
for instance, [63]):	4 5
All-To-All All processes contribute to the result. All processes receive the result.	6
<ul> <li>MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV</li> </ul>	7 8 9
<ul> <li>MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW</li> </ul>	10 11
<ul> <li>MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER</li> </ul>	12 13 14
MPI_BARRIER, MPI_IBARRIER	15 16
All-To-One All processes contribute to the result. One process receives the result.	17
MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV	18 19
<ul> <li>MPI_REDUCE, MPI_IREDUCE</li> </ul>	20
<b>One-To-All</b> One process contributes to the result. All processes receive the result.	21 22
MPI_BCAST, MPI_IBCAST	23
<ul> <li>MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV</li> </ul>	24 25
<b>Other</b> Collective operations that do not fit into one of the above categories.	26
<ul> <li>MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN</li> </ul>	27 28
The data movement patterns of MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, and	29 30
MPI_IEXSCAN do not fit this taxonomy. The application of collective communication to inter-communicators is best described	31
in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be	32 33
described as collecting data from all members of one group with the result appearing in all	34
members of the other group (see Figure 6.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the	35
other group. Collective computation operations such as MPI_REDUCE_SCATTER have a	$\frac{36}{37}$
similar interpretation (see Figure 6.3). For intra-communicators, these two groups are the	38
same. For inter-communicators, these two groups are distinct. For the all-to-all operations,	39
each such operation is described in two phases, so that it has a symmetric, full-duplex behavior.	40
The following collective operations also apply to inter-communicators:	41 42
MPI_BARRIER, MPI_IBARRIER	43
MPI_BCAST, MPI_IBCAST	$44 \\ 45$
<ul> <li>MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV,</li> </ul>	46
	47 48
<ul> <li>MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV,</li> </ul>	20

**Unofficial Draft for Comment Only** 

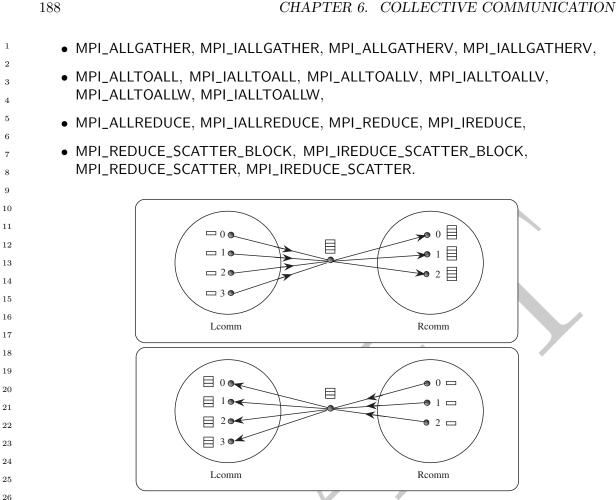


Figure 6.2: Inter-communicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

29 30  $^{31}$ 

32

27

28

#### Specifics for Inter-Communicator Collective Operations 6.2.3

All processes in both groups identified by the inter-communicator must call the collective 33 routine.

34 Note that the "in place" option for intra-communicators does not apply to inter-35 communicators since in the inter-communicator case there is no communication from a 36 process to itself. 37

For inter-communicator collective communication, if the operation is in the All-To-One 38 or One-To-All categories, then the transfer is unidirectional. The direction of the transfer is 39 indicated by a special value of the root argument. In this case, for the group containing the 40 root process, all processes in the group must call the routine using a special argument for 41 the root. For this, the root process uses the special root value MPI\_ROOT; all other processes 42in the same group as the root use MPI\_PROC\_NULL. All processes in the other group (the 43 group that is the remote group relative to the root process) must call the collective routine 44 and provide the rank of the root. If the operation is in the All-To-All category, then the 45 transfer is bidirectional. 46

47Rationale. Operations in the All-To-One and One-To-All categories are unidirectional 48 by nature, and there is a clear way of specifying direction. Operations in the All-To-All

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

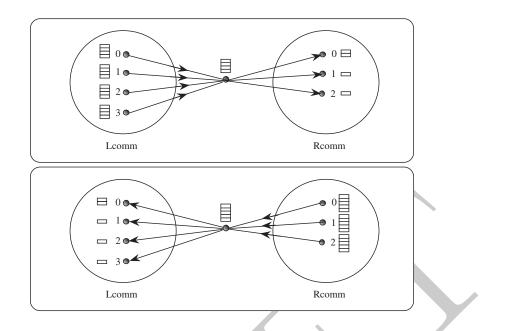


Figure 6.3: Inter-communicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

category will often occur as part of an exchange, where it makes sense to communicate in both directions at once. (*End of rationale.*)

#### 6.3 Barrier Synchronization

MPI_BARRIER(comm)	
IN comm co	mmunicator (handle)
C binding	
<pre>int MPI_Barrier(MPI_Comm comm)</pre>	
Fortran 2008 binding	
MPI_Barrier(comm, ierror)	
TYPE(MPI_Comm), INTENT(IN) :: com	ım
INTEGER, OPTIONAL, INTENT(OUT) ::	ierror
Fortran binding	
MPI_BARRIER(COMM, IERROR)	
INTEGER COMM, IERROR	

If comm is an intra-communicator, 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.

If comm is an inter-communicator, MPI\_BARRIER involves two groups. The call returns at processes in one group (group A) of the inter-communicator only after all members of 

#### Unofficial Draft for Comment Only

1 the other group (group B) have entered the call (and vice versa). A process may return  $\mathbf{2}$ from the call before all processes in its own group have entered the call. 3 4 Broadcast 6.4 56 7 8 MPI\_BCAST(buffer, count, datatype, root, comm) 9 INOUT buffer starting address of buffer (choice) 10 IN number of entries in buffer (non-negative integer) count 11 12IN datatype data type of buffer (handle) 13 IN root rank of broadcast root (integer) 14communicator (handle) IN comm 151617C binding int MPI\_Bcast(void \*buffer, int count, MPI\_Datatype datatype, int root, 18 19MPI\_Comm comm) 20int MPI\_Bcast\_c(void \*buffer, MPI\_Count count, MPI\_Datatype datatype, 21int root, MPI\_Comm comm) 2223Fortran 2008 binding 24MPI\_Bcast(buffer, count, datatype, root, comm, ierror) 25TYPE(\*), DIMENSION(..) :: buffer 26INTEGER, INTENT(IN) :: count, root TYPE(MPI\_Datatype), INTENT(IN) :: datatype 27TYPE(MPI\_Comm), INTENT(IN) :: comm 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2930 MPI\_Bcast(buffer, count, datatype, root, comm, ierror)  $^{31}$ TYPE(\*), DIMENSION(..) :: buffer 32 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 33 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 34 INTEGER, INTENT(IN) :: root 35TYPE(MPI\_Comm), INTENT(IN) :: comm 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 38 Fortran binding 39 MPI\_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) 40<type> BUFFER(\*) 41 INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

If comm is an intra-communicator, 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.

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

6.4.1 Example using MPI\_BCAST

The examples in this section use intra-communicators.

Example 6.1 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.

#### 6.5 Gather

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

IN	sendbuf	starting address of send buffer (choice)	33
IN	sendcount	number of elements in send buffer (non-negative	34
		integer)	35
IN	sendtype	data type of send buffer elements (handle)	36
			37
OUT	recvbuf	address of receive buffer (choice, significant only at	38
		root)	39
IN	recvcount	number of elements for any single receive	40
		(non-negative integer, significant only at root)	41
IN	recvtype	data type of recv buffer elements (handle, significant	42
		only at root)	43
		• /	44
IN	root	rank of receiving process (integer)	45
IN	comm	communicator (handle)	46
			47

#### C binding

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

32

```
1
     int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
\mathbf{2}
                    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
3
                    MPI_Comm comm)
4
     int MPI_Gather_c(const void *sendbuf, MPI_Count sendcount,
5
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
6
                    MPI_Datatype recvtype, int root, MPI_Comm comm)
7
8
     Fortran 2008 binding
9
     MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
10
                    root, comm, ierror)
11
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
12
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
13
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
14
         TYPE(*), DIMENSION(..) :: recvbuf
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
18
                    root, comm, ierror)
19
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
20
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
21
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
22
         TYPE(*), DIMENSION(..) :: recvbuf
23
         INTEGER, INTENT(IN) :: root
24
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     Fortran binding
28
     MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
29
                    ROOT, COMM, IERROR)
30
          <type> SENDBUF(*), RECVBUF(*)
31
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
32
         If comm is an intra-communicator, each process (root process included) sends the con-
33
     tents of its send buffer to the root process. The root process receives the messages and stores
34
     them in rank order. The outcome is as if each of the n processes in the group (including
35
     the root process) had executed a call to
36
37
        MPI_Send(sendbuf, sendcount, sendtype, root, ...),
38
39
     and the root had executed n calls to
40
41
        MPI_Recv(recvbuf+i· recvcount· extent(recvtype), recvcount, recvtype, i,...),
42
43
     where extent(recvtype) is the type extent obtained from a call to MPI_Type_get_extent.
         An alternative description is that the n messages sent by the processes in the group
44
45
     are concatenated in rank order, and the resulting message is received by the root as if by a
46
     call to MPI_RECV(recvbuf, recvcount \cdot n, recvtype, ...).
47
         The receive buffer is ignored for all non-root processes.
48
```

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

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

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

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

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

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

 $\mathbf{2}$ 

 $\overline{7}$ 

12	MPI_GATH	IERV(sendbuf, sendcount, send comm)	ltype, recvbuf, recvcounts, displs, recvtype, root,
3 4	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcount	number of elements in send buffer (non-negative integer)
7	IN	sendtype	data type of send buffer elements (handle)
8 9 10	OUT	recvbuf	address of receive buffer (choice, significant only at root)
11 12 13	IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
14 15 16 17 18	IN	displs	integer array (of length group size). Entry i specifies the displacement relative to <b>recvbuf</b> at which to place the incoming data from process i (significant only at root)
19 20	IN	recvtype	data type of recv buffer elements (handle, significant only at root)
21	IN	root	rank of receiving process (integer)
22	IN	comm	communicator (handle)
23 24 25 26 27 28 29 30 31 32 33 34	int MPI_G Fortran 2	atherv(const void *sendbu void *recvbuf, const MPI_Datatype recvtyp atherv_c(const void *send MPI_Datatype sendtyp const MPI_Count recv MPI_Datatype recvtyp 008 binding	<pre>counts[], const MPI_Aint displs[], e, int root, MPI_Comm comm)</pre>
35 36 37 38 39 40 41 42	TYPE( INTEG TYPE( TYPE( TYPE(	<pre>recvtype, root, comm *), DIMENSION(), INTENT</pre>	<pre>F(IN) :: sendbuf unt, recvcounts(*), displs(*), root ) :: sendtype, recvtype vbuf comm</pre>
43 44 45 46 47 48	TYPE( INTEG TYPE(	<pre>recvtype, root, comm *), DIMENSION(), INTENT</pre>	<pre>I(IN) :: sendbuf INTENT(IN) :: sendcount, recvcounts(*) ) :: sendtype, recvtype</pre>

```
1
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                               \mathbf{2}
    INTEGER, INTENT(IN) :: root
                                                                                               3
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                               4
                                                                                               5
Fortran binding
                                                                                               6
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                RECVTYPE, ROOT, COMM, IERROR)
     <type> SENDBUF(*), RECVBUF(*)
                                                                                               9
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
                                                                                               10
                 COMM, IERROR
                                                                                               11
    MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count
                                                                                               12
                                                                                               13
of data from each process, since recvcounts is now an array. It also allows more flexibility
                                                                                               14
as to where the data is placed on the root, by providing the new argument, displs.
                                                                                               15
    If comm is an intra-communicator, the outcome is as if each process, including the
                                                                                               16
root process, sends a message to the root,
                                                                                               17
   MPI_Send(sendbuf, sendcount, sendtype, root, ...),
                                                                                               18
                                                                                               19
and the root executes n receives,
                                                                                               20
                                                                                               21
   MPI_Recv(recvbuf+displs[j] · extent(recvtype), recvcounts[j], recvtype, i, ...).
                                                                                               22
                                                                                               23
    The data received from process j is placed into recvbuf of the root process beginning at
                                                                                               ^{24}
offset displs[j] elements (in terms of the recvtype).
                                                                                               25
    The receive buffer is ignored for all non-root processes.
                                                                                               26
    The type signature implied by sendcount, sendtype on process i must be equal to the
                                                                                               27
type signature implied by recvcounts[i], recvtype at the root. This implies that the amount
                                                                                               28
of data sent must be equal to the amount of data received, pairwise between each process
                                                                                               29
and the root. Distinct type maps between sender and receiver are still allowed, as illustrated
                                                                                               30
in Example 6.6.
                                                                                               31
    All arguments to the function are significant on process root, while on other processes,
                                                                                               32
only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments
                                                                                               33
root and comm must have identical values on all processes.
                                                                                               34
  The specification of counts, types, and displacements should not cause any location on
                                                                                               35
the root to be written more than once. Such a call is erroneous.
                                                                                               36
    The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE
                                                                                               37
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
```

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

47 48

41

42

43

44

45

```
CHAPTER 6. COLLECTIVE COMMUNICATION
```

1 2 3 4 5 6 7	100 100 100 all processes 100 100 100 at root
8	rbuf
9 10	Figure 6.4: The root process gathers 100 ints from each process in the group.
11 12 13	6.5.1 Examples using MPI_GATHER, MPI_GATHERV
14	The examples in this section use intra-communicators.
15 16	<b>Example 6.2</b> Gather 100 ints from every process in group to root. See Figure 6.4.
17	MPI_Comm comm;
18	<pre>int gsize,sendarray[100];</pre>
19 20	<pre>int root, *rbuf;</pre>
21	MPI_Comm_size(comm, &gsize);
22	rbuf = (int *)malloc(gsize*100*sizeof(int));
23	MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
24	
25	
26	<b>Example 6.3</b> Previous example modified—only the root allocates memory for the receive
27	buffer.
28	MPI_Comm comm;
29	int gsize, sendarray [100];
30	int root, myrank, *rbuf;
31	····
32 33	<pre>MPI_Comm_rank(comm, &amp;myrank);</pre>
34	if (myrank == root) {
35	MPI_Comm_size(comm, &gsize);
36	<pre>rbuf = (int *)malloc(gsize*100*sizeof(int));</pre>
37	}
38	<pre>MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);</pre>
39	
40	<b>Example 6.4</b> Do the same as the previous example, but use a derived datatype. Note
41	that the type cannot be the entire set of gsize*100 ints since type matching is defined
42	pairwise between the root and each process in the gather.
43	r
44	
45	
45 46 47	

 $^{48}$ 

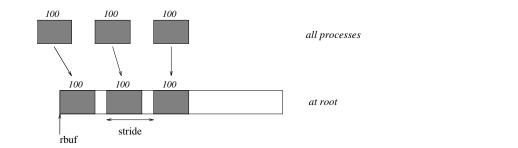


Figure 6.5: The root process gathers 100 ints from each process in the group, each set is placed stride ints apart.

```
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_commit(&rtype);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

**Example 6.5** 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 6.5.

```
27
MPI_Comm comm;
                                                                                    28
int gsize,sendarray[100];
                                                                                    29
int root, *rbuf, stride;
                                                                                    30
int *displs,i,*rcounts;
                                                                                    31
                                                                                    32
                                                                                    33
                                                                                    34
MPI_Comm_size(comm, &gsize);
                                                                                    35
rbuf = (int *)malloc(gsize*stride*sizeof(int));
                                                                                    36
displs = (int *)malloc(gsize*sizeof(int));
                                                                                    37
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                    38
for (i=0; i<gsize; ++i) {</pre>
                                                                                    39
    displs[i] = i*stride;
                                                                                    40
    rcounts[i] = 100;
                                                                                    41
}
                                                                                    42
MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                                                    43
             root, comm);
                                                                                    44
                                                                                    45
Note that the program is erroneous if stride < 100.
                                                                                    46
```

**Example 6.6** Same as Example 6.5 on the receiving side, but send the 100 ints from the 0th column of a  $100 \times 150$  int array, in C. See Figure 6.6.

#### Unofficial Draft for Comment Only

1

2

5

6

9

10

11 12

13

14

15

16

17

18

19

20

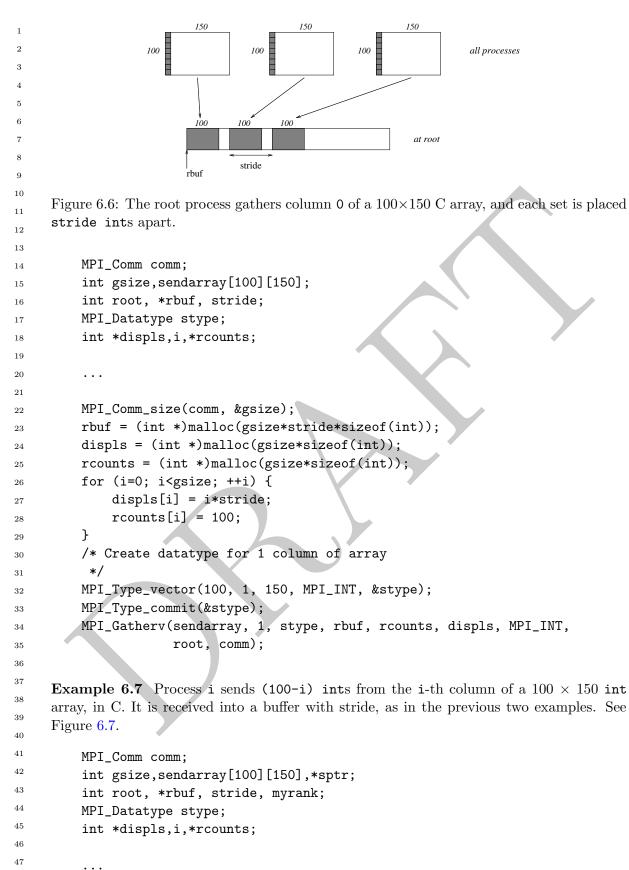
21

22 23

 $^{24}$ 

25 26

47



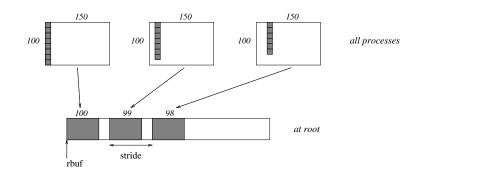


Figure 6.7: The root process gathers 100-i ints from column i of a  $100 \times 150$  C array, and each set is placed stride ints apart.

```
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = i*stride;
                             /* note change from previous example */
    rcounts[i] = 100-i;
}
/* Create datatype for the column we are sending
 */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
/* sptr is the address of start of "myrank" column
 */
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
            root, comm);
```

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

**Example 6.8** Same as Example 6.7, but done in a different way at the sending end. We create a datatype that causes the correct striding at the sending end so that we read a column of a C array. A similar thing was done in Example 5.16, Section 5.1.14.

```
MPI_Comm comm;
                                                                                  38
                                                                                  39
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, stride, myrank;
                                                                                   40
                                                                                   41
MPI_Datatype stype;
                                                                                   42
int *displs, i, *rcounts;
                                                                                   43
                                                                                   44
. . .
                                                                                   45
                                                                                   46
MPI_Comm_size(comm, &gsize);
                                                                                   47
MPI_Comm_rank(comm, &myrank);
                                                                                   48
rbuf = (int *)malloc(gsize*stride*sizeof(int));
```

1 2

10

11

12 13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31 32

33 34

35

36

```
1
         displs = (int *)malloc(gsize*sizeof(int));
\mathbf{2}
         rcounts = (int *)malloc(gsize*sizeof(int));
3
         for (i=0; i<gsize; ++i) {</pre>
4
              displs[i] = i*stride;
5
              rcounts[i] = 100-i;
6
         }
7
          /* Create datatype for one int, with extent of entire row
8
           */
9
         MPI_Type_create_resized(MPI_INT, 0, 150*sizeof(int), &stype);
10
         MPI_Type_commit(&stype);
11
         sptr = &sendarray[0][myrank];
12
         MPI_Gatherv(sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
13
                       root, comm);
14
15
     Example 6.9 Same as Example 6.7 at sending side, but at receiving side we make the
16
     stride between received blocks vary from block to block. See Figure 6.8.
17
18
         MPI_Comm comm;
19
         int gsize, sendarray[100][150], *sptr;
20
         int root, *rbuf, *stride, myrank, bufsize;
21
         MPI_Datatype stype;
22
         int *displs,i,*rcounts,offset;
23
24
          . . .
25
26
         MPI_Comm_size(comm, &gsize);
27
         MPI_Comm_rank(comm, &myrank);
28
29
         stride = (int *)malloc(gsize*sizeof(int));
30
          . . .
31
          /* stride[i] for i = 0 to gsize-1 is set somehow
32
           */
33
34
         /* set up displs and rcounts vectors first
35
           */
36
         displs = (int *)malloc(gsize*sizeof(int));
37
         rcounts = (int *)malloc(gsize*sizeof(int));
38
         offset = 0;
39
         for (i=0; i<gsize; ++i) {</pre>
40
              displs[i] = offset;
41
              offset += stride[i];
42
              rcounts[i] = 100-i;
43
         }
44
         /* the required buffer size for rbuf is now easily obtained
45
           */
46
         bufsize = displs[gsize-1]+rcounts[gsize-1];
47
         rbuf = (int *)malloc(bufsize*sizeof(int));
48
```

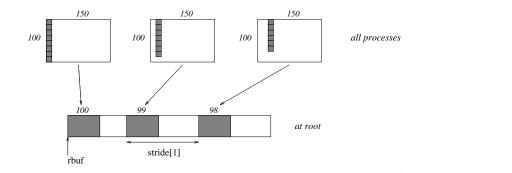


Figure 6.8: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride[i] ints apart (a varying stride).

**Example 6.10** Process i sends num ints from the i-th column of a  $100 \times 150$  int array, in C. The complicating factor is that the various values of num are not known to root, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

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

 $^{24}$ 

```
1
          }
2
          /* And, create receive buffer
3
           */
4
          rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
5
                                                                           *sizeof(int));
6
          /* Create datatype for one int, with extent of entire row
7
           */
8
          MPI_Type_create_resized(MPI_INT, 0, 150*sizeof(int), &stype);
9
          MPI_Type_commit(&stype);
10
          sptr = &sendarray[0][myrank];
11
          MPI_Gatherv(sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
12
                        root, comm);
13
14
          Scatter
     6.6
15
16
17
     MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
18
19
       IN
                 sendbuf
                                             address of send buffer (choice, significant only at
20
                                             root)
21
       IN
                 sendcount
                                             number of elements sent to each process
22
                                             (non-negative integer, significant only at root)
23
24
       IN
                                             data type of send buffer elements (handle, significant
                 sendtype
25
                                             only at root)
26
       OUT
                                             address of receive buffer (choice)
                 recvbuf
27
       IN
                 recvcount
                                             number of elements in receive buffer (non-negative
28
                                             integer)
29
30
       IN
                                             data type of receive buffer elements (handle)
                 recvtype
31
       IN
                                             rank of sending process (integer)
                 root
32
       IN
                                             communicator (handle)
                 comm
33
34
     C binding
35
     int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
36
37
                    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                    MPI_Comm comm)
38
39
     int MPI_Scatter_c(const void *sendbuf, MPI_Count sendcount,
40
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
41
                    MPI_Datatype recvtype, int root, MPI_Comm comm)
42
     Fortran 2008 binding
43
44
     MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
45
                    root, comm, ierror)
46
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
47
          INTEGER, INTENT(IN) :: sendcount, recvcount, root
48
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(*), DIMENSION(...) :: recvbuf
                                                                                            2
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
               root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                            10
    INTEGER, INTENT(IN) :: root
                                                                                            11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                            12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                            13
                                                                                            14
Fortran binding
                                                                                            15
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
                                                                                            16
               ROOT, COMM, IERROR)
                                                                                            17
     <type> SENDBUF(*), RECVBUF(*)
     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
                                                                                            18
                                                                                            19
    MPI_SCATTER is the inverse operation to MPI_GATHER.
                                                                                           20
    If comm is an intra-communicator, the outcome is as if the root executed n send
                                                                                           21
operations,
                                                                                           22
                                                                                           23
   MPI_Send(sendbuf+i· sendcount· extent(sendtype), sendcount, sendtype, i,...),
                                                                                            24
                                                                                            25
and each process executed a receive,
                                                                                            26
                                                                                           27
   MPI_Recv(recvbuf, recvcount, recvtype, i,...).
                                                                                           28
    An alternative description is that the root sends a message with MPI_Send(sendbuf,
                                                                                           29
sendcount \cdot n, sendtype, ...). This message is split into n equal segments, the i-th segment is
                                                                                           30
sent to the i-th process in the group, and each process receives this message as above.
                                                                                           31
    The send buffer is ignored for all non-root processes.
                                                                                           32
    The type signature associated with sendcount, sendtype at the root must be equal to
                                                                                           33
the type signature associated with recvcount, recvtype at all processes (however, the type
                                                                                           34
maps may be different). This implies that the amount of data sent must be equal to the
                                                                                           35
amount of data received, pairwise between each process and the root. Distinct type maps
                                                                                           36
between sender and receiver are still allowed.
                                                                                           37
    All arguments to the function are significant on process root, while on other processes,
                                                                                           38
only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments
                                                                                           39
root and comm must have identical values on all processes.
                                                                                            40
                                                                                           41
    The specification of counts and types should not cause any location on the root to be
read more than once.
                                                                                           42
                                                                                           43
     Rationale.
                   Though not needed, the last restriction is imposed so as to achieve
                                                                                           44
     symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write
                                                                                            45
     restriction) is necessary. (End of rationale.)
                                                                                            46
                                                                                            47
    The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE
                                                                                            48
as the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and
```

#### Unofficial Draft for Comment Only

<sup>1</sup> root "sends" no data to itself. The scattered vector is still assumed to contain n segments, <sup>2</sup> where n is the group size; the *root*-th segment, which root should "send to itself," is not <sup>3</sup> moved.

<sup>4</sup> If comm is an inter-communicator, then the call involves all processes in the inter-<sup>5</sup> communicator, but with one group (group A) defining the root process. All processes in <sup>6</sup> the other group (group B) pass the same value in argument root, which is the rank of the <sup>7</sup> root in group A. The root passes the value MPI\_ROOT in root. All other processes in group <sup>8</sup> A pass the value MPI\_PROC\_NULL in root. Data is scattered from the root to all processes <sup>9</sup> in group B. The receive buffer arguments of the processes in group B must be consistent <sup>10</sup> with the send buffer argument of the root.

11 12

<sup>13</sup> MPI\_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, <sup>14</sup> comm)

15 16	IN	sendbuf	address of send buffer (choice, significant only at root)
17 18 19	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)
20 21 22 23 24	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 (significant only at root)
25 26	IN	sendtype	data type of send buffer elements (handle, significant only at root)
27 28	OUT	recvbuf	address of receive buffer (choice)
29 30	IN	recvcount	number of elements in receive buffer (non-negative integer)
31	IN	recvtype	data type of receive buffer elements (handle)
32 33	IN	root	rank of sending process (integer)
34	IN	comm	communicator (handle)
35			
36	C binding		a
37 38 39	int MPI_S	<pre>const int displs[],</pre>	ouf, const int sendcounts[], MPI_Datatype sendtype, void *recvbuf, atatype recvtype, int root, MPI_Comm comm)
40 41 42 43 44	int MPI_S	const MPI_Aint displ	ndbuf, const MPI_Count sendcounts[], s[], MPI_Datatype sendtype, void *recvbuf, MPI_Datatype recvtype, int root,
45 46 47 48	MPI_Scatt	2008 binding erv(sendbuf, sendcounts, recvtype, root, comm *), DIMENSION(), INTEN	

```
INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
                                                                                          1
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                          2
    TYPE(*), DIMENSION(..) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
               recvtype, root, comm, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcount
                                                                                          10
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                          11
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                          12
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                          13
    INTEGER, INTENT(IN) :: root
                                                                                          14
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                          15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                          16
                                                                                          17
Fortran binding
                                                                                          18
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                          19
               RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
                                                                                          20
                                                                                          21
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
                                                                                          22
                COMM, IERROR
                                                                                          23
    MPI_SCATTERV is the inverse operation to MPI_GATHERV.
                                                                                          24
    MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying
                                                                                          25
count of data to be sent to each process, since sendcounts is now an array. It also allows
                                                                                          26
more flexibility as to where the data is taken from on the root, by providing an additional
                                                                                          27
argument, displs.
                                                                                          28
    If comm is an intra-communicator, the outcome is as if the root executed n send oper-
                                                                                          29
ations,
                                                                                          30
                                                                                          31
   MPI_Send(sendbuf+displs[i] extent(sendtype), sendcounts[i], sendtype, i,...),
                                                                                          32
                                                                                          33
and each process executed a receive,
                                                                                          34
                                                                                          35
   MPI_Recv(recvbuf, recvcount, recvtype, i,...).
                                                                                          36
    The send buffer is ignored for all non-root processes.
                                                                                          37
    The type signature implied by sendcount[i], sendtype at the root must be equal to the
                                                                                          38
type signature implied by recvcount, recvtype at process i (however, the type maps may be
                                                                                          39
different). This implies that the amount of data sent must be equal to the amount of data
                                                                                          40
                                                                                          41
received, pairwise between each process and the root. Distinct type maps between sender
and receiver are still allowed.
                                                                                          42
    All arguments to the function are significant on process root, while on other processes,
                                                                                          43
only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments
                                                                                          44
root and comm must have identical values on all processes.
                                                                                          45
    The specification of counts, types, and displacements should not cause any location on
                                                                                          46
the root to be read more than once.
                                                                                          47
                                                                                          48
```

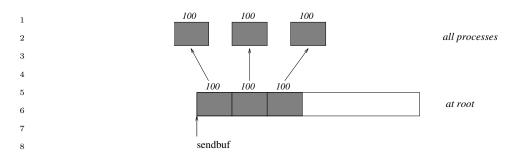


Figure 6.9: The root process scatters sets of 100 ints to each process in the group.

The "in place" option for intra-communicators 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.

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

<sup>25</sup> 6.6.1 Examples using MPI\_SCATTER, MPI\_SCATTERV

 $_{27}$  The examples in this section use intra-communicators.

Example 6.11 The reverse of Example 6.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 6.9.

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

<sup>41</sup> **Example 6.12** The reverse of Example 6.5. The root process scatters sets of 100 ints to <sup>42</sup> the other processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires <sup>43</sup> use of MPI\_SCATTERV. Assume *stride*  $\geq$  100. See Figure 6.10.

206

9

10 11

12

13

14

15 16

 $^{24}$ 

26

 $^{31}$ 

32

33

34

35

36

37 38

39 40

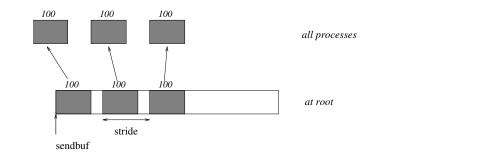


Figure 6.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

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

**Example 6.13** The reverse of Example 6.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the *i*-th column of a  $100 \times 150$  C array. See Figure 6.11.

```
MPI_Comm comm;
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 * sendbuf comes from elsewhere
 */
```

1

 $\mathbf{2}$ 

5

6

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

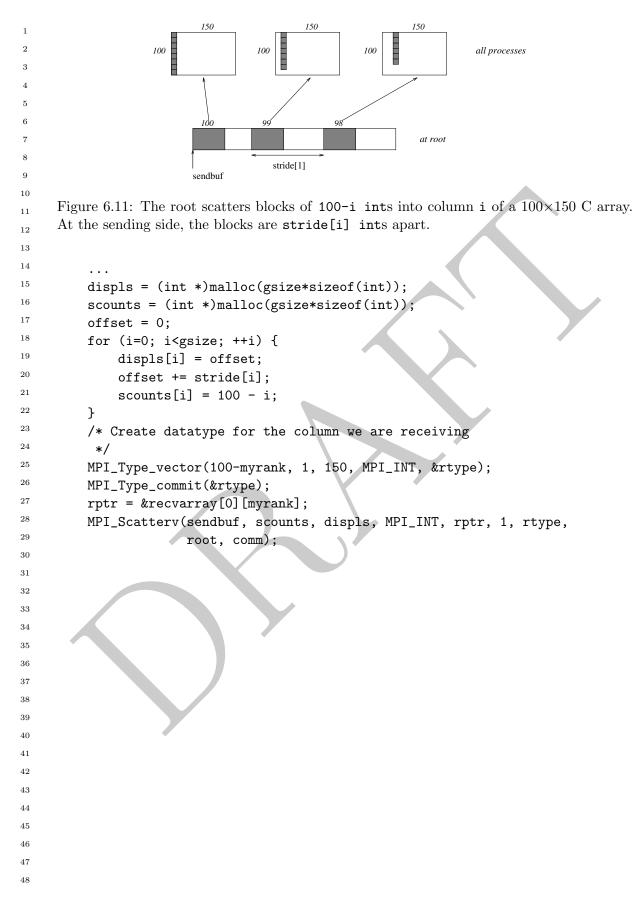
42

43 44

45

46

47



Gather-to-all MPI\_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm) sendbuf IN 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) 10 OUT recvbuf address of receive buffer (choice) 11 number of elements received from any process 12IN recvcount 13 (non-negative integer) 14IN recvtype data type of receive buffer elements (handle) 15IN communicator (handle) comm 1617 C binding 18 int MPI\_Allgather(const void \*sendbuf, int sendcount, 19 MPI\_Datatype sendtype, void \*recvbuf, int recvcount, 20MPI\_Datatype recvtype, MPI\_Comm comm) 2122 int MPI\_Allgather\_c(const void \*sendbuf, MPI\_Count sendcount, 23MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 24MPI\_Datatype recvtype, MPI\_Comm comm) 2526Fortran 2008 binding MPI\_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 27comm, ierror) 28TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf 29 INTEGER, INTENT(IN) :: sendcount, recvcount 30 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 31TYPE(\*), DIMENSION(..) :: recvbuf 32 TYPE(MPI\_Comm), INTENT(IN) :: comm 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35MPI\_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 36 comm, ierror) 37 TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 38 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 39 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 40 TYPE(\*), DIMENSION(..) :: recvbuf 41 TYPE(MPI\_Comm), INTENT(IN) :: comm 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 Fortran binding 44 MPI\_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 4546

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

**Unofficial Draft for Comment Only** 

1 2

5

6

9

47

1 2 3 4 5 6 7 8	MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. If comm is an intra-communicator, the outcome of a call to MPI_ALLGATHER() is as if all processes executed n calls to
9 10	<pre>MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount, recvtype,root,comm)</pre>
11 12 13 14 15 16 17 18 19 20 21 22	for root = 0,, n-1. The rules for correct usage of MPI_ALLGATHER are easily found from the corresponding rules for MPI_GATHER. The "in place" option for intra-communicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer. If comm is an inter-communicator, then each process of one group (group A) contributes sendcount data items; these data are concatenated and the result is stored at each process in the other group (group B). Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.
23 24 25 26 27 28 29 30 31	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.)
32 33	
34 35	
36	
37 38	
39	
40	
41 42	
42 43	
44	
45	
46	
47	
48	

Unofficial Draft for Comment Only

MPI_ALLO	GATHERV(sendbuf, sendcount comm)	, sendtype, recvbuf, recvcounts, displs, recvtype,	1 $2$
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements in send buffer (non-negative integer)	4 5 6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	address of receive buffer (choice)	8
IN	recvcounts	non-negative integer array (of length group size)	9
IIN		containing the number of elements that are received	10 11
		from each process	12
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to <b>recvbuf</b> ) at which to	13 14
		place the incoming data from process i	15
IN	recvtype	data type of receive buffer elements (handle)	16 17
IN	comm	communicator (handle)	18
			19
C binding	g		20
int MPI_A	Allgatherv(const void *se		21
		<pre>pe, void *recvbuf, const int recvcounts[],</pre>	22
	const int dispis[],	MPI_Datatype recvtype, MPI_Comm comm)	23 24
int MPI_A	-	sendbuf, MPI_Count sendcount,	25
	MPI_Datatype sendty		26
	const MPI_Count rec MPI_Datatype recvty	vcounts[], const MPI_Aint displs[],	27
		pe, MFI_Comm comm)	28
	2008 binding		29
MPI_Allga		, sendtype, recvbuf, recvcounts, displs,	30 31
TVDF	recvtype, comm, ier: (*), DIMENSION(), INTEN		32
		<pre>punt, recvcounts(*), displs(*)</pre>	33
	(MPI_Datatype), INTENT(IN	-	34
	(*), DIMENSION() :: rec		35
	(MPI_Comm), INTENT(IN) ::		36
INTEC	GER, OPTIONAL, INTENT(OUT	I) :: ierror	37
MPI_Allga		, sendtype, recvbuf, recvcounts, displs,	38 39
TYPF	recvtype, comm, ier: (*), DIMENSION(), INTEN		40
		, INTENT(IN) :: sendcount, recvcounts(*)	41
	(MPI_Datatype), INTENT(IN		42 43
	(*), DIMENSION() :: rec	VI VI	43 44
		), INTENT(IN) :: displs(*)	45
	(MPI_Comm), INTENT(IN) ::		46
INTEC	GER, OPTIONAL, INTENT(OUT	I) :: lerror	47
Fortran binding 48			

1	MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
2	RECVTYPE, COMM, IERROR)
3	<type> SENDBUF(*), RECVBUF(*)</type>
4	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
5	IERROR
6	
7	MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes re-
8	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 need not all be the same size.
10	The type signature associated with sendcount, sendtype, at process j must be equal to
11	the type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[j], recvtype at any other process.
12	If comm is an intra-communicator, the outcome is as if all processes executed calls to
13	
14	<pre>MPI_Gatherv(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,</pre>
15	recvtype,root,comm),
16 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 intra-communicators 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 inter-communicator, 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	
29	6.7.1 Example using MPI_ALLGATHER
30	The example in this section uses intra-communicators.
31 32	
33	<b>Example 6.14</b> The all-gather version of Example 6.2. Using MPI_ALLGATHER, we will
34	gather 100 ints from every process in the group to every process.
35	
36	<pre>MPI_Comm comm; int gsize,sendarray[100];</pre>
37	int *rbuf;
38	
39	MPI_Comm_size(comm, &gsize);
40	<pre>rbuf = (int *)malloc(gsize*100*sizeof(int));</pre>
41	<pre>MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);</pre>
42	
43	After the call, every process has the group-wide concatenation of the sets of data.
44 45	
45 46	
47	
48	

# 6.8 All-to-All Scatter/Gather

2 MPI\_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm) 5 sendbuf IN starting address of send buffer (choice) 6 7 IN sendcount number of elements sent to each process (non-negative integer) IN sendtype data type of send buffer elements (handle) 10 OUT recvbuf address of receive buffer (choice) 11 IN number of elements received from any process 12recvcount 13 (non-negative integer) 14data type of receive buffer elements (handle) IN recvtype 15IN communicator (handle) comm 1617C binding 18 int MPI\_Alltoall(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 19 void \*recvbuf, int recvcount, MPI\_Datatype recvtype, 20MPI\_Comm comm) 2122 int MPI\_Alltoall\_c(const void \*sendbuf, MPI\_Count sendcount, 23MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 24MPI\_Datatype recvtype, MPI\_Comm comm) 2526Fortran 2008 binding MPI\_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 27comm, ierror) 28TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf 29 INTEGER, INTENT(IN) :: sendcount, recvcount 30 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 31TYPE(\*), DIMENSION(..) :: recvbuf 32 TYPE(MPI\_Comm), INTENT(IN) :: comm 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 35MPI\_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 36 comm, ierror) 37 TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf 38 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 39 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 40 TYPE(\*), DIMENSION(..) :: recvbuf 41 TYPE(MPI\_Comm), INTENT(IN) :: comm 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44 Fortran binding MPI\_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 45COMM, IERROR) 4647<type> SENDBUF(\*), RECVBUF(\*) 48 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

1	MDI ALLTOALL is an automaion of MDI ALLCATHED to the case where each presses
2	MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process
3	sends distinct data to each of the receivers. The j-th block sent from process i is received
	by process j and is placed in the i-th block of recvbuf.
4	The type signature associated with sendcount, sendtype, at a process must be equal to
5	the type signature associated with recvcount, recvtype at any other process. This implies
6	that the amount of data sent must be equal to the amount of data received, pairwise between
7	every pair of processes. As usual, however, the type maps may be different.
8	If comm is an intra-communicator, the outcome is as if each process executed a send
9	to each process (itself included) with a call to,
10	
11	MPI_Send(sendbuf+i· sendcount· extent(sendtype),sendcount,sendtype,i,),
12	and a receive from every other process with a call to,
13	and a receive from every other process with a can to,
14 15	$MPI_Recv(recvbuf+i\cdot recvcount\cdot extent(recvtype), recvcount, recvtype, i, \ldots).$
16	All arguments on all processes are significant. The argument <b>comm</b> must have identical
17	values on all processes.
18	The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE to
19	the argument sendbuf at <i>all</i> processes. In such a case, sendcount and sendtype are ignored.
20	The data to be sent is taken from the recvbuf and replaced by the received data. Data sent
21	
22	and received must have the same type map as specified by recvcount and recvtype.
23	Rationale. For large MPI_ALLTOALL instances, allocating both send and receive
24	buffers may consume too much memory. The "in place" option effectively halves the
25	application memory consumption and is useful in situations where the data to be sent
26	will not be used by the sending process after the MPI_ALLTOALL exchange (e.g., in
27	parallel Fast Fourier Transforms). (End of rationale.)
28	
29	Advice to implementors. Users may opt to use the "in place" option in order to
30	conserve memory. Quality MPI implementations should thus strive to minimize system
31	buffering. (End of advice to implementors.)
32	
33	If comm is an inter-communicator, then the outcome is as if each process in group A
34	sends a message to each process in group B, and vice versa. The j-th send buffer of process
35	i in group A should be consistent with the i-th receive buffer of process j in group B, and
36	vice versa.
37	
38	Advice to users. When a complete exchange is executed on an intercommunication
39	domain, then the number of data items sent from processes in group A to processes
40	in group B need not equal the number of items sent in the reverse direction. In
41	particular, one can have unidirectional communication by specifying sendcount $= 0$ in
42	the reverse direction. (End of advice to users.)
43	\ <b>v</b> /
44	
45	
46	
47	
48	

	recvtype, comm)	
IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank
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
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to <b>recvbuf</b> ) at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
C bindin nt MPT	-	*sendbuf, const int sendcounts[],
		ls[], MPI_Datatype sendtype, void *recvbuf,
		ounts[], const int rdispls[],
	MPI_Datatype re	cvtype, MPI_Comm comm)
nt MPI.	_Alltoallv_c(const vo	id *sendbuf, const MPI_Count sendcounts[],
	const MPI_Aint	<pre>sdispls[], MPI_Datatype sendtype,</pre>
	<pre>void *recvbuf,</pre>	<pre>const MPI_Count recvcounts[],</pre>
	const MPI_Aint	rdispls[], MPI_Datatype recvtype,
	MPI_Comm comm)	
`ortran	2008 binding	
	<b>2008</b> binding toallv(sendbuf, sendco	ounts, sdispls, sendtype, recvbuf, recvcounts,
	toallv(sendbuf, sendc	ounts, sdispls, sendtype, recvbuf, recvcounts, pe, comm, ierror)
PI_Allt	toallv(sendbuf, sendcordispls, recvty	
PI_Allt TYPE	toallv(sendbuf, sendco rdispls, recvty E(*), DIMENSION(),	pe, comm, ierror)
PI_Allt TYPE	toallv(sendbuf, sendco rdispls, recvty E(*), DIMENSION(),	pe, comm, ierror) INTENT(IN) :: sendbuf
PI_Allt TYPI INTI	toallv(sendbuf, sendco rdispls, recvty E(*), DIMENSION(), E EGER, INTENT(IN) :: so rdispls(*)	pe, comm, ierror) INTENT(IN) :: sendbuf
PI_Allt TYPI INTI TYPI TYPI	<pre>toallv(sendbuf, sendco rdispls, recvty E(*), DIMENSION(), T EGER, INTENT(IN) :: so rdispls(*) E(MPI_Datatype), INTE E(*), DIMENSION() :</pre>	<pre>pe, comm, ierror) INTENT(IN) :: sendbuf endcounts(*), sdispls(*), recvcounts(*), NT(IN) :: sendtype, recvtype : recvbuf</pre>
PI_Allt TYPF INTF TYPF TYPF TYPF	<pre>toallv(sendbuf, sendcy rdispls, recvty E(*), DIMENSION(), T EGER, INTENT(IN) :: so rdispls(*) E(MPI_Datatype), INTEN E(*), DIMENSION() : E(MPI_Comm), INTENT(INTENT)</pre>	<pre>pe, comm, ierror) INTENT(IN) :: sendbuf endcounts(*), sdispls(*), recvcounts(*), NT(IN) :: sendtype, recvtype : recvbuf N) :: comm</pre>
IPI_Allt TYPF INTF TYPF TYPF TYPF	<pre>toallv(sendbuf, sendco rdispls, recvty E(*), DIMENSION(), T EGER, INTENT(IN) :: so rdispls(*) E(MPI_Datatype), INTE E(*), DIMENSION() :</pre>	<pre>pe, comm, ierror) INTENT(IN) :: sendbuf endcounts(*), sdispls(*), recvcounts(*), NT(IN) :: sendtype, recvtype : recvbuf N) :: comm</pre>
IPI_Allt TYPF INTF TYPF TYPF TYPF INTF	<pre>toallv(sendbuf, sendcy rdispls, recvty E(*), DIMENSION(), T EGER, INTENT(IN) :: so rdispls(*) E(MPI_Datatype), INTEN E(*), DIMENSION() : E(MPI_Comm), INTENT(IN EGER, OPTIONAL, INTENT</pre>	<pre>pe, comm, ierror) INTENT(IN) :: sendbuf endcounts(*), sdispls(*), recvcounts(*), NT(IN) :: sendtype, recvtype : recvbuf N) :: comm</pre>

rdispls, recvtype, comm, ierror)
TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf

```
1
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
2
                      recvcounts(*)
3
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4
5
          TYPE(*), DIMENSION(...) :: recvbuf
6
          TYPE(MPI_Comm), INTENT(IN) :: comm
7
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     Fortran binding
9
     MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
10
                     RDISPLS, RECVTYPE, COMM, IERROR)
11
          <type> SENDBUF(*), RECVBUF(*)
12
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
13
                      RECVTYPE, COMM, IERROR
14
15
          MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for
16
      the send is specified by sdispls and the location of the placement of the data on the receive
17
     side is specified by rdispls.
18
          If comm is an intra-communicator, then the j-th block sent from process i is received
19
      by process j and is placed in the i-th block of recvbuf. These blocks need not all have the
20
      same size.
21
          The type signature associated with sendcounts[j], sendtype at process i must be equal
22
      to the type signature associated with recvcounts[i], recvtype at process j. This implies that
23
      the amount of data sent must be equal to the amount of data received, pairwise between
^{24}
      every pair of processes. Distinct type maps between sender and receiver are still allowed.
25
          The outcome is as if each process sent a message to every other process with,
26
         MPI_Send(sendbuf+sdispls[i] extent(sendtype),sendcounts[i],sendtype,i,...),
27
28
      and received a message from every other process with a call to
29
30
         MPI_Recv(recvbuf+rdispls[i] · extent(recvtype),recvcounts[i],recvtype,i,...).
^{31}
32
          All arguments on all processes are significant. The argument comm must have identical
33
     values on all processes.
34
        The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE to
35
      the argument sendbuf at all processes. In such a case, sendcounts, sdispls and sendtype are
36
      ignored. The data to be sent is taken from the recvbuf and replaced by the received data.
37
      Data sent and received must have the same type map as specified by the recvcounts array
38
      and the recvtype, and is taken from the locations of the receive buffer specified by rdispls.
39
40
                                Specifying the "in place" option (which must be given on all
           Advice to users.
41
           processes) implies that the same amount and type of data is sent and received between
42
           any two processes in the group of the communicator. Different pairs of processes can
43
           exchange different amounts of data. Users must ensure that recvcounts[j] and recvtype
44
           on process i match recvcounts[i] and recvtype on process j. This symmetric exchange
45
           can be useful in applications where the data to be sent will not be used by the sending
46
           process after the MPI_ALLTOALLV exchange. (End of advice to users.)
47
48
```

If comm is an inter-communicator, 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.

The definitions of MPI\_ALLTOALL and MPI\_ALLTOALLV give as much Rationale. 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.)

Although the discussion of collective communication in Advice to implementors. 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.)

WPI_ALL	recvtypes, comm	n)	19
IN	sendbuf	starting address of send buffer (choice)	20 21
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each	22 23
IN	sdispls	rank 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)	24 25 26 27 28
IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)	29 30 31 32
OUT	recvbuf	address of receive buffer (choice)	33
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank	34 35 36 37
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)	38 39 40 41
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)	42 43 44
IN	comm	communicator (handle)	45 46 47

# MPI\_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls,

# C binding

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9 10

11

12

13

14

```
1
     int MPI_Alltoallw(const void *sendbuf, const int sendcounts[],
\mathbf{2}
                    const int sdispls[], const MPI_Datatype sendtypes[],
3
                   void *recvbuf, const int recvcounts[], const int rdispls[],
4
                    const MPI_Datatype recvtypes[], MPI_Comm comm)
5
     int MPI_Alltoallw_c(const void *sendbuf, const MPI_Count sendcounts[],
6
                    const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
7
                   void *recvbuf, const MPI_Count recvcounts[],
8
                    const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
9
                   MPI_Comm comm)
10
11
     Fortran 2008 binding
12
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
13
                   rdispls, recvtypes, comm, ierror)
14
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
15
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
16
                    rdispls(*)
17
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
18
         TYPE(*), DIMENSION(..) :: recvbuf
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
22
                   rdispls, recvtypes, comm, ierror)
23
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
24
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
25
                    recvcounts(*)
26
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
27
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
28
         TYPE(*), DIMENSION(..) :: recvbuf
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     Fortran binding
33
     MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
34
                   RDISPLS, RECVTYPES, COMM, IERROR)
35
         <type> SENDBUF(*), RECVBUF(*)
36
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
37
                    RDISPLS(*), RECVTYPES(*), COMM, IERROR
38
         MPI_ALLTOALLW is the most general form of complete exchange. Like
39
     MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW al-
40
     lows separate specification of count, displacement and datatype. In addition, to allow max-
41
     imum flexibility, the displacement of blocks within the send and receive buffers is specified
42
     in bytes.
43
         If comm is an intra-communicator, then the j-th block sent from process i is received
44
     by process j and is placed in the i-th block of recvbuf. These blocks need not all have the
45
     same size.
46
         The type signature associated with sendcounts[i], sendtypes[i] at process i must be equal
47
     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 intra-communicators is specified by passing MPI\_IN\_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtypes are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the received and receives arrays, and is taken from the locations of the receive buffer specified by rdispls.

If **comm** is an inter-communicator, 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.*)

# 6.9 Global Reduction Operations

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

 $\mathbf{2}$ 

 $^{24}$ 

```
220
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.9.1 Reduce
\mathbf{2}
3
4
     MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
5
                sendbuf
       IN
                                            address of send buffer (choice)
6
       OUT
                recvbuf
                                            address of receive buffer (choice, significant only at
7
8
                                            root)
9
       IN
                count
                                            number of elements in send buffer (non-negative
10
                                            integer)
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
                                            communicator (handle)
                comm
16
17
     C binding
18
     int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
19
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
20
21
     int MPI_Reduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
22
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
23
     Fortran 2008 binding
^{24}
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
25
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
26
         TYPE(*), DIMENSION(..) :: recvbuf
27
         INTEGER, INTENT(IN) :: count, root
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Op), INTENT(IN) :: op
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
34
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
35
         TYPE(*), DIMENSION(...) :: recvbuf
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         TYPE(MPI_Op), INTENT(IN) :: op
39
         INTEGER, INTENT(IN) :: root
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     Fortran binding
43
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
44
          <type> SENDBUF(*), RECVBUF(*)
45
          INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
46
47
48
```

If comm is an intra-communicator, MPI\_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers of the same length, with elements of the same type as the output buffer at the root. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is MPI\_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI\_FLOAT), then recvbuf(1) = global max(sendbuf(1)) and recvbuf(2) = global max(sendbuf(2)).

Section 6.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes to which each operation can be applied.

In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 6.9.5.

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

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

Advice to users. Some applications may not be able to ignore the non-associative nature of floating-point operations or may use user-defined operations (see Section 6.9.5) 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 6.9.2 and Section 6.9.4. 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

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

1 2

3

4

 $\mathbf{5}$ 

6 7

8 9

10

11

12

13

14 15

16

17

18

19

20

21

22

23 24

25

26

27

28

29

30 31

32

33

34

35

36

37

38

39

40 41

42

43

44

45

46

47

by such a datatype, which may contain several basic values. This is further explained in  $\mathbf{2}$ Section 6.9.5. 3

> Users should make no assumptions about how MPI\_REDUCE is Advice to users. 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 intra-communicators 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 inter-communicator, 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.

Predefined Reduction Operations 6.9.2

21The following predefined operations are supplied for MPI\_REDUCE and related functions 22 MPI\_ALLREDUCE, MPI\_REDUCE\_SCATTER\_BLOCK, MPI\_REDUCE\_SCATTER,

Meaning

23MPI\_SCAN, MPI\_EXSCAN, all nonblocking variants of those (see Section 6.12), and

 $^{24}$ MPI\_REDUCE\_LOCAL. These operations are invoked by placing the following in op.

25	
26	

 $^{31}$ 

32

33

34

35

36

37

38

39

40

4546

Name 27

28MPI\_MAX 29 30

maximum MPI\_MIN minimum MPI\_SUM  $\operatorname{sum}$ MPI\_PROD product MPI\_LAND logical and MPI\_BAND bit-wise and logical or MPI\_LOR MPI\_BOR bit-wise or MPI\_LXOR logical exclusive or (xor) MPI\_BXOR bit-wise exclusive or (xor) MPI\_MAXLOC max value and location MPI\_MINLOC min value and location

The two operations MPI\_MINLOC and MPI\_MAXLOC are discussed separately in Sec-41 tion 6.9.4. For the other predefined operations, we enumerate below the allowed combi-42nations of op and datatype arguments. First, define groups of MPI basic datatypes in the 43 following way. 44

47	C integer:	MPI_INT, MPI_LONG, MPI_SHORT,
48		MPI_UNSIGNED_SHORT, MPI_UNSIGNED,

1

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18 19

	MPI_UNSIGNED_LONG,	1
	MPI_LONG_LONG_INT,	2
	MPI_LONG_LONG (as synonym),	3
	MPI_UNSIGNED_LONG_LONG,	4
	MPI_SIGNED_CHAR,	5
	MPI_UNSIGNED_CHAR,	6
	MPI_INT8_T, MPI_INT16_T,	7
	MPI_INT32_T, MPI_INT64_T,	8
	MPI_UINT8_T, MPI_UINT16_T,	9
	MPI_UINT32_T, and MPI_UINT64_T	10
Fortran integer:	MPI_INTEGER	11
	and handles returned from	12
	MPI_TYPE_CREATE_F90_INTEGER	13
	and, if available, $MPI_{I}INTEGER1$ ,	14
	MPI_INTEGER2, MPI_INTEGER4,	15
	MPI_INTEGER8, and MPI_INTEGER16	16
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL,	10
	MPI_DOUBLE_PRECISION,	18
	MPI_LONG_DOUBLE,	19
	and handles returned from	20
	MPI_TYPE_CREATE_F90_REAL	20
	and, if available, MPI_REAL2,	21
	MPI_REAL4, MPI_REAL8, and MPI_REAL16	22
Logical:	MPI_LOGICAL, MPI_C_BOOL,	
	and MPI_CXX_BOOL	24
Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	25
	MPI_C_FLOAT_COMPLEX (as synonym),	26
	MPI_C_DOUBLE_COMPLEX,	27
	MPI_C_LONG_DOUBLE_COMPLEX,	28
	MPI_CXX_FLOAT_COMPLEX,	29
	MPI_CXX_DOUBLE_COMPLEX,	30
	MPI_CXX_LONG_DOUBLE_COMPLEX,	31
	and handles returned from	32
	MPI_TYPE_CREATE_F90_COMPLEX	33
	and, if available, MPI_DOUBLE_COMPLEX,	34
	MPI_COMPLEX4, MPI_COMPLEX8,	35
	MPI_COMPLEX16, and MPI_COMPLEX32	36
Byte:	MPI_BYTE	37
Multi-language types:	MPI_AINT, MPI_OFFSET, and MPI_COUNT	38
Now, the valid datatypes for each o	operation are specified below.	39
		40
		41
Op	Allowed Types	42
		43
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point,	44
	Multi-language types	45
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex,	46
	Multi-language types	47
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical	48

Unofficial Draft for Comment Only

```
1
       MPI_BAND, MPI_BOR, MPI_BXOR
                                             C integer, Fortran integer, Byte, Multi-language types
\mathbf{2}
          These operations together with all listed datatypes are valid in all supported program-
3
     ming languages, see also Reduce Operations on page 834 in Section 19.3.6.
4
          The following examples use intra-communicators.
5
6
     Example 6.15 A routine that computes the dot product of two vectors that are distributed
7
     across a group of processes and returns the answer at node zero.
8
9
     SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
     REAL a(m), b(m)
10
                               ! local slice of array
11
     REAL c
                               ! result (at node zero)
12
     REAL sum
13
     INTEGER m, comm, i, ierr
14
15
     ! local sum
16
     sum = 0.0
17
     DO i = 1, m
18
         sum = sum + a(i)*b(i)
19
     END DO
20
21
     ! global sum
     CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
22
23
     RETURN
^{24}
     END
25
26
     Example 6.16 A routine that computes the product of a vector and an array that are
27
     distributed across a group of processes and returns the answer at node zero.
28
     SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
29
30
                              ! local slice of array
     REAL a(m), b(m,n)
31
     REAL c(n)
                              ! result
32
     REAL sum(n)
33
     INTEGER n, comm, i, j, ierr
34
35
     ! local sum
36
     DO j=1,n
37
         sum(j) = 0.0
38
         DO i=1,m
39
            sum(j) = sum(j) + a(i)*b(i,j)
40
         END DO
41
     END DO
42
43
     ! global sum
44
     CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
45
```

```
    <sup>46</sup> ! return result at node zero (and garbage at the other nodes)
    <sup>47</sup> RETURN
```

48

END

# 6.9.3 Signed Characters and Reductions

The types MPI\_SIGNED\_CHAR and MPI\_UNSIGNED\_CHAR can be used in reduction operations. MPI\_CHAR, MPI\_WCHAR, and MPI\_CHARACTER (which represent printable characters) cannot be used in reduction operations. In a heterogeneous environment, MPI\_CHAR, MPI\_WCHAR, and MPI\_CHARACTER will be translated so as to preserve the printable character, whereas MPI\_SIGNED\_CHAR and MPI\_UNSIGNED\_CHAR will be translated so as to preserve the integer value.

Advice to users. The types MPI\_CHAR, MPI\_WCHAR, and MPI\_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI\_SIGNED\_CHAR and MPI\_UNSIGNED\_CHAR should be used in C if the integer value should be preserved. (*End of advice to users.*)

# 6.9.4 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

#### Unofficial Draft for Comment Only

1 (u, r), where  $u = \max_i u_i$  and r is the index of the first global maximum in the sequence.  $\mathbf{2}$ Thus, if each process supplies a value and its rank within the group, then a reduce operation 3 with  $op = MPI_MAXLOC$  will return the maximum value and the rank of the first process with 4 that value. Similarly, MPI\_MINLOC can be used to return a minimum and its index. More 5generally, MPI\_MINLOC computes a *lexicographic minimum*, where elements are ordered 6 according to the first component of each pair, and ties are resolved according to the second 7component. 8 The reduce operation is defined to operate on arguments that consist of a pair: value 9 and index. For both Fortran and C, types are provided to describe the pair. The potentially 10 mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, 11for Fortran, by having the MPI-provided type consist of a pair of the same type as value, 12and coercing the index to this type also. In C, the MPI-provided pair type has distinct 13 types and the index is an int. 14In order to use MPI\_MINLOC and MPI\_MAXLOC in a reduce operation, one must provide 15a datatype argument that represents a pair (value and index). MPI provides nine such 16predefined datatypes. The operations MPI\_MAXLOC and MPI\_MINLOC can be used with 17each of the following datatypes. 18 Fortran: 19 Name Description 20pair of REALs MPI\_2REAL 21MPI\_2DOUBLE\_PRECISION pair of DOUBLE PRECISION variables 22 pair of INTEGERs MPI\_2INTEGER 23242526C: 27Name Description MPI\_FLOAT\_INT float and int 28MPI\_DOUBLE\_INT double and int 29MPI\_LONG\_INT long and int 30 MPI\_2INT pair of int 31short and int MPI\_SHORT\_INT 32 MPI\_LONG\_DOUBLE\_INT long double and int 33 34 The datatype MPI\_2REAL is as if defined by the following (see Section 5.1). 35 36 MPI\_Type\_contiguous(2, MPI\_REAL, MPI\_2REAL); 37 38 Similar statements apply for MPI\_2INTEGER, MPI\_2DOUBLE\_PRECISION, and MPI\_2INT. 39 The datatype MPI\_SHORT\_INT is as if defined by the following sequence of instructions. 40 struct mystruct { 41 short val; 42int rank; 43 }; 44 type[0] = MPI\_SHORT; 45 type[1] = MPI\_INT; 46 disp[0] = 0;47disp[1] = offsetof(struct mystruct, rank); 48

```
1
block[0] = 1;
                                                                                          \mathbf{2}
block[1] = 1;
                                                                                          3
MPI_Type_create_struct(2, block, disp, type, MPI_SHORT_INT);
                                                                                          4
Similar statements apply for MPI_FLOAT_INT, MPI_LONG_INT and MPI_DOUBLE_INT.
                                                                                          5
    The following examples use intra-communicators.
                                                                                          6
                                                                                          7
Example 6.17 Each process has an array of 30 doubles, in C. For each of the 30 locations,
                                                                                          8
compute the value and rank of the process containing the largest value.
                                                                                          9
                                                                                          10
    . . .
                                                                                          11
    /* each process has an array of 30 double: ain[30]
                                                                                          12
     */
                                                                                          13
    double ain[30], aout[30];
                                                                                          14
    int ind[30];
                                                                                          15
    struct {
                                                                                          16
        double val;
                                                                                          17
         int
               rank;
                                                                                          18
    } in[30], out[30];
                                                                                          19
    int i, myrank, root;
                                                                                          20
                                                                                          21
    MPI_Comm_rank(comm, &myrank);
                                                                                          22
    for (i=0; i<30; ++i) {
                                                                                          23
         in[i].val = ain[i];
                                                                                          ^{24}
         in[i].rank = myrank;
                                                                                          25
    }
                                                                                          26
    MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
                                                                                          27
    /* At this point, the answer resides on process root
                                                                                          28
     */
                                                                                          29
    if (myrank == root) {
                                                                                          30
         /* read ranks out
                                                                                          31
          */
                                                                                          32
         for (i=0; i<30; ++i) {
                                                                                          33
             aout[i] = out[i].val;
                                                                                          34
             ind[i] = out[i].rank;
                                                                                          35
                                                                                          36
    }
                                                                                          37
                                                                                          38
Example 6.18 Same example, in Fortran.
                                                                                          39
                                                                                          40
                                                                                          41
. . .
                                                                                          42
! each process has an array of 30 double: ain(30)
                                                                                          43
DOUBLE PRECISION ain(30), aout(30)
                                                                                          44
INTEGER ind(30)
                                                                                          45
DOUBLE PRECISION in(2,30), out(2,30)
                                                                                          46
INTEGER i, myrank, root, ierr
                                                                                          47
                                                                                          48
```

```
1
     CALL MPI_COMM_RANK(comm, myrank, ierr)
\mathbf{2}
     DO i=1,30
3
        in(1,i) = ain(i)
4
        in(2,i) = myrank
                               ! myrank is coerced to a double
\mathbf{5}
     END DO
6
7
     CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,&
8
                       comm, ierr)
9
     ! At this point, the answer resides on process root
10
11
     IF (myrank .EQ. root) THEN
12
         ! read ranks out
13
        DO i=1,30
14
            aout(i) = out(1,i)
15
            ind(i) = out(2,i) ! rank is coerced back to an integer
16
        END DO
17
     END IF
18
19
     Example 6.19 Each process has a non-empty array of values. Find the minimum global
20
     value, the rank of the process that holds it and its index on this process.
21
22
     #define LEN
                      1000
23
24
                              /* local array of values */
     float val[LEN];
25
                               /* local number of values */
     int count;
26
     int myrank, minrank, minindex;
27
     float minval;
28
29
     struct {
30
         float value;
^{31}
         int
               index;
32
     } in, out;
33
34
         /* local minloc
35
     in.value = val[0];
36
     in.index = 0;
37
     for (i=1; i < count; i++)</pre>
38
         if (in.value > val[i]) {
39
              in.value = val[i];
40
              in.index = i;
41
         }
42
43
         /* global minloc */
44
     MPI_Comm_rank(comm, &myrank);
45
     in.index = myrank*LEN + in.index;
46
     MPI_Reduce(&in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm);
47
         /* At this point, the answer resides on process root
48
```

```
*/
                                                                                         \mathbf{2}
if (myrank == root) {
                                                                                          3
    /* read answer out
     */
                                      By assigning a value other than \code{myrank} to the
    minval = out.value;
                                      \code{in.index} field, a programmer can provide a different
    minrank = out.index / LEN;
                                     definition of \const{MPI\_MAXLOC} and \const{MPI
    minindex = out.index % LEN;
                                      \_MINLOC}, if so desired.
}
                                                                                         9
     Rationale.
                  The definition of MPI_MINLOC and MPI_MAXLOC given here has the
                                                                                         10
     advantage that it does not require any special-case handling of these two operations:
                                                                                         11
                                                                                            #-error!
     they are handled like any other reduce operation. A programmer can provide his or
                                                                                         12
                                                                                         13 #-PR415
     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
                                                                                         <sup>14</sup> #-update5
     to be coerced to the same type, in Fortran. (End of rationale.)
                                                                                         15
                                                                                            Done
                                                                                         16
                                                                                            lin
                                                                                            PR415
                                                                                         17
6.9.5 User-Defined Reduction Operations
                                                                                         18
                                                                                         19
                                                                                         20
MPI_OP_CREATE(user_fn, commute, op)
                                                                                         21
 IN
           user_fn
                                       user defined function (function)
                                                                                         22
 IN
           commute
                                       true if commutative; false otherwise.
                                                                                         23
                                                                                         24
 OUT
                                       operation (handle)
           op
                                                                                         25
                                                                                         26
C binding
                                                                                         27
int MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)
                                                                                         28
int MPI_Op_create_c(MPI_User_function_c *user_fn, int commute, MPI_Op *op)
                                                                                         29
                                                                                         30
Fortran 2008 binding
                                                                                         31
MPI_Op_create(user_fn, commute, op, ierror)
                                                                                         32
    PROCEDURE(MPI_User_function), INTENT(IN) :: user_fn
                                                                                         33
    LOGICAL, INTENT(IN) :: commute
                                                                                         34
    TYPE(MPI_Op), INTENT(OUT) :: op
                                                                                         35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         36
                                                                                         37
MPI_Op_create_c(user_fn, commute, op, ierror)
                                                                                         38
    PROCEDURE(MPI_User_function_c), INTENT(IN) :: user_fn
                                                                                         39
    LOGICAL, INTENT(IN) :: commute
                                                                                         40
    TYPE(MPI_Op), INTENT(OUT) :: op
                                                                                         41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         42
Fortran binding
                                                                                         43
MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR)
                                                                                         44
    EXTERNAL USER_FN
                                                                                         45
    LOGICAL COMMUTE
```

INTEGER OP, IERROR

46

1 2 3 4 5 6 7 8 9 10 11 12 #-error! 13 #-other 15	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 6.12), and MPI_REDUCE_LOCAL. The user-defined operation is assumed to be associative. If commute = true, then the operation should be both commutative and associative. If commute = false, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, talking advantage of the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity. The argument user_fn is the user-defined function, which must have the following four arguments: invec, inoutvec, len, and datatype. MPI_USER_FUNCTION also supports large count types in separate additional MPI procedures in C (suffixed with the "_c") and interface polymorphism in Fortran when using USE mpi_f08
This is a callback prototype, use correct macro!	<pre>USE mpi_f08. The ISO C prototypes for the functions are the following. typedef void MPI_User_function(void *invec, void *inoutvec, int *len,</pre>
in 19 PR449 20 21	<pre>typedef void MPI_User_function_c(void *invec, void *inoutvec,</pre>
22 23 24 25 26 27 28 29 30 31 32 33 34 35	<pre>The Fortran declarations of the user-defined function user_fn appear below. ABSTRACT INTERFACE SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: invec, inoutvec INTEGER :: len TYPE(MPI_Datatype) :: datatype ABSTRACT INTERFACE SUBROUTINE MPI_User_function_c(invec, inoutvec, len, datatype) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: invec, inoutvec INTEGER(KIND=MPI_COUNT_KIND) :: len TYPE(MPI_Datatype) :: datatype</pre>
33 36 37 38 39 40 41 42 43 44 45 46 47 48	<pre>SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)</pre>

# Unofficial Draft for Comment Only

Informally, we can think of invec and inoutvec as arrays of len elements that user\_fn is combining. The result of the reduction over-writes values in inoutvec, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on len elements: i.e., the function returns in inoutvec[i] the value invec[i]  $\circ$  inoutvec[i], for i=0, ..., count-1, where  $\circ$  is the combining operation computed by the function.

*Rationale.* The len argument allows MPI\_REDUCE to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different data types. (*End of rationale.*)

When calling any reduction or prefix scan MPI procedure with a user-defined MPI operator, the type of the count parameter in the call to the reduction or prefix scan MPI procedure does not need to be identical to the type of the len parameter in the user function associated with the user-defined MPI operator. If the count parameter has a type of int in C or INTEGER in Fortran and the len parameter has a type of MPI\_COUNT, then MPI will perform the appropriate widening type conversion of the len parameter. If the count parameter has a type of MPI\_COUNT and the len parameter has a type of int in C or INTEGER in Fortran, then MPI will perform the appropriate narrowing type conversion of the len parameter. If this narrowing conversion would result in truncation of the len value, then MPI will call the user function multiple times with a sequence of values for len that sum to the value of count.

Advice to implementors. If the number of data items cannot be represented in len, the implementation may need to invoke user\_fn multiple times. (End of advice to implementors.)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. MPI\_ABORT may be called inside the function in case of an error.

Advice to users. Suppose one defines a library of user-defined reduce functions that are overloaded: the datatype argument is used to select the right execution path at each invocation, according to the types of the operands. The user-defined reduce function cannot "decode" the datatype argument that it is passed, and cannot identify, by itself, the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the datatypes that are used by the library, and store handles to these datatypes in global, static variables that are shared by the user code and the library code.

The Fortran version of MPI\_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a datatype handle. 48

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

Users who plan to mix languages should define their reduction functions accordingly. (*End of advice to users.*)

Advice to implementors. We outline below a naive and inefficient implementation of MPI\_REDUCE not supporting the "in place" option.

```
7
                 MPI_Comm_size(comm, &groupsize);
8
                 MPI_Comm_rank(comm, &rank);
9
                 if (rank > 0) {
10
                     MPI_Recv(tempbuf, count, datatype, rank-1,...);
11
                     User_reduce(tempbuf, sendbuf, count, datatype);
12
                 }
13
                 if (rank < groupsize-1) {</pre>
14
                     MPI_Send(sendbuf, count, datatype, rank+1, ...);
15
                 }
                 /* answer now resides in process groupsize-1 ... now send to root
16
17
                  */
18
                 if (rank == root) {
19
                     MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
                 }
20
21
                 if (rank == groupsize-1) {
22
                     MPI_Send(sendbuf, count, datatype, root, ...);
23
                 }
24
                 if (rank == root) {
25
                     MPI_Wait(&req, &status);
26
                 }
27
28
          The reduction computation proceeds, sequentially, from process 0 to process
29
          groupsize-1. This order is chosen so as to respect the order of a possibly non-
30
```

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.*)

40 41 42

43

44 45

46

31

32

33

34

35

36

37

38

39

MPI\_OP\_FREE(op)

INOUT op operation (handle) C binding

47 int MPI\_Op\_free(MPI\_Op \*op)

<sup>48</sup> Fortran 2008 binding

1

2

3

4

```
1
MPI_Op_free(op, ierror)
                                                                                           \mathbf{2}
    TYPE(MPI_Op), INTENT(INOUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
                                                                                           5
MPI_OP_FREE(OP, IERROR)
                                                                                           6
    INTEGER OP, IERROR
    Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.
                                                                                           9
                                                                                           10
Example of User-Defined Reduce
                                                                                          11
It is time for an example of user-defined reduction. The example in this section uses an
                                                                                          12
intra-communicator.
                                                                                          13
                                                                                          14
Example 6.20 Compute the product of an array of complex numbers, in C.
                                                                                           15
                                                                                           16
typedef struct {
                                                                                           17
    double real, imag;
                                                                                           18
} Complex;
                                                                                           19
/* the user-defined function
                                                                                           20
                                                                                          21
 */
void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)
                                                                                          22
                                                                                          23
{
                                                                                           ^{24}
    int i;
                                                                                          25
    Complex c;
                                                                                           26
    Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;
                                                                                          27
    for (i=0; i< *len; ++i) {</pre>
                                                                                          28
                                                                                          29
         c.real = inout->real*in->real -
                                                                                           30
                      inout->imag*in->imag;
                                                                                           31
         c.imag = inout->real*in->imag +
                                                                                           32
                      inout->imag*in->real;
                                                                                           33
         *inout = c;
                                                                                          34
         in++; inout++;
                                                                                          35
    }
}
                                                                                          36
                                                                                          37
                                                                                           38
/* and, to call it.
                                                                                           39
 */
                                                                                           40
. . .
                                                                                           41
                                                                                          42
    /* each process has an array of 100 Complexes
      */
                                                                                           43
                                                                                           44
    Complex a[100], answer[100];
                                                                                           45
    MPI_Op myOp;
                                                                                           46
    MPI_Datatype ctype;
                                                                                           47
                                                                                           48
    /* explain to MPI how type Complex is defined
```

```
1
           */
\mathbf{2}
          MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
3
          MPI_Type_commit(&ctype);
4
          /* create the complex-product user-op
5
           */
6
          MPI_Op_create(myProd, 1, &myOp);
7
8
          MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
9
10
          /* At this point, the answer, which consists of 100 Complexes,
11
           * resides on process root
12
           */
13
14
     Example 6.21 How to use the mpi_f08 interface of the Fortran MPI_User_function.
15
16
     subroutine my_user_function(invec, inoutvec, len, type)
                                                                         bind(c)
17
         use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
18
         use mpi_f08
19
         type(c_ptr), value :: invec, inoutvec
20
         integer :: len
21
         type(MPI_Datatype) :: type
22
         real, pointer :: invec_r(:), inoutvec_r(:)
23
         if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
^{24}
            call c_f_pointer(invec, invec_r, (/ len /))
25
            call c_f_pointer(inoutvec, inoutvec_r, (/ len /))
26
            inoutvec_r = invec_r + inoutvec_r
27
         end if
28
     end subroutine
29
30
     6.9.6 All-Reduce
^{31}
     MPI includes a variant of the reduce operations where the result is returned to all processes
32
     in a group. MPI requires that all processes from the same group participating in these
33
     operations receive identical results.
34
35
36
     MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)
37
       IN
                 sendbuf
                                              starting address of send buffer (choice)
38
39
       OUT
                 recvbuf
                                              starting address of receive buffer (choice)
40
       IN
                 count
                                              number of elements in send buffer (non-negative
41
                                              integer)
42
       IN
                                              data type of elements of send buffer (handle)
                 datatype
43
44
       IN
                                              operation (handle)
                 ор
45
       IN
                                              communicator (handle)
                 comm
46
47
     C binding
48
```

	1
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,	1 2
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	- 3
<pre>int MPI_Allreduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>	4
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	5
Fortran 2008 binding	6
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)	7
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	8
TYPE(*), DIMENSION() :: recvbuf	9
INTEGER, INTENT(IN) :: count	10
TYPE(MPI_Datatype), INTENT(IN) :: datatype	11
TYPE(MPI_Op), INTENT(IN) :: op	12
TYPE(MPI_Comm), INTENT(IN) :: comm	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
	15
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)	16
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	17
TYPE(*), DIMENSION() :: recvbuf	18
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype	19
TYPE(MPI_Datatype), INTENT(IN) datatype TYPE(MPI_Op), INTENT(IN) :: op	20
TYPE(MPI_Comm), INTENT(IN) :: comm	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
	23
Fortran binding	24
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	25 26
<type> SENDBUF(*), RECVBUF(*)</type>	20 27
INTEGER COUNT, DATATYPE, OP, COMM, IERROR	27
If comm is an intra-communicator, MPI_ALLREDUCE behaves the same as	29
MPI_REDUCE except that the result appears in the receive buffer of all the group members.	30
	31
Advice to implementors. The all-reduce operations can be implemented as a re-	32
duce, followed by a broadcast. However, a direct implementation can lead to better	33
performance. (End of advice to implementors.)	34
	35
The "in place" option for intra-communicators is specified by passing the value	36
MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is	37
taken at each process from the receive buffer, where it will be replaced by the output data.	38
If comm is an inter-communicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group P and vice years. Both groups	39
by processes in group A is stored at each process in group B, and vice versa. Both groups should provide <b>count</b> and <b>datatype</b> arguments that specify the same type signature.	40
The following example uses an intra-communicator.	41
The following example uses an initia communicator.	42
<b>Example 6.22</b> A routine that computes the product of a vector and an array that are	43
distributed across a group of processes and returns the answer at all nodes (see also Exam-	44
ple 6.16).	45
	46
	47
	48

```
1
     SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
\mathbf{2}
     REAL a(m), b(m,n)
                              ! local slice of array
3
     REAL c(n)
                              ! result
4
     REAL sum(n)
\mathbf{5}
     INTEGER n, comm, i, j, ierr
6
7
      ! local sum
     DO j=1,n
8
9
         sum(j) = 0.0
10
         DO i=1,m
11
            sum(j) = sum(j) + a(i)*b(i,j)
12
         END DO
13
     END DO
14
15
      ! global sum
16
     CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
17
18
      ! return result at all nodes
19
     RETURN
20
     END
21
22
     6.9.7
             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
                 count
                                              number of elements in inbuf and inoutbuf buffers
34
                                              (non-negative integer)
35
       IN
                                              data type of elements of inbuf and inoutbuf buffers
                  datatype
36
                                              (handle)
37
38
       IN
                                              operation (handle)
                  op
39
40
     C binding
41
      int MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count,
42
                     MPI_Datatype datatype, MPI_Op op)
43
      int MPI_Reduce_local_c(const void *inbuf, void *inoutbuf, MPI_Count count,
44
                     MPI_Datatype datatype, MPI_Op op)
45
46
      Fortran 2008 binding
47
      MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
48
```

```
1
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                        2
    TYPE(*), DIMENSION(..) :: inoutbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        4
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        5
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        6
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                        a
    TYPE(*), DIMENSION(..) :: inoutbuf
                                                                                        10
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                        11
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        12
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        14
                                                                                        15
Fortran binding
                                                                                        16
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
                                                                                        17
    <type> INBUF(*), INOUTBUF(*)
                                                                                        18
    INTEGER COUNT, DATATYPE, OP, IERROR
                                                                                        19
    The function applies the operation given by op element-wise to the elements of inbuf
                                                                                        20
and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined
                                                                                        21
operations in Section 6.9.5. Both inbuf and inoutbuf (input as well as result) have the
                                                                                        22
same number of elements given by count and the same datatype given by datatype. The
                                                                                        23
MPI_IN_PLACE option is not allowed.
                                                                                        24
    Reduction operations can be queried for their commutativity.
                                                                                        25
                                                                                        26
                                                                                        27
MPI_OP_COMMUTATIVE(op, commute)
                                                                                        28
  IN
                                      operation (handle)
           op
                                                                                        29
  OUT
           commute
                                       true if op is commutative, false otherwise (logical)
                                                                                        30
                                                                                        31
                                                                                        32
C binding
                                                                                        33
int MPI_Op_commutative(MPI_Op op, int *commute)
                                                                                        34
Fortran 2008 binding
                                                                                        35
MPI_Op_commutative(op, commute, ierror)
                                                                                        36
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        37
    LOGICAL, INTENT(OUT) :: commute
                                                                                        38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        39
                                                                                        40
Fortran binding
                                                                                        41
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
                                                                                        42
    INTEGER OP, IERROR
                                                                                        43
    LOGICAL COMMUTE
                                                                                        44
                                                                                        45
```

Reduce-Scatter 6.10 1  $\mathbf{2}$ MPI includes variants of the reduce operations where the result is scattered to all processes 3 in a group on return. One variant scatters equal-sized blocks to all processes, while another 4 variant scatters blocks that may vary in size for each process. 56 6.10.1 MPI\_REDUCE\_SCATTER\_BLOCK 7 8 9 10 MPI\_REDUCE\_SCATTER\_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm) 11 IN sendbuf starting address of send buffer (choice) 12OUT recvbuf starting address of receive buffer (choice) 13 14IN element count per block (non-negative integer) recvcount 15IN datatype data type of elements of send and receive buffers 16 (handle) 17IN operation (handle) 18 op 19 IN comm communicator (handle) 2021C binding 22 int MPI\_Reduce\_scatter\_block(const void \*sendbuf, void \*recvbuf, 23int recvcount, MPI\_Datatype datatype, MPI\_Op op, 24MPI\_Comm comm) 2526int MPI\_Reduce\_scatter\_block\_c(const void \*sendbuf, void \*recvbuf, MPI\_Count recvcount, MPI\_Datatype datatype, MPI\_Op op, 27MPI\_Comm comm) 2829 Fortran 2008 binding 30 MPI\_Reduce\_scatter\_block(sendbuf, recvbuf, recvcount, datatype, op, comm,  $^{31}$ ierror) 32 TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf 33 TYPE(\*), DIMENSION(..) :: recvbuf 34 INTEGER, INTENT(IN) :: recvcount 35TYPE(MPI\_Datatype), INTENT(IN) :: datatype 36 TYPE(MPI\_Op), INTENT(IN) :: op 37 TYPE(MPI\_Comm), INTENT(IN) :: comm 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40MPI\_Reduce\_scatter\_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 41 ierror) 42TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf TYPE(\*), DIMENSION(..) :: recvbuf 43 44INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: recvcount 45TYPE(MPI\_Datatype), INTENT(IN) :: datatype 46TYPE(MPI\_Op), INTENT(IN) :: op 47 TYPE(MPI\_Comm), INTENT(IN) :: comm 48 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Unofficial Draft for Comment Only

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

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

The "in place" option for intra-communicators is specified by passing MPI\_IN\_PLACE in the sendbuf argument on *all* processes. In this case, the input data is taken from the receive buffer.

If comm is an inter-communicator, 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.*)

# 6.10.2 MPI\_REDUCE\_SCATTER

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

 $\mathbf{2}$ 

 $^{24}$ 

```
1
     MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)
\mathbf{2}
       IN
                sendbuf
                                            starting address of send buffer (choice)
3
       OUT
                recvbuf
                                            starting address of receive buffer (choice)
4
5
       IN
                                            non-negative integer array (of length group size)
                 recvcounts
6
                                            specifying the number of elements of the result
7
                                            distributed to each process.
8
       IN
                                            data type of elements of send and receive buffers
                datatype
9
                                            (handle)
10
       IN
                                            operation (handle)
                 ор
11
12
       IN
                 comm
                                            communicator (handle)
13
14
     C binding
15
     int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,
16
                    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
17
                    MPI_Comm comm)
18
     int MPI_Reduce_scatter_c(const void *sendbuf, void *recvbuf,
19
                    const MPI_Count recvcounts[], MPI_Datatype datatype,
20
                    MPI_Op op, MPI_Comm comm)
21
22
     Fortran 2008 binding
23
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
24
                    ierror)
25
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
26
         TYPE(*), DIMENSION(..) :: recvbuf
27
         INTEGER, INTENT(IN) :: recvcounts(*)
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Op), INTENT(IN) :: op
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
33
34
                    ierror)
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
35
         TYPE(*), DIMENSION(..) :: recvbuf
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcounts(*)
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         TYPE(MPI_Op), INTENT(IN) :: op
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     Fortran binding
43
     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
44
                    IERROR)
45
          <type> SENDBUF(*), RECVBUF(*)
46
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
47
48
```

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

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

The "in place" option for intra-communicators 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 inter-communicator, 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.*)

# 6.11 Scan

6.11.1	Inclusive	e Scan	

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

IN	sendbuf	starting address of send buffer (choice)	38
OUT	recvbuf	starting address of receive buffer (choice)	39
IN	count	number of elements in input buffer (non-negative integer)	40 41 42
IN	datatype	data type of elements of input buffer (handle)	42 43
IN	ор	operation (handle)	44
IN	comm	communicator (handle)	45
			46

# C binding

 $\mathbf{2}$ 

 $^{24}$ 

```
1
     int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
\mathbf{2}
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
3
     int MPI_Scan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
4
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
5
6
     Fortran 2008 binding
\overline{7}
     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
8
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
9
         TYPE(*), DIMENSION(..) :: recvbuf
10
         INTEGER, INTENT(IN) :: count
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         TYPE(MPI_Op), INTENT(IN) :: op
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm,
                                                                  ierror)
16
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
17
         TYPE(*), DIMENSION(...) :: recvbuf
18
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
19
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
         TYPE(MPI_Op), INTENT(IN) :: op
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     Fortran binding
25
     MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
26
          <type> SENDBUF(*), RECVBUF(*)
27
         INTEGER COUNT, DATATYPE, OP, COMM, IERROR
28
         If comm is an intra-communicator, MPI_SCAN is used to perform a prefix reduction
29
     on data distributed across the group. The operation returns, in the receive buffer of the
30
     process with rank i, the reduction of the values in the send buffers of processes with ranks
31
     0, \ldots, i (inclusive). The routine is called by all group members using the same arguments
32
     for count, datatype, op and comm, except that for user-defined operations, the same rules
33
     apply as for MPI_REDUCE. The type of operations supported, their semantics, and the
34
     constraints on send and receive buffers are as for MPI_REDUCE.
35
         The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE
36
     in the sendbuf argument. In this case, the input data is taken from the receive buffer, and
37
     replaced by the output data.
38
         This operation is invalid for inter-communicators.
39
40
41
42
43
44
45
46
47
48
```

6.11.2	Exclusive Scan		1	
			2	
			3	
MPI_E	XSCAN(sendbuf, recvbuf, co	unt, datatype, op, comm)	4 5	
IN	sendbuf	starting address of send buffer (choice)	6	
OUT	recvbuf	starting address of receive buffer (choice)	7	
IN	count	number of elements in input buffer (non-negative	8	
	count	integer)	9	
IN	datatype	data type of elements of input buffer (handle)	10 11	
IN	ор	operation (handle)	12	
IN	comm	intra-communicator (handle)	13	
			14	
C bine	ling		15	
int MF	I_Exscan(const void *se	endbuf, void *recvbuf, int count,	16 17	
	MPI_Datatype da	tatype, MPI_Op op, MPI_Comm comm)	18	
int MF	I_Exscan_c(const void >	<pre>*sendbuf, void *recvbuf, MPI_Count count,</pre>	19	
		tatype, MPI_Op op, MPI_Comm comm)	20	
Fortran 2008 binding MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)				
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf				
	PE(*), DIMENSION() :		24 25	
	TEGER, INTENT(IN) :: co		26	
	PE(MPI_Datatype), INTEN		27	
	PE(MPI_Op), INTENT(IN)	-	28	
TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
ΞN	IEGER, UPIIONAL, INIEN.		30	
		count, datatype, op, comm, ierror)	31	
	PE(*), DIMENSION(), I		32	
	PE(*), DIMENSION() ::		33 34	
	PE(MPI_Datatype), INTEN	IND), INTENT(IN) :: count	35	
	PE(MPI_Op), INTENT(IN)	V -	36	
	PE(MPI_Comm), INTENT(IN	-	37	
IN	TEGER, OPTIONAL, INTENT	Γ(OUT) :: ierror	38	
Fortra	n binding		39	
	0	COUNT, DATATYPE, OP, COMM, IERROR)	40	
	ype> SENDBUF(*), RECVBU		41 42	
INTEGER COUNT, DATATYPE, OP, COMM, IERROR				

44If comm is an intra-communicator, 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 4546undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process 47with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes 48 with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the

# **Unofficial Draft for Comment Only**

reduction of the values in the send buffers of processes with ranks  $0, \ldots, i-1$  (inclusive). The  $\mathbf{2}$ 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 intra-communicators 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 inter-communicators.

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.)

6.11.3 Example using MPI\_SCAN

The example in this section uses an intra-communicator.

**Example 6.23** 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	$v_1$	$v_1 + v_2$	$v_3$	$v_3 + v_4  v_3 + v_4$	$4 + v_5 v_6$	$v_6 + v_7$	$v_8$

The operator that produces this effect is

 $\begin{pmatrix} v \\ i \end{pmatrix} = \begin{pmatrix} w \\ i \end{pmatrix},$ 

where

$w = \left\{ \right.$	u + v	if $i = j$	
$w = \left\{ \right.$	v	if $i \neq j$	•

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

 $\overline{7}$ 

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

Note that the inout argument to the user-defined function corresponds to the righthand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.

```
28
int i, base;
                                                                                  29
SegScanPair a, answer;
                                                                                  30
MPI_Op
              myOp;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
                                                                                  31
                                                                                  32
MPI_Aint
              disp[2];
                                                                                  33
int
              blocklen[2] = { 1, 1};
                                                                                  34
MPI_Datatype sspair;
                                                                                  35
                                                                                  36
/* explain to MPI how type SegScanPair is defined
 */
                                                                                  37
MPI_Get_address(&a, disp);
                                                                                  38
                                                                                  39
MPI_Get_address(&a.log, disp+1);
                                                                                  40
base = disp[0];
                                                                                  41
for (i=0; i<2; ++i) disp[i] -= base;</pre>
                                                                                  42
MPI_Type_create_struct(2, blocklen, disp, type, &sspair);
MPI_Type_commit(&sspair);
                                                                                  43
                                                                                  44
/* create the segmented-scan user-op
                                                                                  45
 */
                                                                                  46
MPI_Op_create(segScan, 0, &myOp);
                                                                                  47
. . .
                                                                                  48
MPI_Scan(&a, &answer, 1, sspair, myOp, comm);
```

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

#### 1 2

7

#### 6.12 Nonblocking Collective Operations

As described in Section 3.7, performance of many applications can be improved by over-3 lapping communication and computation, and many systems enable this. Nonblocking 4 collective operations combine the potential benefits of nonblocking point-to-point opera-5tions, to exploit overlap and to avoid synchronization, with the optimized implementation 6 and message scheduling provided by collective operations [34, 38]. One way of doing this would be to perform a blocking collective operation in a separate thread. An alternative 8 mechanism that often leads to better performance (e.g., avoids context switching, scheduler 9 overheads, and thread management) is to use nonblocking collective communication [36]. 10

The nonblocking collective communication model is similar to the model used for non-11 blocking point-to-point communication. A nonblocking call initiates a collective operation, 12which must be completed in a separate completion call. Once initiated, the operation 13 may progress independently of any computation or other communication at participating 14processes. In this manner, nonblocking collective operations can mitigate possible synchro-15nizing effects of collective operations by running them in the "background." In addition to 16enabling communication-computation overlap, nonblocking collective operations can per-17form collective operations on overlapping communicators, which would lead to deadlocks 18 with blocking operations. Their semantic advantages can also be useful in combination with 19point-to-point communication. 20

As in the nonblocking point-to-point case, all calls are local and return immediately, 21irrespective of the status of other processes. The call initiates the operation, which indicates 22 that the system may start to copy data out of the send buffer and into the receive buffer. 23Once initiated, all associated send buffers and buffers associated with input arguments (such  $^{24}$ as arrays of counts, displacements, or datatypes in the vector versions of the collectives) 25should not be modified, and all associated receive buffers should not be accessed, until the 26collective operation completes. The call returns a request handle, which must be passed to 27a completion call. 28

All completion calls (e.g., MPI\_WAIT) described in Section 3.7.3 are supported for 29 nonblocking collective operations. Similarly to the blocking case, nonblocking collective 30 operations are considered to be complete when the local part of the operation is finished,  $^{31}$ i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be 32 safely accessed and modified. Completion does not indicate that other processes have 33 completed or even started the operation (unless otherwise implied by the description of 34the operation). Completion of a particular nonblocking collective operation also does not 35 indicate completion of any other posted nonblocking collective (or send-receive) operations, 36 whether they are posted before or after the completed operation. 37

- 38
- 39 40 41

42

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 43 completes, the values of the MPI\_SOURCE and MPI\_TAG fields in the associated status object, 44 if any, are undefined. The value of MPI\_ERROR may be defined, if appropriate, according 45to the specification in Section 3.2.5. It is valid to mix different request types (i.e., any 46 combination of collective requests, I/O requests, generalized requests, or point-to-point 47requests) in functions that enable multiple completions (e.g., MPI\_WAITALL). It is erroneous 48

to call MPI\_REQUEST\_FREE or MPI\_CANCEL for a request associated with a nonblocking collective operation. Nonblocking collective requests created using the APIs described in this section are not persistent. However, persistent collective requests can be created using persistent collective operations described in Sections 6.13 and 8.8.

Rationale. Freeing an active nonblocking collective request could cause similar problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a request is not supported because the semantics of this operation are not well-defined. (End of rationale.)

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

Unlike point-to-point operations, nonblocking collective operations do not match with blocking collective operations, and collective operations do not have a tag argument. All processes must call collective operations (blocking and nonblocking) in the same order per communicator. In particular, once a process calls a collective operation, all other processes in the communicator must eventually call the same collective operation, and no other collective operation with the same communicator in between. This is consistent with 20the ordering rules for blocking collective operations in threaded environments.

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

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 intra-communicators and inter-communicators 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.

46Advice to implementors. Nonblocking collective operations can be implemented with 47local execution schedules [37] using nonblocking point-to-point communication and a 48 reserved tag-space. (End of advice to implementors.)

## **Unofficial Draft for Comment Only**

1  $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

2122

23

24

25

26

27

28

29 30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

42

43

44

1	6.12.1 Nonblocking Barrier Synchronization				
3					
4	MPI_IBAR	MPI_IBARRIER(comm, request)			
5	IN	comm	communicator (handle)		
6 7	OUT	request	communication request (handle)		
8					
9 10 11	C binding int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)				
12 13 14 15 16	Fortran 2008 binding MPI_Ibarrier(comm, request, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
17 18 19 20	MPI_IBARRIER(COMM, REQUEST, IERROR) INTEGER COMM, REQUEST, IERROR				
21 22 23 24 25 26 27 28 29 30 31 32 33	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 inter-communicator 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.)				
34					
35	6.12.2 Nonblocking Broadcast				
36 37					
38	MPI_IBCA	ST(buffer, count, datatype, ro	oot, comm, request)		
39	INOUT	buffer	starting address of buffer (choice)		
40 41	IN	count	number of entries in buffer (non-negative integer)		
42	IN	datatype	data type of buffer (handle)		
43	IN	root	rank of broadcast root (integer)		
44 45	IN	comm	communicator (handle)		
45 46	OUT	request	communication request (handle)		
47					
48	<sup>48</sup> C binding				

```
1
int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root,
                                                                                      2
              MPI_Comm comm, MPI_Request *request)
int MPI_Ibcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype,
              int root, MPI_Comm comm, MPI_Request *request)
Fortran 2008 binding
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: count, root
                                                                                     10
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                     11
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                     12
                                                                                     13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     14
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
                                                                                     15
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                     16
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                     17
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                     18
    INTEGER, INTENT(IN) :: root
                                                                                     19
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     20
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                     21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     22
                                                                                     23
Fortran binding
                                                                                     ^{24}
MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                     25
    <type> BUFFER(*)
                                                                                     26
    INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR
                                                                                     27
    This call starts a nonblocking variant of MPI_BCAST (see Section 6.4).
                                                                                     28
                                                                                     29
Example using MPI_IBCAST
                                                                                     30
                                                                                     31
The example in this section uses an intra-communicator.
                                                                                     32
                                                                                     33
Example 6.24 Start a broadcast of 100 ints from process 0 to every process in the
                                                                                     34
group, perform some computation on independent data, and then complete the outstanding
                                                                                     35
broadcast operation.
                                                                                     36
                                                                                     37
    MPI_Comm comm;
    int array1[100], array2[100];
                                                                                     38
                                                                                     39
    int root=0;
    MPI_Request req;
                                                                                     40
                                                                                     41
    . . .
                                                                                     42
    MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
    compute(array2, 100);
                                                                                     43
                                                                                     44
    MPI_Wait(&req, MPI_STATUS_IGNORE);
                                                                                     45
                                                                                     46
                                                                                     47
                                                                                     48
```

1	6123 N	lonblocking Gather	
2	0.12.0	ionsidering dutier	
3 4 5	MPI_IGAT	HER(sendbuf, sendcount, send request)	dtype, recvbuf, recvcount, recvtype, root, comm,
6 7	IN	sendbuf	starting address of send buffer (choice)
8 9	IN	sendcount	number of elements in send buffer (non-negative integer)
10	IN	sendtype	data type of send buffer elements (handle)
11 12 13	OUT	recvbuf	address of receive buffer (choice, significant only at root)
14 15	IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)
16 17	IN	recvtype	data type of recv buffer elements (handle, significant only at root)
18 19	IN	root	rank of receiving process (integer)
20	IN	comm	communicator (handle)
21 22	OUT	request	communication request (handle)
26 27 28 29 30 31	<pre>void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request) int MPI_Igather_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>		
32 33 34 35 36 37 38 39 40 41 42	<pre>Fortran 2008 binding MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,</pre>		
43 44 45 46 47 48	<pre>MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,</pre>		

	EGER, INTENT(IN) ::	1000	1			
	TYPE(MPI_Comm), INTENT(IN) :: comm					
	TYPE(MPI_Request), INTENT(OUT) :: request					
TNU	EGER, UPTIONAL, INT		4 5			
Fortran	binding		6			
MPI_IGA			7			
			8			
• •	pe> SENDBUF(*), REC	DTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	9			
11111	IERROR		10			
			$11 \\ 12$			
This	call starts a nonblock	see Section 0.5).	13			
			14			
MPI_IGA	THERV(sendbuf, sendo comm, request)		$15 \\ 16$			
IN	sendbuf		17			
			18			
IN	sendcount	integer)	19 20			
IN	sendtype	data type of send buffer elements (handle)	21			
OUT	recvbuf	address of receive buller (choice, significant only at	22 23			
IN	recvcounts	non-negative integer array (of length group size)	24			
		containing the number of elements that are received from each process (significant only at root)	25 26 27			
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to <b>recvbuf</b> at which to place the incoming data from process i (significant only at	28 29 30 31			
IN	recvtype	data type of feev buner elements (handle, significant	32 33			
IN	root	rank of receiving process (integer)	34			
IN	comm		35			
OUT			$\frac{36}{37}$			
001	request	communication request (handle)	38			
C bindi	ng	3	39			
	U U	.d *sendbuf, int sendcount, MPI_Datatype sendtype,	40			
	-	f, const int recvcounts[], const int displs[],	41			
	MPI_Datatype	recvtype, int root, MPI_Comm comm,	$42 \\ 43$			
	MPI_Request	*request)	44 44			
int MPI	_Igatherv_c(const v		45			
	MPI_Datatype sendtype, void *recvbuf, 46					
	const MPI_Co	unt recvcounts[], const MPI_Aint displs[], 4	47			
		4	48			

```
1
                   MPI_Datatype recvtype, int root, MPI_Comm comm,
2
                   MPI_Request *request)
3
     Fortran 2008 binding
4
     MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
5
                   recvtype, root, comm, request, ierror)
6
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
7
         INTEGER, INTENT(IN) :: sendcount, root
8
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
16
                   recvtype, root, comm, request, ierror)
17
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
18
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
19
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
21
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
22
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
23
         INTEGER, INTENT(IN) :: root
24
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     Fortran binding
28
     MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
29
                   RECVTYPE, ROOT, COMM, REQUEST, IERROR)
30
         <type> SENDBUF(*), RECVBUF(*)
31
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
32
                    COMM, REQUEST, IERROR
33
34
         This call starts a nonblocking variant of MPI_GATHERV (see Section 6.5).
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

CHAPTER 6. COLLECTIVE COMMUNICATION

# 6.12.4 Nonblocking Scatter

			2 3		
MPI_ISC	ATTER(sendbuf, sendcou request)	nt, sendtype, recvbuf, recvcount, recvtype, root, comm,	3 4 5		
IN	sendbuf	address of send buffer (choice, significant only at root)	6 7 8		
IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)	9 10		
IN	sendtype	data type of send buffer elements (handle, significant only at root)	11 12		
OUT	recvbuf	address of receive buffer (choice)	13 14		
IN	recvcount	number of elements in receive buffer (non-negative integer)	15 16		
IN	recvtype	data type of receive buffer elements (handle)	17		
IN	root	rank of sending process (integer)	18 19		
IN	comm	communicator (handle)	20		
OUT	request	communication request (handle)	21		
	·		22		
C bindir	ng		23 24		
int MPI_		*sendbuf, int sendcount, MPI_Datatype sendtype,	24 25		
	void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, $_{ m _{2l}}$				
MPI_Comm comm, MPI_Request *request)					
int MPI_	Iscatter_c(const voi	d *sendbuf, MPI_Count sendcount,	28		
		endtype, void *recvbuf, MPI_Count recvcount,	29		
		ecvtype, int root, MPI_Comm comm,	30		
	MPI_Request *r	equest)	31 32		
	2008 binding		33		
MPI_Isca		ount, sendtype, recvbuf, recvcount, recvtype,	34		
TUDE	root, comm, ree	-	35		
		INTENT(IN), ASYNCHRONOUS :: sendbuf sendcount, recvcount, root	36		
		ENT(IN) :: sendtype, recvtype	37		
		ASYNCHRONOUS :: recvbuf	38		
	C(MPI_Comm), INTENT(]		39		
	C(MPI_Request), INTEN		40 41		
INTE	GER, OPTIONAL, INTEN	NT(OUT) :: ierror	42		
MPT Isca	tter(sendbuf, sendco	ount, sendtype, recvbuf, recvcount, recvtype,	43		
	root, comm, re		44		
TYPE		INTENT(IN), ASYNCHRONOUS :: sendbuf	45		
INTE	GER(KIND=MPI_COUNT_P	(IND), INTENT(IN) :: sendcount, recvcount	46		
	• -	ENT(IN) :: sendtype, recvtype	47		
TYPE	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf48				

**Unofficial Draft for Comment Only** 

1	INTEGER, INTENT(IN) :: root			
2	TYPE(MPI_Comm), INTENT(IN) :: comm			
3	TYPE(MPI_Request), INTENT(OUT) :: request			
4	INTE	GER, OPTIONAL, INT	TENT(OUT) :: ierror	
5	Fortran			
6		0	DCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	
7	111 1_1001		REQUEST, IERROR)	
8	<tvn< th=""><th>e&gt; SENDBUF(*), REG</th><th></th></tvn<>	e> SENDBUF(*), REG		
9	• 1		NDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	
10 11		IERROR		
12	This	call starts a nonblock	sing variant of $MPI_SCATTER$ (see Section 6.6).	
13	1 1115	can starts a nonbiotr	ang variant of With_SCATTER (see beetion 0.0).	
14				
15	MPI_ISCA	ATTERV(sendbuf, send	dcounts, displs, sendtype, recvbuf, recvcount, recvtype, root,	
16		comm, request)		
17	IN	sendbuf	address of send buffer (choice, significant only at	
18			root)	
19	IN	sendcounts	non-negative integer array (of length group size)	
20			specifying the number of elements to send to each	
21			rank (significant only at root)	
22	IN	displs	integer array (of length group size). Entry i specifies	
23	IIN	uispis	the displacement (relative to sendbuf) from which to	
24			take the outgoing data to process i (significant only	
25 26			at root)	
27	IN	sendtype	data type of send buffer elements (handle, significant	
28			only at root)	
29 30	OUT	recvbuf	address of receive buffer (choice)	
31	IN	recvcount	number of elements in receive buffer (non-negative	
32			integer)	
33 34	IN	recvtype	data type of receive buffer elements (handle)	
35	IN	root	rank of sending process (integer)	
36	IN	comm	communicator (handle)	
37	OUT	request	communication request (handle)	
38				
39	C binding			
40	int MPI_Iscatterv(const void *sendbuf, const int sendcounts[],			
41	const int displs[], MPI_Datatype sendtype, void *recvbuf,			
42			t, MPI_Datatype recvtype, int root, MPI_Comm comm,	
43		MPI_Request		
44	Las NOT	•	•	
45	int MPL_		void *sendbuf, const MPI_Count sendcounts[],	
46			<pre>nt displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	
47			cvcount, MPI_Datatype recvtype, int root,	
48		MPI_COMM COM	m, MPI_Request *request)	

```
Fortran 2008 binding
                                                                                     1
                                                                                     \mathbf{2}
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
              recvtype, root, comm, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    4
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
                                                                                     5
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                     6
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: recvcount, root
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    10
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    11
                                                                                    12
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                    13
              recvtype, root, comm, request, ierror)
                                                                                    14
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    15
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
                                                                                    16
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                    17
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                    18
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                    19
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                    20
    INTEGER, INTENT(IN) :: root
                                                                                    21
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    24
                                                                                    25
Fortran binding
                                                                                    26
MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
              RECVTYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                    27
                                                                                    28
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
                                                                                    29
               COMM, REQUEST, IERROR
                                                                                    30
                                                                                    31
    This call starts a nonblocking variant of MPI_SCATTERV (see Section 6.6).
                                                                                    32
                                                                                    33
                                                                                    34
                                                                                    35
                                                                                    36
                                                                                    37
                                                                                    38
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
                                                                                    46
                                                                                    47
                                                                                    48
```

```
1
     6.12.5
            Nonblocking Gather-to-all
\mathbf{2}
3
4
     MPI_IALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
5
                    request)
6
       IN
                sendbuf
                                            starting address of send buffer (choice)
7
       IN
                                            number of elements in send buffer (non-negative
8
                sendcount
                                            integer)
9
10
       IN
                sendtype
                                            data type of send buffer elements (handle)
11
       OUT
                recvbuf
                                            address of receive buffer (choice)
12
                                            number of elements received from any process
       IN
                 recvcount
13
                                            (non-negative integer)
14
15
                                            data type of receive buffer elements (handle)
       IN
                 recvtype
16
                                            communicator (handle)
       IN
                comm
17
       OUT
                 request
                                            communication request (handle)
18
19
     C binding
20
     int MPI_Iallgather(const void *sendbuf, int sendcount,
21
                    MPI_Datatype sendtype, void *recvbuf, int recvcount,
22
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
23
^{24}
     int MPI_Iallgather_c(const void *sendbuf, MPI_Count sendcount,
25
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
26
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
27
     Fortran 2008 binding
28
     MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
29
30
                    comm, request, ierror)
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
31
32
         INTEGER, INTENT(IN) :: sendcount, recvcount
33
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
36
         TYPE(MPI_Request), INTENT(OUT) :: request
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
39
                    comm, request, ierror)
40
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
42
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
44
         TYPE(MPI_Comm), INTENT(IN) :: comm
45
         TYPE(MPI_Request), INTENT(OUT) :: request
46
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
```

CHAPTER 6. COLLECTIVE COMMUNICATION

```
<sup>48</sup> Fortran binding
```

MPI_IALLGATHER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,						
	COMM, REQUEST, IERROR) <sup>2</sup>					
	<pre><type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 4</type></pre>					
		5	5			
Th	is call starts a nonblocking	variant of MPI_ALLGATHER (see Section 6.7). $_{6}$	3			
		7				
MPI_IA	LLGATHERV(sendbuf, send comm, request)	count, sendtype, recvbuf, recvcounts, displs, recvtype,	9			
IN	sendbuf	starting address of send buffer (choice) $1$				
IN	sendcount	number of elements in send buffer (non-negative12integer)13				
IN	sendtype	data type of send buffer elements (handle)				
OUT	recvbuf	address of receive buffer (choice)				
IN	recvcounts	non-negative integer array (of length group size)1containing the number of elements that are received14from each process14	8			
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	21 2			
IN	recvtype	data type of receive buffer elements (handle)				
IN	comm	communicator (handle) <sup>24</sup>	5			
OUT	request	communication request (handle)				
		2'				
C bind	U	22				
int MP	-	id *sendbuf, int sendcount,	0			
		ndtype, void *recvbuf, const int recvcounts[], 3	1			
	Const int displ MPI_Request *re	s[], MPI_Datatype recvtype, MPI_Comm comm,	2			
	MP1_Request *ie	autor and a state of the state	3			
int MP		void *sendbuf, MPI_Count sendcount, 34	4			
		ndtype, void *recvbuf, <sup>33</sup>				
		recvcounts[], const MPI_Aint displs[], <sup>34</sup>				
	MPI_Datatype re	cvtype, MPI_Comm comm, MPI_Request *request) 3				
Fortra	n 2008 binding	3:				
MPI_Ia	llgatherv(sendbuf, send	dcount, sendtype, recvbuf, recvcounts, displs, ${4\over4}$				
	recvtype, comm,	request, ierror)				
		INTENT(IN), ASYNCHRONOUS :: sendbuf				
	TEGER, INTENT(IN) :: se	4.	3			
		NT(IN) :: sendtype, recvtype	4			
		ASYNCHRONOUS :: recvbuf	5			
		<pre>VCHRONOUS :: recvcounts(*), displs(*) 44</pre>	6			
	PE(MPI_Comm), INTENT(IN PE(MPI_Bogwogt) INTENT	4	7			
1 Y.	TYPE(MPI_Request), INTENT(OUT) :: request 48					

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
3
                    recvtype, comm, request, ierror)
4
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
6
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
9
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS A: displs(*)
10
          TYPE(MPI_Comm), INTENT(IN) :: comm
11
          TYPE(MPI_Request), INTENT(OUT) :: request
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
16
                    RECVTYPE, COMM, REQUEST, IERROR)
17
          <type> SENDBUF(*), RECVBUF(*)
18
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
19
                     REQUEST, IERROR
20
         This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 6.7).
21
22
     6.12.6 Nonblocking All-to-All Scatter/Gather
23
24
25
     MPI_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)
26
27
28
       IN
                 sendbuf
                                            starting address of send buffer (choice)
29
       IN
                sendcount
                                            number of elements sent to each process
30
                                            (non-negative integer)
31
       IN
                 sendtype
                                            data type of send buffer elements (handle)
32
33
       OUT
                                            address of receive buffer (choice)
                 recvbuf
34
       IN
                 recvcount
                                            number of elements received from any process
35
                                            (non-negative integer)
36
       IN
                 recvtype
                                            data type of receive buffer elements (handle)
37
38
       IN
                 comm
                                            communicator (handle)
39
       OUT
                 request
                                            communication request (handle)
40
41
     C binding
42
     int MPI_Ialltoall(const void *sendbuf, int sendcount,
43
                    MPI_Datatype sendtype, void *recvbuf, int recvcount,
44
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
45
46
     int MPI_Ialltoall_c(const void *sendbuf, MPI_Count sendcount,
47
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
48
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

```
Fortran 2008 binding
                                                                                      \mathbf{2}
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
              comm, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                      4
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                      10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      11
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                      12
              comm, request, ierror)
                                                                                      13
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                      14
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                      15
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                      16
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                      17
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                      18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                      19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      20
                                                                                     21
Fortran binding
MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
                                                                                     22
                                                                                     23
              COMM, REQUEST, IERROR)
                                                                                      ^{24}
    <type> SENDBUF(*), RECVBUF(*)
                                                                                     25
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
                                                                                     26
    This call starts a nonblocking variant of MPI_ALLTOALL (see Section 6.8).
                                                                                     27
                                                                                     28
                                                                                     29
                                                                                      30
                                                                                      31
                                                                                      32
                                                                                      33
                                                                                     34
                                                                                     35
                                                                                      36
                                                                                     37
                                                                                      38
                                                                                      39
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
                                                                                      45
                                                                                      46
                                                                                      47
                                                                                      48
```

12	MPI_IALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request)		
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6 7	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank
8 9 10	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
11 12	IN	sendtype	data type of send buffer elements (handle)
13	OUT	recvbuf	address of receive buffer (choice)
14 15 16	IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank
17 18 19 20	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
21	IN	recvtype	data type of receive buffer elements (handle)
22	IN	comm	communicator (handle)
23 24	OUT	request	communication request (handle)
25 26 27 28 29 30 31	C binding int MPI_Ialltoallv(const void *sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)		
32 33 34 35 36 37	<pre>int MPI_Ialltoallv_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>		
38	<pre>MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request, ierror)</pre>		
39 40	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf		
41	<pre>INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),</pre>		
42	recvcounts(*), rdispls(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype		
43 44	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf		
45	TYPE(	<pre>MPI_Comm), INTENT(IN) ::</pre>	comm
46		MPI_Request), INTENT(OUT)	-
47 48	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: lerror

MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,	1
rdispls, recvtype, comm, request, ierror)	2
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	3
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::	4
<pre>sendcounts(*), recvcounts(*)</pre>	5
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),</pre>	6
rdispls(*)	7
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	8
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	9
TYPE(MPI_Comm), INTENT(IN) :: comm	10
TYPE(MPI_Request), INTENT(OUT) :: request	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
	13
Fortran binding	14
MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,	15
RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)	16
<type> SENDBUF(*), RECVBUF(*)</type>	17
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	18
RECVTYPE, COMM, REQUEST, IERROR	19
This call starts a nonblocking variant of $MPI_ALLTOALLV$ (see Section 6.8).	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30
	31
	32
	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46

1			
3		recvtypes, comm, request	, ,
4	IN	sendbuf	starting address of send buffer (choice)
5 6 7	IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)
8 9 10 11	IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
12 13 14 15	IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
16	OUT	recvbuf	address of receive buffer (choice)
17 18 19	IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers)
20 21 22 23 24	IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
25 26 27	IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
28	IN	comm	communicator (handle)
29 30	OUT	request	communication request (handle)
31	C binding	F	
32	-		ndbuf, const int sendcounts[],
33 34	_		<pre>const MPI_Datatype sendtypes[],</pre>
35		void *recvbuf, const	<pre>int recvcounts[], const int rdispls[],</pre>
36		const MPI_Datatype r	ecvtypes[], MPI_Comm comm,
37		MPI_Request *request	)
38	int MPI I	alltoallw c(const void *s	sendbuf, const MPI_Count sendcounts[],
39			ls[], const MPI_Datatype sendtypes[],
40		-	<pre>MPI_Count recvcounts[],</pre>
41		const MPI_Aint rdisp	<pre>ls[], const MPI_Datatype recvtypes[],</pre>
42		MPI_Comm comm, MPI_R	equest *request)
43 44	Fortran 2	008 binding	
44		0	s, sdispls, sendtypes, recvbuf,
46		recvcounts, rdispls,	recvtypes, comm, request, ierror)
47	TYPE(	*), DIMENSION(), INTENT	C(IN), ASYNCHRONOUS :: sendbuf
48			

```
1
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                     2
               recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
               recvtypes(*)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                     10
              recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                     11
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                     12
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                     13
               sendcounts(*), recvcounts(*)
                                                                                     14
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                     15
               rdispls(*)
                                                                                     16
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*)
                                                                                     17
               recvtypes(*)
                                                                                     18
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                     19
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     20
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    22
                                                                                    23
Fortran binding
                                                                                     ^{24}
MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                                                                                    25
              RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
                                                                                     26
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
                                                                                    27
               RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
                                                                                    28
                                                                                    29
    This call starts a nonblocking variant of MPI_ALLTOALLW (see Section 6.8).
                                                                                     30
                                                                                     31
                                                                                     32
                                                                                     33
                                                                                    34
                                                                                    35
                                                                                    36
                                                                                    37
                                                                                     38
                                                                                     39
                                                                                     40
                                                                                     41
                                                                                     42
                                                                                     43
                                                                                     44
                                                                                     45
                                                                                     46
                                                                                     47
```

```
264
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.12.7 Nonblocking Reduce
\mathbf{2}
3
4
     MPI_IREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, request)
5
       IN
                sendbuf
                                            address of send buffer (choice)
6
       OUT
                recvbuf
7
                                            address of receive buffer (choice, significant only at
8
                                            root)
9
       IN
                count
                                            number of elements in send buffer (non-negative
10
                                            integer)
11
                                            data type of elements of send buffer (handle)
       IN
                datatype
12
       IN
                                            reduce operation (handle)
13
                ор
14
       IN
                root
                                            rank of root process (integer)
15
       IN
                comm
                                            communicator (handle)
16
17
       OUT
                                            communication request (handle)
                request
18
19
     C binding
20
     int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,
21
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
22
                    MPI_Request *request)
23
     int MPI_Ireduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
24
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
25
                    MPI_Request *request)
26
27
     Fortran 2008 binding
28
     MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
29
                    ierror)
30
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
31
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
32
         INTEGER, INTENT(IN) :: count, root
33
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
         TYPE(MPI_Op), INTENT(IN) :: op
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         TYPE(MPI_Request), INTENT(OUT) :: request
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
39
                    ierror)
40
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
41
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
42
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         TYPE(MPI_Op), INTENT(IN) :: op
45
         INTEGER, INTENT(IN) :: root
46
         TYPE(MPI_Comm), INTENT(IN) :: comm
47
         TYPE(MPI_Request), INTENT(OUT) :: request
48
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 Fortran binding MPI\_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(\*), RECVBUF(\*) INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR This call starts a nonblocking variant of  $MPI_REDUCE$  (see Section 6.9.1). 9 Advice to implementors. The implementation is explicitly allowed to use different 10 algorithms for blocking and nonblocking reduction operations that might change the 11 order of evaluation of the operations. However, as for MPL\_REDUCE, it is strongly 12recommended that MPI\_IREDUCE be implemented so that the same result be obtained 13 whenever the function is applied on the same arguments, appearing in the same order. 14Note that this may prevent optimizations that take advantage of the physical location 1516of processes. (End of advice to implementors.) 17Advice to users. For operations which are not truly associative, the result delivered 18 upon completion of the nonblocking reduction may not exactly equal the result deliv-19 ered by the blocking reduction, even when specifying the same arguments in the same 20order. (End of advice to users.) 2122 6.12.8 Nonblocking All-Reduce 23 $^{24}$ 2526MPI\_IALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, request) 27IN sendbuf starting address of send buffer (choice) 28OUT recvbuf starting address of receive buffer (choice) 29 30 IN number of elements in send buffer (non-negative count 31integer) 32 IN data type of elements of send buffer (handle) datatype 33 IN operation (handle) 34 op 35IN comm communicator (handle) 36 OUT communication request (handle) request 37 38 C binding 39 int MPI\_Iallreduce(const void \*sendbuf, void \*recvbuf, int count, 40 MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm, 41 MPI\_Request \*request) 4243 int MPI\_Iallreduce\_c(const void \*sendbuf, void \*recvbuf, MPI\_Count count, 44 MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm, 45MPI\_Request \*request) 46Fortran 2008 binding 4748

```
1
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
\mathbf{2}
                    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
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
12
                    ierror)
13
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
14
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
15
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
16
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
17
          TYPE(MPI_Op), INTENT(IN) :: op
18
          TYPE(MPI_Comm), INTENT(IN) :: comm
19
          TYPE(MPI_Request), INTENT(OUT) :: request
20
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     Fortran binding
23
     MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
24
                    IERROR)
25
          <type> SENDBUF(*), RECVBUF(*)
26
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
27
         This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 6.9.6).
28
29
     6.12.9
             Nonblocking Reduce-Scatter with Equal Blocks
30
^{31}
32
     MPI_IREDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm,
33
34
                    request)
35
                 sendbuf
       IN
                                            starting address of send buffer (choice)
36
       OUT
                 recvbuf
                                            starting address of receive buffer (choice)
37
38
       IN
                 recvcount
                                            element count per block (non-negative integer)
39
                 datatype
                                            data type of elements of send and receive buffers
       IN
40
                                            (handle)
41
       IN
                 ор
                                            operation (handle)
42
43
       IN
                 comm
                                            communicator (handle)
44
       OUT
                 request
                                            communication request (handle)
45
46
     C binding
47
48
```

```
1
int MPI_Ireduce_scatter_block(const void *sendbuf, void *recvbuf,
                                                                                    2
              int recvcount, MPI_Datatype datatype, MPI_Op op,
              MPI_Comm comm, MPI_Request *request)
int MPI_Ireduce_scatter_block_c(const void *sendbuf, void *recvbuf,
                                                                                    5
              MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,
                                                                                    6
              MPI_Comm comm, MPI_Request *request)
Fortran 2008 binding
                                                                                    9
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                    10
              request, ierror)
                                                                                    11
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    12
                                                                                    13
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                    14
                                                                                    15
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                    16
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    17
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    19
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                   20
              request, ierror)
                                                                                   21
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   22
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   23
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                    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
                                                                                   30
Fortran binding
                                                                                    31
MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
                                                                                    32
              REQUEST, IERROR)
                                                                                    33
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   34
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
                                                                                   35
    This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Sec-
                                                                                   36
tion 6.10.1).
                                                                                   37
                                                                                    38
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
                                                                                    46
                                                                                    47
                                                                                    48
```

```
268
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.12.10 Nonblocking Reduce-Scatter
\mathbf{2}
3
4
     MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)
5
       IN
                sendbuf
                                            starting address of send buffer (choice)
6
       OUT
                recvbuf
7
                                            starting address of receive buffer (choice)
8
       IN
                                            non-negative integer array specifying the number of
                recvcounts
9
                                            elements in result distributed to each process. This
10
                                            array must be identical on all calling processes.
11
                                            data type of elements of input buffer (handle)
       IN
                datatype
12
       IN
                                            operation (handle)
13
                ор
14
       IN
                comm
                                            communicator (handle)
15
                                            communication request (handle)
       OUT
                request
16
17
     C binding
18
     int MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf,
19
                    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
20
                    MPI_Comm comm, MPI_Request *request)
21
22
     int MPI_Ireduce_scatter_c(const void *sendbuf, void *recvbuf,
23
                    const MPI_Count recvcounts[], MPI_Datatype datatype,
24
                    MPI_Op op, MPI_Comm comm, MPI_Request *request)
25
     Fortran 2008 binding
26
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
27
                    request, ierror)
28
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
29
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
30
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Op), INTENT(IN) :: op
33
          TYPE(MPI_Comm), INTENT(IN) :: comm
34
         TYPE(MPI_Request), INTENT(OUT) :: request
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
38
                    request, ierror)
39
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
40
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         TYPE(MPI_Op), INTENT(IN) :: op
44
         TYPE(MPI_Comm), INTENT(IN) :: comm
45
         TYPE(MPI_Request), INTENT(OUT) :: request
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
     Fortran binding
48
```

MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, REQUEST, IERROR) <sup>1</sup>					
	<type> SENDBUF(*), RECVBUF(*) 3</type>				
INT	TEGER RECVCOUNTS(*), DATATY		4 5		
Thi	s call starts a nonblocking variant	nt of MPI_REDUCE_SCATTER (see Section $6.10.2$ ).	6 7		
6.12.11	Nonblocking Inclusive Scan		8		
			9		
	CAN(sendbuf, recvbuf, count, data		10 11		
	<b>X</b> · · · · ·		12		
IN	sendbuf		13		
OUT	recvbuf		14		
IN	count	integer)	15 16		
IN	datatype	data type of elements of input buffer (handle)	17 18		
IN	ор	operation (handle)	19		
IN	comm	communicator (handle)	20		
OUT	request	communication request (handle)	21		
			22 23		
C bind	6		24		
int MP]		, void *recvbuf, int count,	25		
	MPI_Datatype datatyp MPI_Request *request	)	26		
int MD		2	27 28		
int MPI	<pre>int MPI_Iscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>				
	MPI_Request *request		30		
Fortrai	1 2008 binding		31		
		datatype op comm request jerror)	32		
		T(TN) ASYNCHRONOUS ·· sendbuf	$33 \\ 34$		
	PE(*), DIMENSION(), ASYNC	HRONOUS :: recvbuf	35		
	TEGER, INTENT(IN) :: count		36		
	PE(MPI_Datatype), INTENT(IN) PE(MPI_Op), INTENT(IN) :: op		37		
	PE(MPI_Comm), INTENT(IN) ::	COMM	38		
TYF	TYPE(MPI_Request), INTENT(OUT) :: request       39				
INT	TEGER, OPTIONAL, INTENT(OUT)	) ·· jerror	41		
MPI_Iso	can(sendbuf, recvbuf, count	, datatype, op, comm, request, ierror) 4	42		
		I(IN), ASINGINGNOUS Sendbul	43		
	PE(*), DIMENSION(), ASYNC		$44 \\ 45$		
	<pre>TEGER(KIND=MPI_COUNT_KIND), PE(MPI_Datatype), INTENT(IN)</pre>		$45 \\ 46$		
	PE(MPI_Datatype), INTENI(IN) PE(MPI_Op), INTENT(IN) :: op	uatatype	47		
	TYPE(MPI_Comm), INTENT(IN) :: comm 48				

$\frac{1}{2}$	TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
3	Fortran binding		
4			NT, DATATYPE, OP, COMM, REQUEST, IERROR)
5		<pre>we&gt; SENDBUF(*), RECVBUF(</pre>	
6			, COMM, REQUEST, IERROR
7			
8 9	This	call starts a nonblocking va	riant of $MPI_SCAN$ (see Section 6.11).
10 11	6.12.12	Nonblocking Exclusive Sca	n
12 13			
14	MPI_IEXS	SCAN(sendbuf, recvbuf, coun	t, datatype, op, comm, request)
15	IN	sendbuf	starting address of send buffer (choice)
16	OUT	recvbuf	starting address of receive buffer (choice)
17	IN	count	number of elements in input buffer (non-negative
18 19			integer)
20	IN	datatype	data type of elements of input buffer (handle)
21	IN	ор	operation (handle)
22	IN	comm	intra-communicator (handle)
23 24	OUT	request	communication request (handle)
25			
26	C bindir	ng	
27	int MPI_	Iexscan(const void *sen	dbuf, void *recvbuf, int count,
28	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,		
29 30	MPI_Request *request)		
30 31	<pre>int MPI_Iexscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>		
32			cype, MPI_Op op, MPI_Comm comm,
33		MPI_Request *reque	est)
34	Fortran	2008 binding	
35	MPI_Iexs	can(sendbuf, recvbuf, c	count, datatype, op, comm, request, ierror)
36	TYPE	C(*), DIMENSION(), INT	ENT(IN), ASYNCHRONOUS :: sendbuf
37		<pre>:(*), DIMENSION(), ASY</pre>	
38		GER, INTENT(IN) :: coum	
39 40		C(MPI_Datatype), INTENT(	V 1
41	TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: comm		
42		C(MPI_Request), INTENT(0	
43		GER, OPTIONAL, INTENT(O	-
44			
45			count, datatype, op, comm, request, ierror)
46		C(*), DIMENSION(), INI C(*), DIMENSION(), ASY	ENT(IN), ASYNCHRONOUS :: sendbuf
47		GER(KIND=MPI_COUNT_KIND	
48			., .,

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype

TYPE(MPI_Op), INTENT(IN) :: op

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

TYPE(MPI_Request), INTENT(OUT) :: request

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding
```

This call starts a nonblocking variant of MPI\_EXSCAN (see Section 6.11.2).

### 6.13 Persistent Collective Operations

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 [52, 41] 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 its blocking and nonblocking counterparts for intra-communicators and inter-communicators after completion. Likewise, upon completion, persistent collective reduction operations perform the same operation as their blocking and nonblocking counterparts, and the same restrictions and recommendations on reduction orders apply (see also Section 6.9.1).

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

According to the definitions in Section 2.4.2, the persistent collective initialization 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 associated with the specified communicator.

Advice to users. This is one of the exceptions in which incomplete procedures are non-local and therefore blocking. (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

#### Unofficial Draft for Comment Only

1

2

3

4

5 6

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

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.)

- 12A request must be inactive when it is started. Starting the operation makes the request 13 active. Once any process starts a persistent collective operation, it must complete that 14operation and all other processes in the communicator must eventually start (and complete) 15the same persistent collective operation. Persistent collective operations cannot be matched 16with blocking or nonblocking collective operations. Completion of a persistent collective 17operation makes the corresponding request inactive. After starting a persistent collective 18 operation, all associated send buffers must not be modified and all associated receive buffers 19 must not be accessed until the corresponding persistent request is completed. 20
  - Completing a persistent collective request, for example using MPI\_TEST or 21MPI\_WAIT, makes it inactive, but does not free the request. This is the same behavior as 22 for persistent point-to-point requests. Inactive persistent collective requests can be freed 23using MPI\_REQUEST\_FREE. It is erroneous to free an active persistent collective request. 24Persistent collective operations cannot be canceled; it is erroneous to use MPI\_CANCEL on 25a persistent collective request. 26

For every nonblocking collective communication operation in MPI, there is a corresponding persistent collective operation with the analogous API signature. [mpicdeck[MPI] Info]

#-error! Done

```
27
         28
#-other 29
                   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
         30
               operations may be optimized during communicator creation or by the initialization opera-
         ^{31}
               tion of an individual persistent collective. Note that communicator-scoped hints should be
         32
               provided using MPI_COMM_SET_INFO while, for operation-scoped hints, they are supplied
         33
               to the persistent collective communication initialization functions using the info argument.
         34
         35
                       Persistent Barrier Synchronization
               6.13.1
         36
         37
         38
         39
               MPI_BARRIER_INIT(comm, info, request)
         40
                 IN
                                                         communicator (handle)
                           comm
         41
                 IN
                           info
                                                         info argument (handle)
         42
         43
                 OUT
                           request
                                                         communication request (handle)
         44
         45
               C binding
         46
               int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)
         47
```

Fortran 2008 binding 48

1

2

3

4

5

6

7

8

9

10

	<pre>MPI_Barrier_init(comm, info, request, ierror)</pre>					
TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Info), INTENT(IN) :: info						
	TYPE(MPI_Request), INTENT(OUT) :: request					
	-	, INTENT(OUT) :: ierror	5			
Fortron	hinding		6			
Fortran MPT BARR		, INFO, REQUEST, IERROR)	7			
		), REQUEST, IERROR	8			
			9 10			
Creat	tes a persistent o	collective communication request for the barrier operation.	11			
6130 E	Persistent Broad	cast	12			
0.13.2 1	ersistent Dioau	Cast	13			
			14			
MPI_BCA	ST_INIT(buffer,	count, datatype, root, comm, info, request)	$15 \\ 16$			
INOUT	buffer	starting address of buffer (choice)	10			
IN	count	number of entries in buffer (non-negative integer)	18			
			19			
IN	datatype	data type of buffer (handle)	20			
IN	root	rank of broadcast root (integer)	21			
IN	comm	communicator (handle)	22			
IN	info	info argument (handle)	23 24			
OUT	request	communication request (handle)	25			
	·		26			
C bindin	ıg		27			
int MPI_		id *buffer, int count, MPI_Datatype datatype,	28			
	int root	, MPI_Comm comm, MPI_Info info, MPI_Request *request)	29			
int MPI_	Bcast_init_c(	void *buffer, MPI_Count count, MPI_Datatype datatype,	30 31			
	int root	, MPI_Comm comm, MPI_Info info, MPI_Request *request)	32			
Fortran	2008 binding		33			
	U	, count, datatype, root, comm, info, request, ierror)	34			
		N(), ASYNCHRONOUS :: buffer	35			
		N) :: count, root	36			
		), INTENT(IN) :: datatype	37			
		NTENT(IN) :: comm	38 39			
		NTENT(IN) :: info , INTENT(OUT) :: request	40			
		, INTENI(OUT) :: ierror	41			
			42			
		, count, datatype, root, comm, info, request, ierror)	43			
		N(), ASYNCHRONOUS :: buffer COUNT_KIND), INTENT(IN) :: count	44			
		), INTENT(IN) :: datatype	45			
	GER, INTENT(IN		46 47			
	TYPE(MPI_Comm), INTENT(IN) :: comm       41					

1 2 3	TYPE(	TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
4 5 6 7 8	Fortran k MPI_BCASI <type< th=""><th colspan="3">Fortran binding MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR) <type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR</type></th></type<>	Fortran binding MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR) <type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR</type>		
9 10	Create	es a persistent collective com	nunication request for the broadcast operation.	
10 11 12 13	6.13.3 P	ersistent Gather		
14 15 16	MPI_GATH	HER_INIT(sendbuf, sendcount, info, request)	sendtype, recvbuf, recvcount, recvtype, root, comm,	
17	IN	sendbuf	starting address of send buffer (choice)	
18 19	IN	sendcount	number of elements in send buffer (non-negative integer)	
20 21	IN	sendtype	data type of send buffer elements (handle)	
22 23	OUT	recvbuf	address of receive buffer (choice, significant only at root)	
24 25	IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)	
26 27 28	IN	recvtype	data type of recv buffer elements (handle, significant only at root)	
29	IN	root	rank of receiving process (integer)	
30	IN	comm	communicator (handle)	
31 32	IN	info	info argument (handle)	
33 34 35	OUT C binding	request	communication request (handle)	
36 37 38 39	<pre>int MPI_Gather_init(const void *sendbuf, int sendcount,</pre>			
40 41 42 43 44	int MPI_0	<pre>int MPI_Gather_init_c(const void *sendbuf, MPI_Count sendcount,</pre>		
45 46 47 48	MPI_Gathe	root, comm, info, re	t, sendtype, recvbuf, recvcount, recvtype, quest, ierror) T(IN), ASYNCHRONOUS :: sendbuf	

1 INTEGER, INTENT(IN) :: sendcount, recvcount, root TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 2 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_Gather\_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 9 root, comm, info, request, ierror) 10 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 11 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 12TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 13 TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 14INTEGER, INTENT(IN) :: root 15TYPE(MPI\_Comm), INTENT(IN) :: comm 16TYPE(MPI\_Info), INTENT(IN) :: info 17 TYPE(MPI\_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 Fortran binding 20MPI\_GATHER\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 2122 ROOT, COMM, INFO, REQUEST, IERROR) 23<type> SENDBUF(\*), RECVBUF(\*)  $^{24}$ INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, 25REQUEST, IERROR 26Creates a persistent collective communication request for the gather operation. 272829 30 31 32 33 34 3536 37 38 39 40 41 4243 44 45464748

12	MPI_GATH	IERV_INIT(sendbuf, sendcount root, comm, info, request	t, sendtype, recvbuf, recvcounts, displs, recvtype, t)			
3	IN	sendbuf	starting address of send buffer (choice)			
4 5 6	IN	sendcount	number of elements in send buffer (non-negative integer)			
7	IN	sendtype	data type of send buffer elements (handle)			
8 9 10	OUT	recvbuf	address of receive buffer (choice, significant only at root)			
11 12 13	IN	recvcounts	non-negative integer array (of length group size) containing 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 <b>recvbuf</b> 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 (handle, significant only at root)			
21	IN	root	rank of receiving process (integer)			
22	IN	comm	communicator (handle)			
23 24	IN	info	info argument (handle)			
25	OUT	request	communication request (handle)			
26 27 28 29 30 31 32 33 34 35 36	C binding int MPI_Gatherv_init(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_Info info, MPI_Request *request) int MPI_Gatherv_init_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, const MPI_Count recvcounts[], const MPI_Aint displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,					
37		MPI_Request *request)				
38 39		Fortran 2008 binding				
40	riri_Gatue	<pre>MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,</pre>				
41	TYPE(	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf				
42		INTEGER, INTENT(IN) :: sendcount, root				
43 44		TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype				
45		TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)				
46		TYPE(MPI_Comm), INTENT(IN) :: comm				
47		<pre>MPI_Info), INTENT(IN) ::</pre>				
48	TYPE(	MPI_Request), INTENT(OUT)	) :: request			

INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI\_Gatherv\_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, info, request, ierror) 4 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 5 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount 6 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(\*) a INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN), ASYNCHRONOUS :: displs(\*) 10 INTEGER, INTENT(IN) :: root 11 TYPE(MPI\_Comm), INTENT(IN) :: comm 12TYPE(MPI\_Info), INTENT(IN) :: info 13 TYPE(MPI\_Request), INTENT(OUT) :: request 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516Fortran binding 17MPI\_GATHERV\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 18 RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) 19 <type> SENDBUF(\*), RECVBUF(\*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, ROOT, 2021COMM, INFO, REQUEST, IERROR 22 Creates a persistent collective communication request for the gathery operation. 23246.13.4 Persistent Scatter 252627MPI\_SCATTER\_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, 28comm, info, request) 2930 IN sendbuf address of send buffer (choice, significant only at 31root) 32 IN sendcount number of elements sent to each process 33 (non-negative integer, significant only at root) 34 IN data type of send buffer elements (handle, significant sendtype 35 only at root) 36 37 OUT recvbuf address of receive buffer (choice) 38 IN recvcount number of elements in receive buffer (non-negative 39 integer) 40 IN data type of receive buffer elements (handle) recvtype 41 42IN root rank of sending process (integer) 43 IN comm communicator (handle) 44 IN info info argument (handle) 45OUT request communication request (handle) 4647

#### C binding

```
278
```

```
1
     int MPI_Scatter_init(const void *sendbuf, int sendcount,
\mathbf{2}
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
3
                   MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
4
                   MPI_Request *request)
5
     int MPI_Scatter_init_c(const void *sendbuf, MPI_Count sendcount,
6
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
7
                   MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
8
                   MPI_Request *request)
9
10
     Fortran 2008 binding
11
     MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
12
                   recvtype, root, comm, info, request, ierror)
13
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
14
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
15
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
16
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
17
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         TYPE(MPI_Info), INTENT(IN) :: info
19
         TYPE(MPI_Request), INTENT(OUT) :: request
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
22
                   recvtype, root, comm, info, request, ierror)
23
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
24
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
25
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
26
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
27
         INTEGER, INTENT(IN) :: root
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
     Fortran binding
34
     MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
35
                   RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
36
         <type> SENDBUF(*), RECVBUF(*)
37
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
38
                    REQUEST, IERROR
39
         Creates a persistent collective communication request for the scatter operation.
40
41
42
43
44
45
46
47
48
```

MPI_SCATTERV_INIT(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, <sup>1</sup> root, comm, info, request) <sup>2</sup>						
IN	sendbuf	address of send buffer (choice, significant only at root)	3 4 5			
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)	5 6 7 8			
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 (significant only at root)	9 10 11 12			
IN	sendtype	data type of send buffer elements (handle, significant only at root)	13 14 15			
OUT	recvbuf	address of receive buffer (choice, significant only at root)	16 17			
IN	recvcount	number of elements in receive buffer (non-negative integer)	18 19			
IN	recvtype	data type of receive buffer elements (handle)	20 21			
IN	root	rank of sending process (integer)	21			
IN	comm	communicator (handle)	23			
IN	info	info argument (handle)	24			
OUT	request	communication request (handle)	25 26			
			27			
C binding			28			
int MPI_So		sendbuf, const int sendcounts[],	29			
	_	<pre>IPI_Datatype sendtype, void *recvbuf, tatype recvtype, int root, MPI_Comm comm,</pre>	30 31			
	MPI_Info info, MPI_Re		32			
int MDT C			33			
<pre>int MPI_Scatterv_init_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>						
		MPI_Datatype recvtype, int root,	35			
		fo info, MPI_Request *request)	36 37			
Fortran 2	008 binding		38			
		nts, displs, sendtype, recvbuf,	39			
		root, comm, info, request, ierror)	40			
TYPE(>	v 1	(IN), ASYNCHRONOUS :: sendbuf	41			
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*) 4						
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype						
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf						
INTEGER, INTENT(IN) :: recvcount, root TYPE(MPI_Comm), INTENT(IN) :: comm						
TYPE(MPI_COMM), INTENT(IN) :: COMM TYPE(MPI_Info), INTENT(IN) :: info						
TYPE(MPI_Info), INTENT(IN) :: info       47         TYPE(NDI_D)       INTENT(OUT)						

TYPE(MPI\_Request), INTENT(OUT) :: request

1	INT	EGER, OPTIONAL,	INTENT(OUT) :: ierror			
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,</pre>					
15						
10 17 18 19 20 21	<pre>Fortran binding MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,</pre>					
22	Cre	Creates a persistent collective communication request for the scattery operation.				
23 24	010		request for the seattery operation.			
24 25 26	6.13.5	Persistent Gather-t	o-all			
27 28 29	MPI_AL	LGATHER_INIT(sen info, request	dbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, )			
30	IN	sendbuf	starting address of send buffer (choice)			
31 32 33	IN	sendcount	number of elements in send buffer (non-negative integer)			
34	IN	sendtype	data type of send buffer elements (handle)			
35	OUT	recvbuf	address of receive buffer (choice)			
36 27	IN	recvcount	number of elements received from any process			
37 38			(non-negative integer)			
39	IN	recvtype	data type of receive buffer elements (handle)			
40	IN	comm	communicator (handle)			
41 42	IN	info	info argument (handle)			
43	OUT	request	communication request (handle)			
44						
45	C bind	ing				
46	<pre>int MPI_Allgather_init(const void *sendbuf, int sendcount,</pre>					
47 48	MPI_Datatype sendtype, void *recvbuf, int recvcount,					
-						

```
1
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
                                                                                    2
              MPI_Request *request)
int MPI_Allgather_init_c(const void *sendbuf, MPI_Count sendcount,
              MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
                                                                                    5
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
                                                                                    6
              MPI_Request *request)
Fortran 2008 binding
                                                                                    9
MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                    10
              recvtype, comm, info, request, ierror)
                                                                                    11
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                    12
                                                                                    13
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                    14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    16
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                    17
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    19
MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                    20
              recvtype, comm, info, request, ierror)
                                                                                   21
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   22
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                   23
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                    24
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    26
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                    27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    29
                                                                                    30
Fortran binding
                                                                                    31
MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                    32
              RECVTYPE, COMM, INFO, REQUEST, IERROR)
                                                                                    33
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   34
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
                                                                                   35
               IERROR
                                                                                    36
    Creates a persistent collective communication request for the allgather operation.
                                                                                   37
                                                                                    38
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
                                                                                    46
                                                                                    47
                                                                                    48
```

12	MPI_ALL(	MPI_ALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, info, request)				
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)			
4 5 6	IN	sendcount	number of elements in send buffer (non-negative integer)			
7	IN	sendtype	data type of send buffer elements (handle)			
8	OUT	recvbuf	address of receive buffer (choice)			
9 10 11 12	IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process			
13 14 15	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to <b>recvbuf</b> ) at which to place the incoming data from process i			
16	IN	recvtype	data type of receive buffer elements (handle)			
17 18	IN	comm	communicator (handle)			
19	IN	info	info argument (handle)			
20 21	OUT	request	communication request (handle)			
24 25 26 27 28 29 30 31 32 33	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request) int MPI_Allgatherv_init_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, const MPI_Count recvcounts[], const MPI_Aint displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>					
34		Fortran 2008 binding				
35	MPI_Allga	MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,				
36	TYPE.	<pre>displs, recvtype, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf</pre>				
37		INTEGER, INTENT(IN) :: sendcount				
38 39		TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype				
40		TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf				
41		<pre>INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)</pre>				
42		TYPE(MPI_Comm), INTENT(IN) :: comm				
43		TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request				
44		GER, OPTIONAL, INTENT(C	-			
45 46						
47	MPT_ATTE:		endcount, sendtype, recvbuf, recvcounts, comm, info, request, ierror)			
48	TYPE		TENT(IN), ASYNCHRONOUS :: sendbuf			

1 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount  $\mathbf{2}$ TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 3 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(\*) 4 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN), ASYNCHRONOUS :: displs(\*) 56 TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 Fortran binding 11 MPI\_ALLGATHERV\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 12DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 13 <type> SENDBUF(\*), RECVBUF(\*) 14 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 15INFO, REQUEST, IERROR 1617Creates a persistent collective communication request for the allgathery operation. 18 19 6.13.6 Persistent All-to-All Scatter/Gather 202122 MPI\_ALLTOALL\_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, 23info, request) 24IN sendbuf starting address of send buffer (choice) 2526IN sendcount number of elements sent to each process 27(non-negative integer) 28sendtype data type of send buffer elements (handle) IN 29 OUT recvbuf address of receive buffer (choice) 30 31number of elements received from any process IN recvcount 32 (non-negative integer) 33 IN data type of receive buffer elements (handle) recvtype 34 1N comm communicator (handle) 35IN info info argument (handle) 36 37 OUT request communication request (handle) 38 39 C binding 40 int MPI\_Alltoall\_init(const void \*sendbuf, int sendcount, 41 MPI\_Datatype sendtype, void \*recvbuf, int recvcount, 42MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Info info, 43 MPI\_Request \*request) 44 45int MPI\_Alltoall\_init\_c(const void \*sendbuf, MPI\_Count sendcount, MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 4647MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Info info, 48 MPI\_Request \*request)

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
3
                   recvtype, comm, info, request, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         INTEGER, INTENT(IN) :: sendcount, recvcount
6
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         TYPE(MPI_Info), INTENT(IN) :: info
10
         TYPE(MPI_Request), INTENT(OUT) :: request
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
13
                   recvtype, comm, info, request, ierror)
14
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
15
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
16
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
17
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
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
     Fortran binding
24
     MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
25
                   RECVTYPE, COMM, INFO, REQUEST, IERROR)
26
         <type> SENDBUF(*), RECVBUF(*)
27
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
28
                    IERROR
29
         Creates a persistent collective communication request for the alltoall operation.
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_ALLT	•	uf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, m, info, request)	1 2	
IN	sendbuf	starting address of send buffer (choice)	3	
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	4 5 6 7	
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	8 9 10	
IN	sendtype	data type of send buffer elements (handle)	11 12	
OUT	recvbuf	address of receive buffer (choice)	12	
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank	14 15 16	
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	17 18 19 20	
IN	recvtype	data type of receive buffer elements (handle)	21	
IN	comm	communicator (handle)	22	
IN	info	info argument (handle)	23	
OUT	request	communication request (handle)	24 25	
			26	
C binding	g		27	
int MPI_A	lltoallv_init(co	onst void *sendbuf, const int sendcounts[],	28	
		<pre>displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	29	
		ecvcounts[], const int rdispls[],	30	
	MPI_Datatyp MPI_Request	e recvtype, MPI_Comm comm, MPI_Info info,	31 32	
	Mr1_kequest	*request)	33	
int MPI_A		<pre>(const void *sendbuf, const MPI_Count sendcounts[],</pre>	34	
		<pre>int sdispls[], MPI_Datatype sendtype,</pre>	35	
		uf, const MPI_Count recvcounts[],	36	
		int rdispls[], MPI_Datatype recvtype, mm, MPI_Info info, MPI_Request *request)	37	
		mm, in i_inio inio, in i_nequest wiequest,	38	
	008 binding		39	
MPI_AIIto		if, sendcounts, sdispls, sendtype, recvbuf,	40 41	
TVDF		<pre>rdispls, recvtype, comm, info, request, ierror) ), INTENT(IN), ASYNCHRONOUS :: sendbuf</pre>	41	
		ASYNCHRONOUS :: sendcounts(*), sdispls(*),	43	
0		(*), rdispls(*)	44	
TYPE(		INTENT(IN) :: sendtype, recvtype	45	
		), ASYNCHRONOUS :: recvbuf	46	
	MPI_Comm), INTEN		47	
TYPE(MPI_Info), INTENT(IN) :: info48				

Unofficial Draft for Comment Only

```
1
         TYPE(MPI_Request), INTENT(OUT) :: request
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
4
                   recvcounts, rdispls, recvtype, comm, info, request, ierror)
5
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
6
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
7
                    sendcounts(*), recvcounts(*)
8
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
9
                    rdispls(*)
10
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
11
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
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
     Fortran binding
18
     MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
19
                   RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
20
         <type> SENDBUF(*), RECVBUF(*)
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
21
22
                    RECVTYPE, COMM, INFO, REQUEST, IERROR
23
         Creates a persistent collective communication request for the alltoally operation.
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, info, request)				
IN	sendbuf	starting address of send buffer (choice)	3	
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)	45 6 7	
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)	8 9 10 11	
IN	sendtypes	Array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)	12 13 14 15	
OUT	recvbuf	address of receive buffer (choice)	16	
IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers)	17 18 19	
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)	20 21 22 23 24	
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)	24 25 26 27	
IN	comm	communicator (handle)	28	
IN	info	info argument (handle)	29	
OUT	request	communication request (handle)	30 31 32	
C binding int MPI_Alltoallw_init(const void *sendbuf, const int sendcounts[], const int sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const int recvcounts[], const int rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info, MPI_Request *request)				
<pre>int MPI_Alltoallw_init_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>				

MPI_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdisp	ls,
recvtypes, comm, info, request)	

**Unofficial Draft for Comment Only** 

void \*recvbuf, const MPI\_Count recvcounts[],

MPI\_Alltoallw\_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,

TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf

Fortran 2008 binding

const MPI\_Aint sdispls[], const MPI\_Datatype sendtypes[],

const MPI\_Aint rdispls[], const MPI\_Datatype recvtypes[],

recvcounts, rdispls, recvtypes, comm, info, request, ierror)

MPI\_Comm comm, MPI\_Info info, MPI\_Request \*request)

41

42

43

44

45

46

47

```
1
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
2
                    recvcounts(*), rdispls(*)
3
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
4
                    recvtypes(*)
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         TYPE(MPI_Info), INTENT(IN) :: info
8
         TYPE(MPI_Request), INTENT(OUT) :: request
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
11
                   recvcounts, rdispls, recvtypes, comm, info, request, ierror)
12
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
13
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
14
                    sendcounts(*), recvcounts(*)
15
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
16
                    rdispls(*)
17
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
18
                    recvtypes(*)
19
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
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
25
     Fortran binding
26
     MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
27
                   RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)
28
         <type> SENDBUF(*), RECVBUF(*)
29
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
30
                    RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR
31
         Creates a persistent collective communication request for the alltoally operation.
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

## 6.13.7 Persistent Reduce

0.13.7			2
			$\frac{3}{4}$
	•	unt, datatype, op, root, comm, info, request)	5
IN	sendbuf	address of send buffer (choice)	6
OUT	recvbuf	address of receive buffer (choice, significant only at root)	7 8
IN	count	number of elements in send buffer (non-negative integer)	9 10
IN	datatype	data type of elements of send buffer (handle)	11 12
IN	ор	reduce operation (handle)	13
IN	root	rank of root process (integer)	14
IN	comm	communicator (handle)	15 16
IN	info	info argument (handle)	10
OUT	request	communication request (handle)	18
001	i oquoot		19
C binding	5		20
int MPI_F	Reduce_init(const void *s	endbuf, void *recvbuf, int count,	21 22
	· · · · · ·	e, MPI_Op op, int root, MPI_Comm comm,	23
MPI_Info info, MPI_Request *request)			24
<pre>int MPI_Reduce_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>			25
		e, MPI_Op op, int root, MPI_Comm comm,	26
MPI_Info info, MPI_Request *request)			27 28
Fortran 2008 binding			28 29
MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,			30
ייינעייי (	request, ierror)	T(IN) ACVNCUDONOUS to see that	31
	(*), DIMENSION(), INIEN	T(IN), ASYNCHRONOUS :: sendbuf	32
	ER, INTENT(IN) :: count,		33
	(MPI_Datatype), INTENT(IN		34 35
TYPE (	MPI_Op), INTENT(IN) :: o	p	36
	(MPI_Comm), INTENT(IN) ::		37
	(MPI_Info), INTENT(IN) ::		38
	MPI_Request), INTENT(OUT	-	39
LNIEG	ER, OPTIONAL, INTENT(OUT	) :: lerror	40
MPI_Reduc		count, datatype, op, root, comm, info,	41
יירעייי	request, ierror)		42 43
	(*), DIMENSION(), INTEN (*), DIMENSION(), ASYNC	T(IN), ASYNCHRONOUS :: sendbuf HRONOUS :: recubuf	43 44
	ER(KIND=MPI_COUNT_KIND),		45
	(MPI_Datatype), INTENT(IN		46
	(MPI_Op), INTENT(IN) :: o	• -	47
INTEG	ER, INTENT(IN) :: root		48

Unofficial Draft for Comment Only

1 2 3 4 5	TYPE( TYPE(	MPI_Comm), INTENT(IN) :: MPI_Info), INTENT(IN) :: MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)	info ) :: request
6 7 8 9 10	<type< td=""><td>E_INIT(SENDBUF, RECVBUF, REQUEST, IERROR) &gt; SENDBUF(*), RECVBUF(*)</td><td>COUNT, DATATYPE, OP, ROOT, COMM, INFO, ROOT, COMM, INFO, REQUEST, IERROR</td></type<>	E_INIT(SENDBUF, RECVBUF, REQUEST, IERROR) > SENDBUF(*), RECVBUF(*)	COUNT, DATATYPE, OP, ROOT, COMM, INFO, ROOT, COMM, INFO, REQUEST, IERROR
11 12	Create	es a persistent collective comm	nunication request for the reduce operation.
13 14 15	6.13.8 Pe	ersistent All-Reduce	
16 17	MPI_ALLR	EDUCE_INIT(sendbuf, recvbut	f, count, datatype, op, comm, info, request)
18	IN	sendbuf	starting address of send buffer (choice)
19	OUT	recvbuf	starting address of receive buffer (choice)
20 21 22	IN	count	number of elements in send buffer (non-negative integer)
23	IN	datatype	data type of elements of send buffer (handle)
24	IN	ор	operation (handle)
25	IN	comm	communicator (handle)
26 27	IN	info	info argument (handle)
28 29	OUT	request	communication request (handle)
30	Chindin		
31	C binding		*sendbuf, void *recvbuf, int count,
32 33			e, MPI_Op op, MPI_Comm comm,
34 35	int MPT A	llreduce init c(const vo	id *sendbuf, void *recvbuf,
36	1110 111 1_11		_Datatype datatype, MPI_Op op,
37		MPI_Comm comm, MPI_I	nfo info, MPI_Request *request)
38	Fortran 2	2008 binding	
39 40	MPI_Allre	educe_init(sendbuf, recvb	uf, count, datatype, op, comm, info,
41		request, ierror)	
42		*), DIMENSION(), INIEN *), DIMENSION(), ASYNCI	I(IN), ASYNCHRONOUS :: sendbuf HRONOUS :: recybuf
43		ER, INTENT(IN) :: count	
44 45		MPI_Datatype), INTENT(IN	V1
46		<pre>[MPI_Op), INTENT(IN) :: o] [MDI_Comm) INTENT(IN)</pre>	•
47		<pre>MPI_Comm), INTENT(IN) :: MPI_Info), INTENT(IN) ::</pre>	
48		,,	

	YPE(MPI_Request), INTENT(OUT) :: request	1		
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2		
МРТ	MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,			
	request, ierror)	4 5		
	YPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	6		
	YPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	7		
	NTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	8		
	YPE(MPI_Datatype), INTENT(IN) :: datatype	9		
	YPE(MPI_Op), INTENT(IN) :: op	10		
	YPE(MPI_Comm), INTENT(IN) :: comm YPE(MPI_Info), INTENT(IN) :: info	11		
	YPE(MPI_Request), INTENT(OUT) :: request	12		
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13		
		14 15		
	an binding	16		
MP1_	LLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR)	17		
	type> SENDBUF(*), RECVBUF(*)	18		
	INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR	19		
		20		
(	Creates a persistent collective communication request for the all reduce operation.	21		
6 1 2	) Development Reduce Scotter with Equal Placks	22		
0.15.	Persistent Reduce-Scatter with Equal Blocks	23 24		
		24 25		
MPI	REDUCE_SCATTER_BLOCK_INIT(sendbuf, recvbuf, recvcount, datatype, op, comm,	26		
	info, request)	27		
IN	sendbuf starting address of send buffer (choice)	28		
		29		
OU	recvbuf starting address of receive buffer (choice)	30		
IN	recvcount element count per block (non-negative integer)	31		
IN	data type of elements of send and receive buffers	32		
	(handle)	$\frac{33}{34}$		
- IN	op operation (handle)	35		
IN	comm communicator (handle)	36		
IN	info info argument (handle)	37		
		38		
OU	request communication request (handle)	39		
<u>.</u>		40		
	nding	41		
THE	<pre>IPI_Reduce_scatter_block_init(const void *sendbuf, void *recvbuf, int recvcount, MPI_Datatype datatype, MPI_Op op,</pre>	42 43		
	MPI_Comm comm, MPI_Info info, MPI_Request *request)	43		
		45		
int 1	<pre>IPI_Reduce_scatter_block_init_c(const void *sendbuf, void *recvbuf,</pre>	46		
	MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,	47		
	MPI_Comm comm, MPI_Info info, MPI_Request *request)	48		

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
3
                   comm, info, request, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
         INTEGER, INTENT(IN) :: recvcount
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         TYPE(MPI_Op), INTENT(IN) :: op
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Info), INTENT(IN) :: info
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
14
                   comm, info, request, ierror)
15
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
16
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
17
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
18
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
         TYPE(MPI_Op), INTENT(IN) :: op
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
25
     Fortran binding
26
     MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
27
                   COMM, INFO, REQUEST, IERROR)
28
         <type> SENDBUF(*), RECVBUF(*)
29
         INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
30
         Creates a persistent collective communication request for the reduce-scatter with equal
^{31}
     blocks operation.
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

#### 6.13.10 Persistent Reduce-Scatter

## MPI\_REDUCE\_SCATTER\_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, info,

	request)		Э
	. ,		6
IN	sendbuf	starting address of send buffer (choice)	7
OUT	recvbuf	starting address of receive buffer (choice)	8
IN	recvcounts	non-negative integer array specifying the number of	9
		elements in result distributed to each process. This	10
		array must be identical on all calling processes.	11
			12
IN	datatype	data type of elements of input buffer (handle)	13
IN	ор	operation (handle)	14
IN	comm	communicator (handle)	15
	in fa		16
IN	info	info argument (handle)	17
OUT	request	communication request (handle)	18
			19

#### C binding

int MPI\_Reduce\_scatter\_init(const void \*sendbuf, void \*recvbuf, const int recvcounts[], MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm, MPI\_Info info, MPI\_Request \*request)

int MPI\_Reduce\_scatter\_init\_c(const void \*sendbuf, void \*recvbuf, const MPI\_Count recvcounts[], MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm, MPI\_Info info, MPI\_Request \*request)

#### Fortran 2008 binding

29 MPI\_Reduce\_scatter\_init(sendbuf, recvbuf, recvcounts, datatype, op, comm, 30 info, request, ierror) 31TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 32 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 33 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(\*) 34 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 35 TYPE(MPI\_Op), INTENT(IN) :: op 36 TYPE(MPI\_Comm), INTENT(IN) :: comm 37 TYPE(MPI\_Info), INTENT(IN) :: info 38 TYPE(MPI\_Request), INTENT(OUT) :: request 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 MPI\_Reduce\_scatter\_init(sendbuf, recvbuf, recvcounts, datatype, op, comm, 41 info, request, ierror) 42TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 43 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 44 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(\*) 45 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 46TYPE(MPI\_Op), INTENT(IN) :: op 47TYPE(MPI\_Comm), INTENT(IN) :: comm 48

1 2

5

2021

22

23

24

25

26

1 2		MPI_Info), INTENT(IN) :: MPI_Request), INTENT(OUT)	
3	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror
4	Fortran b	binding	
5 6		0	RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
7		INFO, REQUEST, IERRO	R)
8		<pre>&gt; SENDBUF(*), RECVBUF(*) </pre>	PE, OP, COMM, INFO, REQUEST, IERROR
9			
10 11	Create	es a persistent collective comm	nunication request for the reduce-scatter operation.
12	6 13 11	Persistent Inclusive Scan	
13	0.10.11		
14			
15 16	MPI_SCAN	I_INIT(sendbuf, recvbuf, count	, datatype, op, comm, info, request)
17	IN	sendbuf	starting address of send buffer (choice)
18	OUT	recvbuf	starting address of receive buffer (choice)
19 20	IN	count	number of elements in input buffer (non-negative
20			integer)
22	IN	datatype	data type of elements of input buffer (handle)
23	IN	ор	operation (handle)
24 25	IN	comm	communicator (handle)
26	IN	info	info argument (handle)
27 28	OUT	request	communication request (handle)
29	C binding		
30		5	dbuf, void *recvbuf, int count,
31 32			e, MPI_Op op, MPI_Comm comm,
33		MPI_Info info, MPI_R	equest *request)
34	int MPI_S		endbuf, void *recvbuf, MPI_Count count,
35		MPI_Datatype datatyp MPI_Info info, MPI_R	e, MPI_Op op, MPI_Comm comm,
$\frac{36}{37}$			equest *request)
38		2008 binding	ount detetune on comm info request
39	MF1_SCall_	ierror)	ount, datatype, op, comm, info, request,
40 41	TYPE(		T(IN), ASYNCHRONOUS :: sendbuf
42		(*), DIMENSION(), ASYNC	HRONOUS :: recvbuf
43		ER, INTENT(IN) :: count	
44		<pre>[MPI_Datatype), INTENT(IN] [MPI_Op), INTENT(IN) :: op</pre>	V-1
45		(MPI_Comm), INTENT(IN) ::	•
46 47		<pre>MPI_Info), INTENT(IN) ::</pre>	
48	TYPE(	MPI_Request), INTENT(OUT)	) :: request

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         2
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
               ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                         6
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                         10
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                         11
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                         12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         13
                                                                                         14
Fortran binding
MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
                                                                                         15
                                                                                         16
               IERROR)
                                                                                         17
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         18
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
                                                                                         19
    Creates a persistent collective communication request for the inclusive scan operation.
                                                                                         20
                                                                                         21
6.13.12 Persistent Exclusive Scan
                                                                                         22
                                                                                         23
                                                                                         ^{24}
MPI_EXSCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)
                                                                                         25
                                                                                         26
  IN
           sendbuf
                                       starting address of send buffer (choice)
                                                                                         27
  OUT
           recvbuf
                                       starting address of receive buffer (choice)
                                                                                         28
  IN
                                       number of elements in input buffer (non-negative
                                                                                         29
           count
                                       integer)
                                                                                         30
                                                                                         31
  IN
                                       data type of elements of input buffer (handle)
           datatype
                                                                                         32
  IN
           ор
                                       operation (handle)
                                                                                         33
                                                                                         34
  IN
                                       intra-communicator (handle)
           comm
                                                                                         35
  IN
           info
                                       info argument (handle)
                                                                                         36
  OUT
           request
                                       communication request (handle)
                                                                                         37
                                                                                         38
C binding
                                                                                         39
int MPI_Exscan_init(const void *sendbuf, void *recvbuf, int count,
                                                                                         40
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                                                                                         41
               MPI_Info info, MPI_Request *request)
                                                                                         42
                                                                                         43
int MPI_Exscan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
                                                                                         44
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                                                                                         45
               MPI_Info info, MPI_Request *request)
                                                                                         46
Fortran 2008 binding
                                                                                         47
                                                                                         48
```

```
1
     MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
\mathbf{2}
                    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_Info), INTENT(IN) :: info
10
         TYPE(MPI_Request), INTENT(OUT) :: request
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
13
                    ierror)
14
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
15
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Op), INTENT(IN) :: op
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Info), INTENT(IN) :: info
21
         TYPE(MPI_Request), INTENT(OUT) :: request
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     Fortran binding
25
     MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
26
                   IERROR)
27
         <type> SENDBUF(*), RECVBUF(*)
28
         INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
29
         Creates a persistent collective communication request for the exclusive scan operation.
30
31
32
     6.14
            Correctness
33
34
     A correct, portable program must invoke collective communications so that deadlock will not
     occur, whether collective communications are synchronizing or not. The following examples
35
     illustrate dangerous use of collective routines on intra-communicators.
36
37
     Example 6.25 The following is erroneous.
38
39
     switch(rank) {
40
         case 0:
41
              MPI_Bcast(buf1, count, type, 0, comm);
42
              MPI_Bcast(buf2, count, type, 1, comm);
43
              break;
44
         case 1:
45
              MPI_Bcast(buf2, count, type, 1, comm);
46
              MPI_Bcast(buf1, count, type, 0, comm);
47
```

break;

}

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 6.26 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(buf2, 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 6.27** The following is erroneous.

```
switch(rank) {
   case 0:
      MPI_Bcast(buf1, count, type, 0, comm);
      MPI_Send(buf2, count, type, 1, tag, comm);
      break;
   case 1:
      MPI_Recv(buf2, count, type, 0, tag, comm, status);
      MPI_Bcast(buf1, count, type, 0, comm);
      break;
}
```

}

Process zero executes a broadcast, followed by a blocking send operation. Process one 43 first executes a blocking receive that matches the send, followed by broadcast call that 44 matches the broadcast of process zero. This program may deadlock. The broadcast call on 45 process zero may block until process one executes the matching broadcast call, so that the 46 send is not executed. Process one will definitely block on the receive and so, in this case, 47 never executes the broadcast. 48

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

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

The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

```
Example 6.28 An unsafe, non-deterministic program.
```

```
switch(rank) {
   case 0:
       MPI_Bcast(buf1, count, type, 0, comm);
       MPI_Send(buf2, count, type, 1, tag, comm);
       break;
   case 1:
       MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
       MPI_Bcast(buf1, count, type, 0, comm);
                                                     tag, comm, status);
       MPI_Recv(buf2, count, type, MPI_ANY_SOURCE,
       break;
   case 2:
       MPI_Send(buf2, count, type, 1, tag, comm);
       MPI_Bcast(buf1, count, type, 0, comm);
       break:
}
```

All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

25Two possible executions of this program, with different matchings of sends and receives, 26are illustrated in Figure 6.12. Note that the second execution has the peculiar effect that 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 30 first execution occurs (only when broadcast is synchronizing) is erroneous.  $^{31}$ 

Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication call at a process. In these situations, it is the user's responsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.

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

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

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

45It is the implementor's responsibility to ensure that point-to-point messages are not 46 confused with collective messages. One way to accomplish this is, whenever a commu-47 nicator is created, to also create a "hidden communicator" for collective communica-48 tion. One could achieve a similar effect more cheaply, for example, by using a hidden

#### **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

2122

23

24

27

28

29

32

33

34

35 36

37

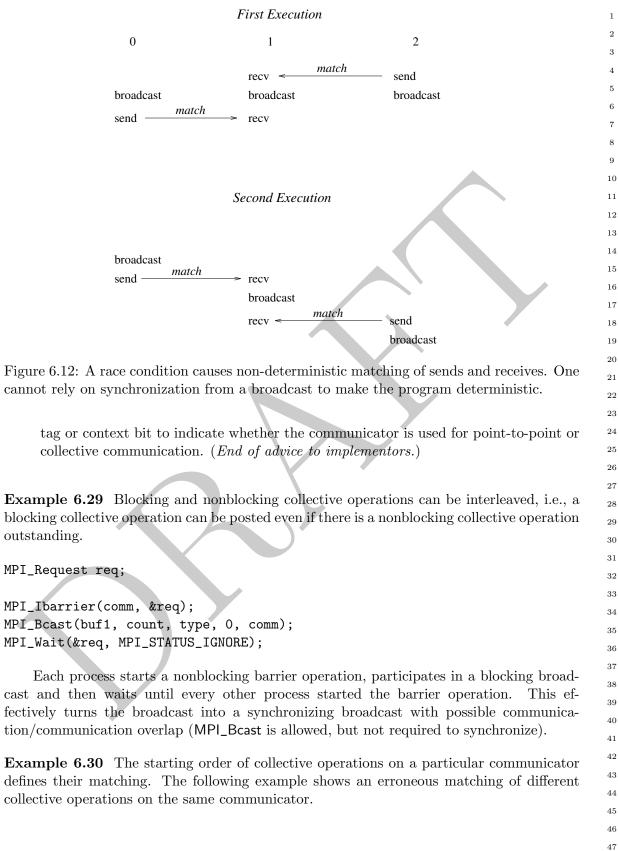
38

39 40

41

42

43



```
1
     MPI_Request req;
\mathbf{2}
     switch(rank) {
3
          case 0:
4
               /* erroneous matching */
5
              MPI_Ibarrier(comm, &req);
6
              MPI_Bcast(buf1, count, type, 0, comm);
7
              MPI_Wait(&req, MPI_STATUS_IGNORE);
8
              break;
9
          case 1:
10
               /* erroneous matching */
11
              MPI_Bcast(buf1, count, type, 0, comm);
12
              MPI_Ibarrier(comm, &req);
13
              MPI_Wait(&req, MPI_STATUS_IGNORE);
14
              break;
15
     }
16
17
          This ordering would match MPI_Ibarrier on rank 0 with MPI_Bcast on rank 1 which is
     erroneous and the program behavior is undefined. However, if such an order is required, the
18
     user must create different duplicate communicators and perform the operations on them.
19
     If started with two processes, the following program would be correct:
20
21
     MPI_Request req;
22
     MPI_Comm dupcomm;
23
     MPI_Comm_dup(comm, &dupcomm);
^{24}
     switch(rank) {
25
          case 0:
26
              MPI_Ibarrier(comm, &req);
27
              MPI_Bcast(buf1, count, type, 0, dupcomm);
28
              MPI_Wait(&req, MPI_STATUS_IGNORE);
29
              break;
30
          case 1:
31
              MPI_Bcast(buf1, count, type, 0, dupcomm);
32
              MPI_Ibarrier(comm, &req);
33
              MPI_Wait(&req, MPI_STATUS_IGNORE);
34
              break;
35
     }
36
37
           Advice to users. The use of different communicators offers some flexibility regarding
38
           the matching of nonblocking collective operations. In this sense, communicators could
39
           be used as an equivalent to tags. However, communicator construction might induce
40
           overheads so that this should be used carefully. (End of advice to users.)
41
42
43
     Example 6.31 Nonblocking collective operations can rely on the same progression rules
44
     as nonblocking point-to-point messages. Thus, if started with two processes, the following
45
     program is a valid MPI program and is guaranteed to terminate:
46
47
```

48

```
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 6.32** 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;
}
```

**Example 6.33** Collective and point-to-point requests can be mixed in functions that enable multiple completions. If started with two processes, the following program is valid.

```
MPI_Request reqs[2];
switch(rank) {
    case 0:
      MPI_Ibarrier(comm, &reqs[0]);
      MPI_Send(buf, count, dtype, 1, tag, comm);
      MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
      break;
    case 1:
```

Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 3

4

5 6

9

10

11

1213

1415

16

17

18 19

20

2122

23 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35 36 37

38

39 40

41 42

> 43 44

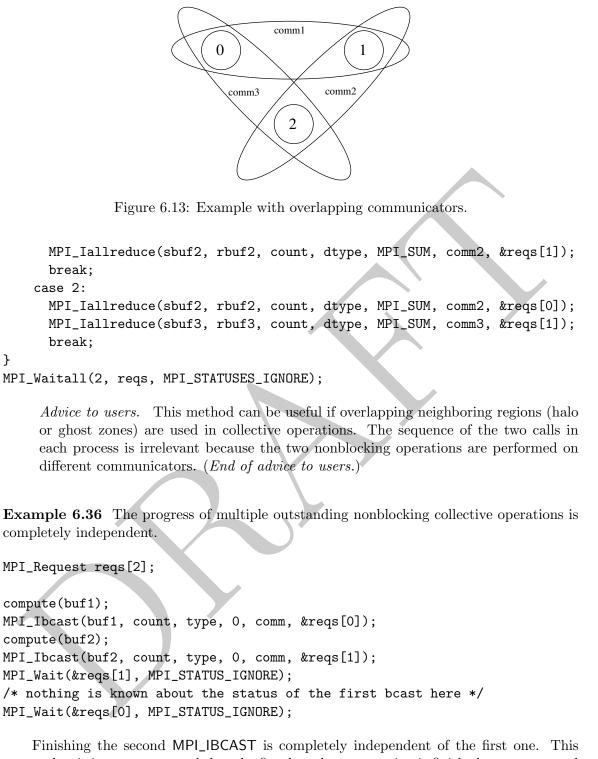
> 45

4647

```
MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
2
            MPI_Ibarrier(comm, &reqs[1]);
3
            MPI_Waitall(2, regs, MPI_STATUSES_IGNORE);
4
            break;
5
     }
6
          The MPI_Waitall call returns only after the barrier and the receive completed.
7
8
     Example 6.34 Multiple nonblocking collective operations can be outstanding on a single
9
     communicator and match in order.
10
11
     MPI_Request reqs[3];
12
13
     compute(buf1);
14
     MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
15
     compute(buf2);
16
     MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
17
     compute(buf3);
18
     MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
19
     MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
20
21
           Advice to users. Pipelining and double-buffering techniques can efficiently be used
22
           to overlap computation and communication. However, having too many outstanding
23
           requests might have a negative impact on performance. (End of advice to users.)
24
25
           Advice to implementors.
                                      The use of pipelining may generate many outstanding
26
           requests. A high-quality hardware-supported implementation with limited resources
27
           should be able to fall back to a software implementation if its resources are exhausted.
28
           In this way, the implementation could limit the number of outstanding requests only
29
           by the available memory. (End of advice to implementors.)
30
^{31}
32
     Example 6.35 Nonblocking collective operations can also be used to enable simultane-
33
     ous collective operations on multiple overlapping communicators (see Figure 6.13). The
34
     following example is started with three processes and three communicators. The first com-
35
     municator comm1 includes ranks 0 and 1, comm2 includes ranks 1 and 2, and comm3 spans
36
     ranks 0 and 2. It is not possible to perform a blocking collective operation on all commu-
37
     nicators because there exists no deadlock-free order to invoke them. However, nonblocking
38
     collective operations can easily be used to achieve this task.
39
     MPI_Request reqs[2];
40
41
42
     switch(rank) {
          case 0:
43
            MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
44
            MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
45
            break;
46
47
          case 1:
48
            MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
```

**Unofficial Draft for Comment Only** 

}



means that it is not guaranteed that the first broadcast operation is finished or even started after the second one is completed via regs[1].

 $\mathbf{2}$ 

 $^{24}$ 



## Chapter 7

# Groups, Contexts, Communicators, and Caching

### 7.1 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 [62] and [4] for further information on writing libraries in MPI, using the features described in this chapter.

#### 7.1.1 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.  $^{24}$ 

## 7.1.2 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.

**Communicators** (see [22, 60, 64]) encapsulate all of these ideas in order to provide the appropriate scope for all communication operations in MPI. Communicators are divided into two kinds: intra-communicators for operations within a single group of processes and inter-communicators for operations between two groups of processes.

<sup>19</sup> Caching. Communicators (see below) provide a "caching" mechanism that allows one to <sup>20</sup> associate new attributes with communicators, on par with MPI built-in features. This can <sup>21</sup> be used by advanced users to adorn communicators further, and by MPI to implement <sup>22</sup> some communicator functions. For example, the virtual-topology functions described in <sup>23</sup> Chapter 8 are likely to be supported this way.

24

1

2

3

4 5

6

7 8

9 10

11

12 13

14

15

16

17 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 8 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. When using the World Model (Section 11.2), 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.

## 7.2 Basic Concepts

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

7.2.1 Groups

1

 $\mathbf{2}$ 

3 4

5 6

> 7 8

9

20

21

22

23

24 25

26

27

28

34

35

<sup>10</sup> A **group** is an ordered set of process identifiers (henceforth processes); processes are imple-<sup>11</sup> mentation-dependent objects. Each process in a group is associated with an integer **rank**. <sup>12</sup> Ranks are contiguous and start from zero. Groups are represented by opaque **group ob-**<sup>13</sup> **jects**, and hence cannot be directly transferred from one process to another. A group is <sup>14</sup> used within a communicator to describe the participants in a communication "universe" <sup>15</sup> and to rank such participants (thus giving them unique names within that "universe" of <sup>16</sup> 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.*)
  - 7.2.2 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.*)

#### 7.2.3 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 8), communicators may also "cache" additional information (see Section 7.7). 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.

#### 7.2.4 Predefined Intra-Communicators

When using the World Model for MPI initialization, 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. When using the Sessions Model (Section 11.3) for initialization of MPI resources, MPI\_COMM\_WORLD and MPI\_COMM\_SELF are not valid for use as a communicator. See the discussion concerning use of MPI named constants in 2.5.4 for valid uses of MPI\_COMM\_WORLD and MPI\_COMM\_SELF prior to initialization of MPI.

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 computation are available after MPI is initialized. For this case, MPI\_COMM\_WORLD is a 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 dynamically join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, MPI\_COMM\_WORLD is a communicator incorporating all processes with which the joining process can immediately

## 309

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

48

#### Unofficial Draft for Comment Only

```
1
     communicate. Therefore, MPI_COMM_WORLD may simultaneously represent disjoint groups
\mathbf{2}
     in different processes.
3
          All MPI implementations are required to provide the MPI_COMM_WORLD communi-
4
     cator. It cannot be deallocated during the life of a process. The group corresponding to
\mathbf{5}
     this communicator does not appear as a pre-defined constant, but it may be accessed using
6
     MPI_COMM_GROUP (see below). MPI does not specify the correspondence between the
7
     process rank in MPI_COMM_WORLD and its (machine-dependent) absolute address. Neither
8
     does MPI specify the function of the host process, if any. Other implementation-dependent,
9
     predefined communicators may also be provided.
10
11
     7.3
            Group Management
12
13
     This section describes the manipulation of process groups in MPI. These operations are
14
     local and their execution does not require interprocess communication.
15
16
     7.3.1 Group Accessors
17
18
19
     MPI_GROUP_SIZE(group, size)
20
21
       IN
                                              group (handle)
                 group
22
       OUT
                                              number of processes in the group (integer)
                 size
23
^{24}
     C binding
25
     int MPI_Group_size(MPI_Group group, int *size)
26
27
     Fortran 2008 binding
28
     MPI_Group_size(group, size, ierror)
29
          TYPE(MPI_Group), INTENT(IN) :: group
30
          INTEGER, INTENT(OUT) :: size
^{31}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     Fortran binding
33
     MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
34
          INTEGER GROUP, SIZE, IERROR
35
36
37
     MPI_GROUP_RANK(group, rank)
38
39
       IN
                                              group (handle)
                 group
40
       OUT
                                              rank of the calling process in group, or
                 rank
41
                                              MPI_UNDEFINED if the process is not a member
42
                                              (integer)
43
44
     C binding
45
     int MPI_Group_rank(MPI_Group group, int *rank)
46
47
     Fortran 2008 binding
48
```

TYPE( INTEC INTEC	<pre>MPI_Group_rank(group, rank, ierror)     TYPE(MPI_Group), INTENT(IN) :: group     INTEGER, INTENT(OUT) :: rank     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>		
MPI_GROUP	Fortran binding MPI_GROUP_RANK(GROUP, RANK, IERROR) INTEGER GROUP, RANK, IERROR		
MPI_GRO	JP_TRANSLATE_RANKS(grou	up1, n, ranks1, group2, ranks2)	10 11
IN	group1	group1 (handle)	12
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)	13
IN	ranks1	array of zero or more valid ranks in group1	14
IN	group2	group2 (handle)	15 16
	•	° · ( )	17
OUT	ranks2	array of corresponding ranks in group2, MPI_UNDEFINED when no correspondence exists.	18
		Win 1_ONDER INCE when no correspondence exists.	19
C binding	2		20
		Group group1, int n, const int ranks1[],	21 22
	MPI_Group group2, in		22
Fortran 2	2008 binding		24
	MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)		
-	MPI_Group), INTENT(IN) :		26
INTEG	ER, INTENT(IN) :: n, rand	xs1(n)	27
	ER, INTENT(OUT) :: ranks		28 29
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	29 30
Fortran b	oinding		31
		N, RANKS1, GROUP2, RANKS2, IERROR)	32
INTEC	ER GROUP1, N, RANKS1(*),	GROUP2, RANKS2(*), IERROR	33
This f	unction is important for detern	nining the relative numbering of the same processes	34
		ne knows the ranks of certain processes in the group	35
		to know their ranks in a subset of that group.	36 37
		nput to MPI_GROUP_TRANSLATE_RANKS, which	38
returns MI	PI_PROC_NULL as the translate	ed rank.	39
			40
MPI_GRO	JP_COMPARE(group1, group2	2, result)	41
IN	group1	first group (handle)	42
IN	group2	second group (handle)	43
OUT	result	result (integer)	44 45
		( ··· O· /	46
C binding	r S		47
	-	roup1, MPI_Group group2, int *result)	48

1	Fortran	2008 binding	
2			p1, group2, result, ierror)
3			NTENT(IN) :: group1, group2
4		-	
		GER, INTENT(OU	
5	INTE	GER, UPTIONAL,	INTENT(OUT) :: ierror
6	Fortran	binding	
7			P1, GROUP2, RESULT, IERROR)
8			
9		GER GRUUPI, GR	OUP2, RESULT, IERROR
10	MPI_IDEN	T results if the gr	oup members and group order is exactly the same in both groups.
11		0	if group1 and group2 are the same handle. MPI_SIMILAR results if
12			e same but the order is different. MPI_UNEQUAL results otherwise.
13	the group	members are the	same but the order is different. With_ONEQUAE results other wise.
14	700 0	<b>C</b>	
15	7.3.2 G	roup Constructor	S
	MPI prov	ides two approact	nes to constructing groups. In the first approach, MPI procedures
16	_		ad superset existing groups. These constructors construct new
17	-		
18			ps. In the second approach, a group is created using a session
19		_	ess set. This second approach is available when using the Sessions
20			baches, these are local operations, and distinct groups may be
21	defined or	1 different process	ses; a process may also define a group that does not include itself.
22	Consisten	t definitions are	required when groups are used as arguments in communicator-
23	building f	functions. When	using the World Model for initializing MPI, the base group, upon
24	which all	other groups are	e defined, is the group associated with the initial communicator
25			ssible through the function MPI_COMM_GROUP).
26		_ (	
27	Rat	ionale. In what	at follows, there is no group duplication function analogous to
			lefined later in this chapter. There is no need for a group dupli-
28			e created, can have several references to it by making copies of
29			owing constructors address the need for subsets and supersets of
30		ting groups. ( $En$	
31	exis	ting groups. (En	a of Tationale.)
32	1 da	vice to implement	tors. Each group constructor behaves as if it returned a new
33		-	
34			n this new group is a copy of an existing group, then one can
35		=	new objects, using a reference-count mechanism. ( $End \ of \ advice$
36	to i	mplementors.)	
37			
38			
39			
40	MPI_CON	/IM_GROUP(com	m, group)
41	IN	comm	communicator (handle)
42	OUT	group	group corresponding to <b>comm</b> (handle)
	501	9. ~ 4P	Storb corresponding to comm (nandro)
43	C h		
44	C bindir		
45	int MPI_	Comm_group(MPI	_Comm comm, MPI_Group *group)
46	Fortran	2008 binding	
47		_group(comm, g	roup, jerror)
48	<u>-</u>	-0r (, 6	

TYPE	(MPI_Comm), INTENT(IN) :	: comm	1
TYPE	(MPI_Group), INTENT(OUT)	:: group	2
INTE	GER, OPTIONAL, INTENT(OU	T) :: ierror	3
Fortran binding			
	_GROUP(COMM, GROUP, IERR	OR)	5
	GER COMM, GROUP, IERROR		6 7
MDI	COMM CROUP noturns in a	roup a handle to the group of comm	8
IVIPI_	COMM_GROOP returns in g	roup a handle to the group of comm.	9
			10
MPI_GRC	OUP_UNION(group1, group2,	newgroup)	11
IN	group1	first group (handle)	12
IN	group2	second group (handle)	13
			14
OUT	newgroup	union group (handle)	15
Chindin			16 17
C bindir	g Group_union(MPI_Group gr	oun1 MPI Group group?	18
IIIC MII_	MPI_Group *newgroup		19
			20
	2008 binding		21
	p_union(group1, group2,		22
	:(MPI_Group), INTENT(IN) :(MPI_Group), INTENT(OUT)		23
	GER, OPTIONAL, INTENT(OU)		24
			25
Fortran			26
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)			27 28
TNLE	GER GROUP1, GROUP2, NEWG	ROUP, IERROR	28 29
			30
			31
	OUP_INTERSECTION(group1)		32
IN	group1	first group (handle)	33
IN	group2	second group (handle)	34
Ουτ	newgroup	intersection group (handle)	35
	5.		36
C bindir	19		37
	0	roup group1, MPI_Group group2,	38 39
	MPI_Group *newgroup		39 40
Fortron	2008 binding		41
	6	roup2, newgroup, ierror)	42
	(MPI_Group), INTENT(IN)		43
	(MPI_Group), INTENT(OUT)		44
	GER, OPTIONAL, INTENT(OU		45
			46
Fortran	0	ROUP2, NEWGROUP, IERROR)	47
III 1_01100		NGOLZ, NEWGROOF, TERROT	48

```
1
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
\mathbf{2}
3
4
      MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
5
       IN
                 group1
                                               first group (handle)
6
       IN
                  group2
                                               second group (handle)
7
8
       OUT
                 newgroup
                                               difference group (handle)
9
10
     C binding
11
      int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
12
                     MPI_Group *newgroup)
13
     Fortran 2008 binding
14
     MPI_Group_difference(group1, group2, newgroup, ierror)
15
          TYPE(MPI_Group), INTENT(IN) :: group1, group2
16
          TYPE(MPI_Group), INTENT(OUT) :: newgroup
17
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
      Fortran binding
20
     MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
21
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
22
23
      The set-like operations are defined as follows:
^{24}
      union All elements of the first group (group1), followed by all elements of second group
25
           (group2) not in the first group.
26
      intersect all elements of the first group that are also in the second group, ordered as in
27
           the first group.
28
29
      difference all elements of the first group that are not in the second group, ordered as in
30
           the first group.
^{31}
32
      Note that for these operations the order of processes in the output group is determined
      primarily by order in the first group (if possible) and then, if necessary, by order in the
33
34
      second group. Neither union nor intersection are commutative, but both are associative.
          The new group can be empty, that is, equal to MPI_GROUP_EMPTY.
35
36
37
      MPI_GROUP_INCL(group, n, ranks, newgroup)
38
39
       IN
                                               group (handle)
                  group
40
       IN
                                               number of elements in array ranks (and size of
                  n
41
                                               newgroup) (integer)
42
       IN
                                               ranks of processes in group to appear in newgroup
                  ranks
43
                                               (array of integers)
44
       OUT
                                               new group derived from above, in the order defined
                  newgroup
45
                                               by ranks (handle)
46
47
48
      C binding
```

#### Fortran 2008 binding

```
MPI_Group_incl(group, n, ranks, newgroup, ierror)
   TYPE(MPI_Group), INTENT(IN) :: group
   INTEGER, INTENT(IN) :: n, ranks(n)
   TYPE(MPI_Group), INTENT(OUT) :: newgroup
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### Fortran binding

```
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

The function MPI\_GROUP\_INCL creates a group newgroup that consists of the n processes in group with ranks ranks[0],..., ranks[n-1]; the process with rank i in newgroup is the process with rank ranks[i] in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct, or else the program is erroneous. If n = 0, then newgroup is MPI\_GROUP\_EMPTY. This function can, for instance, be used to reorder the elements of a group. See also MPI\_GROUP\_COMPARE.

#### MPI\_GROUP\_EXCL(group, n, ranks, newgroup)

			22
IN	group	group (handle)	23
IN	n	number of elements in array ranks (integer)	24
IN	ranks	array of integer ranks of processes in group not to	25
		appear in newgroup	26
OUT	DOWGROUD		27
001	newgroup	new group derived from above, preserving the order defined by group (handle)	28
		defined by group (nandle)	29
			30
C binding			31
int MPI_G		p, int n, const int ranks[],	32
	MPI_Group *newgroup)		33
Fortran 2008 binding			34
	_excl(group, n, ranks, n	ewgroup jerror)	35
	MPI_Group), INTENT(IN) :		36
	ER, INTENT(IN) :: n, ran	<b>.</b>	37
	MPI_Group), INTENT(OUT)		38
	ER, OPTIONAL, INTENT(OUT)		39
INIEG	ER, UPIIONAL, INIENI(UUI		40
Fortran b	binding		41
MPI_GROUP	_EXCL(GROUP, N, RANKS, N	EWGROUP, IERROR)	42
INTEG	ER GROUP, N, RANKS(*), N	EWGROUP, IERROR	43
The fu	unation MDL CROUP EVCL	contras a group of processes pougroup that is obtained	44
		reates a group of processes <b>newgroup</b> that is obtained	45

The function MPI\_GROUP\_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0],..., ranks[n-1]. The ordering of processes in newgroup is identical to the ordering in group. Each of the n elements of ranks

#### 316 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
      must be a valid rank in group and all elements must be distinct; otherwise, the program is
\mathbf{2}
      erroneous. If n = 0, then newgroup is identical to group.
3
4
      MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
5
6
        IN
                                                  group (handle)
                   group
7
        IN
                                                  number of triplets in array ranges (integer)
                   n
8
        IN
                   ranges
                                                  a one-dimensional array of integer triplets, of the
9
                                                  form (first rank, last rank, stride) indicating ranks in
10
                                                  group of processes to be included in newgroup
11
                                                  new group derived from above, in the order defined
12
        OUT
                   newgroup
13
                                                  by ranges (handle)
14
15
      C binding
16
      int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
17
                       MPI_Group *newgroup)
18
      Fortran 2008 binding
19
      MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
20
           TYPE(MPI_Group), INTENT(IN) :: group
21
           INTEGER, INTENT(IN) :: n, ranges(3, n)
22
           TYPE(MPI_Group), INTENT(OUT) :: newgroup
23
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
      Fortran binding
26
      MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
27
           INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR
28
      If ranges consists of the triplets

(first_1, last_1, stride_1), \dots, (first_n, last_n, stride_n)
29
30
^{31}
32
      then newgroup consists of the sequence of processes in group with ranks
33
           first_1, first_1 + stride_1, \dots, first_1 + \left| \frac{last_1 - first_1}{stride_1} \right| stride_1, \dots,
34
35
           first_n, first_n + stride_n, \dots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.
36
37
38
39
           Each computed rank must be a valid rank in group and all computed ranks must be
40
      distinct, or else the program is erroneous. Note that we may have first_i > last_i, and stride_i
41
      may be negative, but cannot be zero.
42
           The functionality of this routine is specified to be equivalent to expanding the array
43
      of ranges to an array of the included ranks and passing the resulting array of ranks and
44
      other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call
45
      to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the
46
      argument ranges.
47
```

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)						
	IN	group	group (handle)	2		
	IN	0		3		
		n	number of triplets in array ranges (integer)	4		
	IN	ranges	a one-dimensional array of integer triplets, of the	5		
			form (first rank, last rank, stride) indicating ranks in	6		
			group of processes to be excluded from the output	8		
			group newgroup (array of integers)	0 9		
	OUT	newgroup	new group derived from above, preserving the order	10		
			in group (handle)	11		

#### MPL GROUP RANGE EXCL(group n ranges newgroup)

#### C binding

<pre>int MPI_Group_range_excl(MPI_Group group,</pre>	int n,	int	<pre>ranges[][3],</pre>
MPI_Group *newgroup)			

#### Fortran 2008 binding

<pre>MPI_Group_range_excl(group, n, ranges, newgroup, ierror)</pre>	
TYPE(MPI_Group), INTENT(IN) :: group	
<pre>INTEGER, INTENT(IN) :: n, ranges(3, n)</pre>	
TYPE(MPI_Group), INTENT(OUT) :: newgroup	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

#### Fortran binding

MPI_	_GROUP_R	ANGE_	EXCL(	GROUP,	N,	RANGE	ES, N	JEWGROU	JP,	IERROR)	
	INTEGER	GROU	JP, N,	RANGES	3(3,	*),	NEWO	ROUP,	IER	ROR	

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.

The range operations do not explicitly enumerate ranks, and Advice to users. 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.)

 $^{31}$ 

```
318 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING
```

```
1
     MPI_GROUP_FROM_SESSION_PSET(session, pset_name, newgroup)
2
       IN
                session
                                            session (handle)
3
       IN
                                            name of process set to use to create the new group
                 pset_name
4
                                            (string)
5
6
       OUT
                                            new group derived from supplied session and process
                newgroup
7
                                            set (handle)
8
9
     C binding
10
     int MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,
11
                    MPI_Group *newgroup)
12
     Fortran 2008 binding
13
     MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)
14
         TYPE(MPI_Session), INTENT(IN) :: session
15
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
16
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     Fortran binding
20
     MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR)
21
          INTEGER SESSION, NEWGROUP, IERROR
22
         CHARACTER*(*) PSET_NAME
23
         The function MPI_GROUP_FROM_SESSION_PSET creates a group newgroup using the
^{24}
     provided session handle and process set. The process set name must be one returned from
25
     an invocation of MPI_SESSION_GET_NTH_PSET using the supplied session handle. If the
26
     pset_name does not exist, MPI_GROUP_NULL will be returned in the newgroup argument.
27
     As with other group constructors, MPI_GROUP_FROM_SESSION_PSET is a local function.
28
     See Section 11.3 for more information on sessions and process sets.
29
30
^{31}
     7.3.3 Group Destructors
32
33
34
     MPI_GROUP_FREE(group)
35
       INOUT group
                                            group (handle)
36
37
     C binding
38
     int MPI_Group_free(MPI_Group *group)
39
40
     Fortran 2008 binding
41
     MPI_Group_free(group, ierror)
42
         TYPE(MPI_Group), INTENT(INOUT) :: group
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     Fortran binding
45
     MPI_GROUP_FREE(GROUP, IERROR)
46
         INTEGER GROUP, IERROR
47
48
```

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.*)

## 7.4 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.)

7.4.1 Communicator Accessors	23
	24
The following are all local operations.	25
	26
MPI_COMM_SIZE(comm, size)	27
	28
IN comm communicator (handle)	29
OUT size number of processes in the group of comm (integ	ger) <sup>30</sup>
	31
C binding	32
int MPI_Comm_size(MPI_Comm comm, int *size)	33
	34
Fortran 2008 binding	35
MPI_Comm_size(comm, size, ierror)	36
TYPE(MPI_Comm), INTENT(IN) :: comm	37
INTEGER, INTENT(OUT) :: size	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
Frates a 1' a l'an	40
Fortran binding	41
MPI_COMM_SIZE(COMM, SIZE, IERROR)	42
INTEGER COMM, SIZE, IERROR	43
	44
Rationale. This function is equivalent to accessing the communicator's group	with 45

*Rationale.* This function is equivalent to accessing the communicator's group with <sup>45</sup> MPI\_COMM\_GROUP (see above), computing the size using MPI\_GROUP\_SIZE, and <sup>46</sup> then freeing the temporary group via MPI\_GROUP\_FREE. However, this function is <sup>47</sup> so commonly used that this shortcut was introduced. (*End of rationale.*) <sup>48</sup>

#### Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9 10 11

12 13

14

15

16

17 18

19

20

21

1 2 3 4 5	Advice to users. This function indicates the number of processes involved in a communicator. For MPI_COMM_WORLD, it indicates the total number of processes available unless the number of processes has been changed by using the functions described in Chapter 11; note that the number of processes in MPI_COMM_WORLD does not change during the life of an MPI program.						
6 7 8 9 10 11	This call is often used with the next call to determine the amount of concurrency available for a specific library or program. The following call, MPI_COMM_RANK indicates the rank of the process that calls it in the range from 0,, size-1, where size is the return value of MPI_COMM_SIZE.( <i>End of advice to users.</i> )						
12 13	MPL COM	IM_RANK(cor	nm. rank)				
14	IN	comm	,	communicator (handle)			
15 16	OUT	rank		rank of the calling process in group of comm (integer)			
17 18 19	C binding	-	PI_Comm comm, i	nt *rank)			
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39	MPI_Comm_ TYPE( INTEC INTEC Fortran h MPI_COMM_ INTEC Rati MPI_ and is so Advi nicat	(MPI_Comm), GER, INTENT( GER, OPTIONA Dinding RANK(COMM, GER COMM, RA Onale. This COMM_GRC then freeing th commonly us fice to users. The tor's group. It y programs	rank, ierror) INTENT(IN) :: (OUT) :: rank AL, INTENT(OUT) RANK, IERROR) ANK, IERROR function is equive OUP (see above), he temporary grou sed that this shor This function give t is useful, as note will be written v				
40 41 42 43 44 45 46 47 48	two	preceding call	-	serve as compute nodes. In this framework, the letermining the roles of the various processes of a <i>isers</i> .)			

MPL COMM COMPARE(comm1 comm2 result)

IN	comm1	first communicator (handle)	
IN	comm2	second communicator (handle)	
OUT	result	result (integer)	

## C binding

int MPI\_Comm\_compare(MPI\_Comm comm1, MPI\_Comm comm2, int \*result)

#### Fortran 2008 binding

```
MPI_Comm_compare(comm1, comm2, result, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
    INTEGER, INTENT(OUT) :: result
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### Fortran binding

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.

#### 7.4.2 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, MPI\_COMM\_CREATE\_FROM\_GROUP, and MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS. MPI\_COMM\_CREATE\_GROUP and MPI\_COMM\_CREATE\_FROM\_GROUP are invoked only by the processes in the group of the new communicator being constructed. MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS is invoked by all the processes in the local and remote groups of the new communicator being constructed. See the discussion below for the definition of local and remote groups.

*Rationale.* Note that, when using the World Model, there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. In the World Model, the base communicator for all MPI communicators is predefined outside of MPI, and is MPI\_COMM\_WORLD. The World 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 intra-communicators and inter-communicators; MPI\_COMM\_CREATE\_GROUP, MPI\_COMM\_CREATE\_FROM\_GROUP, and MPI\_INTERCOMM\_MERGE (see Section 7.6.2) can be used to create intra-communicators; 

```
1
     MPI_INTERCOMM_CREATE and MPI_INTERCOMM_CREATE_FROM_GROUPS (see Sec-
\mathbf{2}
     tion 7.6.2) can be used to create inter-communicators.
3
          An intra-communicator involves a single group while an inter-communicator involves
4
     two groups. Where the following discussions address inter-communicator semantics, the
\mathbf{5}
     two groups in an inter-communicator are called the left and right groups. A process in an
6
     inter-communicator is a member of either the left or the right group. From the point of
7
     view of that process, the group that the process is a member of is called the local group; the
8
     other group (relative to that process) is the remote group. The left and right group labels
9
     give us a way to describe the two groups in an inter-communicator that is not relative to
10
     any particular process (as the local and remote groups are).
11
12
     MPI_COMM_DUP(comm, newcomm)
13
14
       IN
                                              communicator (handle)
                 comm
15
       OUT
                                              copy of comm (handle)
                 newcomm
16
17
     C binding
18
     int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
19
20
     Fortran 2008 binding
21
     MPI_Comm_dup(comm, newcomm, ierror)
22
          TYPE(MPI_Comm), INTENT(IN) :: comm
23
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
^{24}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     Fortran binding
26
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
27
          INTEGER COMM, NEWCOMM, IERROR
28
29
          MPI_COMM_DUP duplicates the existing communicator comm with associated key
30
     values and topology information. For each key value, the respective copy callback function
^{31}
     determines the attribute value associated with this key in the new communicator; one
32
     particular action that a copy callback may take is to delete the attribute from the new
33
     communicator. MPI_COMM_DUP returns in newcomm a new communicator with the same
34
     group or groups, same topology, and any copied cached information, but a new context (see
35
     Section 7.7.1).
36
37
           Advice to users. This operation is used to provide a parallel library with a duplicate
38
           communication space that has the same properties as the original communicator. This
           includes any attributes (see below) and topologies (see Chapter 8). This call is valid
39
40
           even if there are pending point-to-point communications involving the communicator
41
           comm. A typical call might involve a MPI_COMM_DUP at the beginning of the
42
           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.
43
44
           This call applies to both intra- and inter-communicators. (End of advice to users.)
45
46
           Advice to implementors. One need not actually copy the group information, but only
47
           add a new reference and increment the reference count. Copy on write can be used
48
           for the cached information. (End of advice to implementors.)
```

## Unofficial Draft for Comment Only

MPI_COM	MM_DUP_WITH_INF	FO(comm, info, newcomm)	1
IN	comm	communicator (handle)	2 3
IN	info	info object (handle)	4
OUT	newcomm	copy of <b>comm</b> (handle)	5
~			6 7
C bindin		Co(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)	8
	-		9
	2008 binding	omm, info, newcomm, ierror)	10 11
	E(MPI_Comm), INTEN		12
	E(MPI_Info), INTEN		13
		NT(OUT) :: newcomm	14
TNLE	GER, OPTIONAL, IN	NTENT(OUT) :: ierror	15
Fortran	0		16 17
		DMM, INFO, NEWCOMM, IERROR)	18
	EGER COMM, INFO, N		19
		I_INFO behaves exactly as MPI_COMM_DUP except that the	20
hints prov	vided by the argumen	at info are associated with the output communicator newcomm.	21 22
Rat	<i>tionale.</i> It is expected	ed that some hints will only be valid at communicator creation	22
	_	cy reasons, most communicator creation calls do not provide	24
	0	nay associate info hints with a duplicate of any communicator	25
at c	reation time through	a call to MPI_COMM_DUP_WITH_INFO. ( <i>End of rationale.</i> )	26
			27
			28 29
MPI_COM	MM_IDUP(comm, nev	wcomm, request)	30
IN	comm	communicator (handle)	31
OUT	newcomm	copy of <b>comm</b> (handle)	32
OUT	request	communication request (handle)	33
			34 35
C bindi	Ŭ		36
int MPI_	_Comm_idup(MPI_Com	m comm, MPI_Comm *newcomm, MPI_Request *request)	37
Fortran	2008 binding		38
	-	omm, request, ierror)	39
	E(MPI_Comm), INTEN		40 41
		NT(OUT), ASYNCHRONOUS :: newcomm NTENT(OUT) :: request	42
	-	ITENT(OUT) :: ierror	43
Fortran			44
		DMM, REQUEST, IERROR)	45
	-	1, REQUEST, IERROR	46 47
			48

1 2 3 4 5 6	of its nonb was execut after MPI_ assumption MPI_COM	locking behavior, the sema ed at the time that MPI_C COMM_IDUP will not be as for nonblocking collect M_IDUP and the returned	*		
7 8 9 10		rroneous to use the comm before the MPI_COMM_ID	unicator <b>newcomm</b> as an input argument to other MPI UP operation completes.		
11	MPI_COM	M_IDUP_WITH_INFO(cor	nm, info, newcomm, request)		
12	IN	comm	communicator (handle)		
13 14	IN	info	info object (handle)		
15	OUT	newcomm	copy of <b>comm</b> (handle)		
16	OUT	request	communication request (handle)		
17 18 19 20	C binding	g comm_idup_with_info(MP	I_Comm comm, MPI_Info info, , MPI_Request *request)		
22 23 24 25 26 27 28 29 30 31	<pre>Fortran 2008 binding MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm     TYPE(MPI_Info), INTENT(IN) :: info     TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm     TYPE(MPI_Request), INTENT(OUT) :: request     INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR)</pre>				
32	INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR				
<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> </ul>	MPI_COM mantics of executed a or info him communica Section 6.1 It is e	M_DUP_WITH_INFO. Wi MPI_COMM_IDUP_WIT t the time that MPI_COMM ts changed after MPI_COMM ator. All restrictions and 2) apply to MPI_COMM_ rroneous to use the comm	is a nonblocking variant of th the exception of its nonblocking behavior, the se- H_INFO are as if MPI_COMM_DUP_WITH_INFO was M_IDUP_WITH_INFO is called. For example, attributes MM_IDUP_WITH_INFO will not be copied to the new assumptions for nonblocking collective operations (see IDUP_WITH_INFO and the returned request. unicator newcomm as an input argument to other MPI UP_WITH_INFO operation completes.		
42 43 44 45 46 47 48	are o		_IDUP and MPI_COMM_IDUP_WITH_INFO functions at of purely nonblocking libraries (see [40]). ( <i>End of</i>		

IN	comm	communicator (handle)		
IN	group	group, which is a subset of the group of <b>comm</b> (handle)		
OUT	newcomm	new communicator (handle)		

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

#### C binding

int MPI\_Comm\_create(MPI\_Comm comm, MPI\_Group group, MPI\_Comm \*newcomm)

## Fortran 2008 binding

MPI\_Comm\_create(comm, group, newcomm, ierror)
 TYPE(MPI\_Comm), INTENT(IN) :: comm
 TYPE(MPI\_Group), INTENT(IN) :: group
 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_COMM\_CREATE(COMM, GROUP, NEWCOMM, IERROR) INTEGER COMM, GROUP, NEWCOMM, IERROR

If comm is an intra-communicator, this function returns a new communicator newcomm with communication group defined by the group argument. No cached information propagates from comm to newcomm and no virtual topology information is added to the created communicator. 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.

## Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

1

2

3

4 5

6

7

8

9

10

11

15

16

17

18

19

20

25

34

35

36

37 38 • 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.*)

Advice to implementors. When calling MPI\_COMM\_DUP, all processes call with the same group (the group associated with the communicator). When calling

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

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

If comm is an inter-communicator, then the output communicator is also an inter-com-26municator where the local group consists only of those processes contained in group (see 27Figure 7.1). The group argument should only contain those processes in the local group of 28the input inter-communicator that are to be a part of newcomm. All processes in the same 29 local group of comm must specify the same value for group, i.e., the same members in the 30 same order. If either group does not specify at least one process in the local group of the  $^{31}$ inter-communicator, or if the calling process is not included in the group, MPI\_COMM\_NULL 32 is returned. 33

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

<sup>39</sup> **Example 7.1** Inter-communicator creation.

The following example illustrates how the first node in the left side of an inter-communicator could be joined with all members on the right side of an inter-communicator to form a new inter-communicator.

- 10
- 44 45
- 46
- 47
- 48

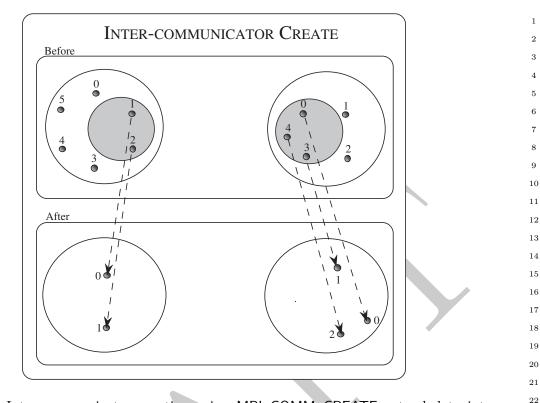


Figure 7.1: Inter-communicator creation using MPI\_COMM\_CREATE extended to intercommunicators. The input groups are those in the grey circle.

```
MPI_Comm inter_comm, new_inter_comm;
MPI_Group local_group, group;
          rank = 0; /* rank on left side to include in
int
                       new inter-comm */
/* Construct the original inter-communicator: "inter_comm" */
. . .
/* Construct the group of processes to be in new
   inter-communicator */
if (/* I'm on the left side of the inter-communicator */) {
  MPI_Comm_group(inter_comm, &local_group);
  MPI_Group_incl(local_group, 1, &rank, &group);
  MPI_Group_free(&local_group);
}
else
  MPI_Comm_group(inter_comm, &group);
MPI_Comm_create(inter_comm, group, &new_inter_comm);
MPI_Group_free(&group);
```

23

 $\frac{24}{25}$ 

26

27

28

29 30

 $^{31}$ 

32 33

34

35

36

37

38

39

40

41

42 43

44

328 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

1 MPI\_COMM\_CREATE\_GROUP(comm, group, tag, newcomm) 2 IN intra-communicator (handle) comm 3 IN group, which is a subset of the group of comm group 4 (handle) 56 IN tag (integer) tag 7 OUT newcomm new communicator (handle) 8 9 C binding 10 int MPI\_Comm\_create\_group(MPI\_Comm comm, MPI\_Group group, int tag, 11 MPI\_Comm \*newcomm) 1213Fortran 2008 binding 14MPI\_Comm\_create\_group(comm, group, tag, newcomm, ierror) 15TYPE(MPI\_Comm), INTENT(IN) :: comm 16TYPE(MPI\_Group), INTENT(IN) :: group 17INTEGER, INTENT(IN) :: tag 18 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20Fortran binding 21MPI\_COMM\_CREATE\_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) 22 INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR 23 $^{24}$ MPI\_COMM\_CREATE\_GROUP is similar to MPI\_COMM\_CREATE; however, 25MPI\_COMM\_CREATE must be called by all processes in the group of comm, whereas 26MPI\_COMM\_CREATE\_GROUP must be called by all processes in group, which is a subgroup 27of the group of comm. In addition, MPI\_COMM\_CREATE\_GROUP requires that comm is 28an intra-communicator. MPI\_COMM\_CREATE\_GROUP returns a new intra-communicator, 29newcomm, for which the group argument defines the communication group. No cached 30 information propagates from comm to newcomm and no virtual topology information is  $^{31}$ added to the created communicator. Each process must provide a group argument that is a 32 subgroup of the group associated with comm; this could be MPI\_GROUP\_EMPTY. If a non-33 empty group is specified, then all processes in that group must call the function, and each of 34these processes must provide the same arguments, including a group that contains the same 35 members with the same ordering. Otherwise the call is erroneous. If the calling process is a 36 member of the group given as the group argument, then newcomm is a communicator with 37 group as its associated group. If the calling process is not a member of group, e.g., group is 38 MPI\_GROUP\_EMPTY, then the call is a local operation and MPI\_COMM\_NULL is returned as 39 newcomm. 40 41 Functionality similar to MPI\_COMM\_CREATE\_GROUP can be imple-Rationale. 42mented through repeated MPI\_INTERCOMM\_CREATE and 43 MPI\_INTERCOMM\_MERGE calls that start with the MPI\_COMM\_SELF communicators 44 at each process in group and build up an intra-communicator with group group [17]. 45Such an algorithm requires the creation of many intermediate communicators; 46MPI\_COMM\_CREATE\_GROUP can provide a more efficient implementation that avoids

- <sup>47</sup> this overhead. (*End of rationale.*)
- 48

Advice to users. An inter-communicator can be created collectively over processes in the union of the local and remote groups by creating the local communicator using MPI\_COMM\_CREATE\_GROUP and using that communicator as the local communicator argument to MPI\_INTERCOMM\_CREATE. (*End of advice to users.*)

The tag argument does not conflict with tags used in point-to-point communication and is not permitted to be a wildcard. If multiple threads at a given process perform concurrent MPI\_COMM\_CREATE\_GROUP operations, the user must distinguish these operations by providing different tag or comm arguments.

*Advice to users.* MPI\_COMM\_CREATE may provide lower overhead than MPI\_COMM\_CREATE\_GROUP because it can take advantage of collective communication on comm when constructing newcomm. (*End of advice to users.*)

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

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assignment (integer)
OUT	newcomm	new communicator (handle)

## C binding

int MPI\_Comm\_split(MPI\_Comm comm, int color, int key, MPI\_Comm \*newcomm)

#### Fortran 2008 binding

<pre>MPI_Comm_split(comm, color, key, newcomm, ierror)</pre>
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: color, key
TYPE(MPI_Comm), INTENT(OUT) :: newcomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

## Fortran binding

```
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. No cached information propagates from comm to newcomm and no virtual topology information is added to the created communicators.

With an intra-communicator 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

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4 5

6

 $\overline{7}$ 

8

9 10

11

12

 $^{24}$ 

25 26

33

34

35 36

37

38

39

40

41

42

43

44

45

46

47

(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.

- Advice to users. This is an extremely powerful mechanism for dividing a single com-5municating group of processes into k subgroups, with k chosen implicitly by the user 6 (by the number of colors asserted over all the processes). Each resulting communica-7 tor will be non-overlapping. Such a division could be useful for defining a hierarchy 8 of computations, such as for multigrid, or linear algebra. For intra-communicators, 9 MPI\_COMM\_SPLIT provides similar capability as MPI\_COMM\_CREATE to split a 10 communicating group into disjoint subgroups. MPI\_COMM\_SPLIT is useful when 11 some processes do not have complete information of the other members in their 12group, but all processes know (the color of) the group to which they belong. In 13 this case, the MPI implementation discovers the other group members via communi-14cation. MPI\_COMM\_CREATE is useful when all processes have complete information 15of the members of their group. In this case, MPI can avoid the extra communica-16tion required to discover group membership. MPI\_COMM\_CREATE\_GROUP is useful 17 when all processes in a given group have complete information of the members of their 18 19 group and synchronization with processes outside the group can be avoided.
- <sup>20</sup> Multiple calls to MPI\_COMM\_SPLIT can be used to overcome the requirement that <sup>21</sup> any call have no overlap of the resulting communicators (each process is of only one <sup>22</sup> color per call). In this way, multiple overlapping communication structures can be <sup>23</sup> 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 inter-communicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the inter-communicator (see Figure 7.2). For those colors that are specified only on one side of the inter-communicator, MPI\_COMM\_NULL is returned. MPI\_COMM\_NULL is also returned to those processes that specify MPI\_UNDEFINED as the color.

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

47 48

43

44

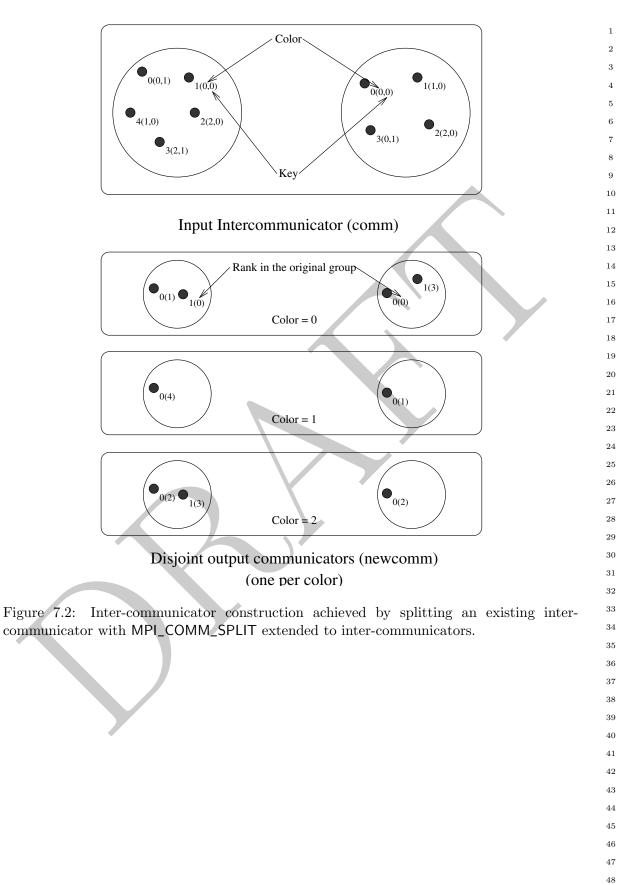
45

46

33

34

35 36



```
1
     Example 7.2 Parallel client-server model.
\mathbf{2}
     The following client code illustrates how clients on the left side of an inter-communicator
3
     could be assigned to a single server from a pool of servers on the right side of an inter-
4
     communicator.
5
              /* Client code */
6
              MPI_Comm multiple_server_comm;
7
              MPI_Comm
                         single_server_comm;
8
              int
                         color, rank, num_servers;
9
10
              /* Create inter-communicator with clients and servers:
11
                 multiple_server_comm */
12
              . . .
13
14
              /* Find out the number of servers available */
15
              MPI_Comm_remote_size(multiple_server_comm, &num_servers);
16
17
              /* Determine my color */
18
              MPI_Comm_rank(multiple_server_comm, &rank);
19
              color = rank % num_servers;
20
21
              /* Split the inter-communicator */
22
              MPI_Comm_split(multiple_server_comm, color, rank,
23
                               &single_server_comm);
24
25
     The following is the corresponding server code:
26
27
              /* Server code */
              MPI_Comm multiple_client_comm;
28
29
              MPI_Comm single_server_comm;
30
              int
                         rank;
31
32
              /* Create inter-communicator with clients and servers:
33
                 multiple_client_comm */
34
              . . .
35
36
              /* Split the inter-communicator for a single server per group
37
                 of clients */
38
              MPI_Comm_rank(multiple_client_comm, &rank);
39
              MPI_Comm_split(multiple_client_comm, rank, 0,
40
                               &single_server_comm);
41
42
43
44
45
46
47
48
```

N	MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)			
	IN	comm	communicator (handle)	
	IN	split_type	type of processes to be grouped together (integer)	
	IN	key control of rank assignment (integer)		
	INOUT	info	info argument (handle)	
	OUT	newcomm	new communicator (handle)	

# · c

#### C binding

int MPI\_Comm\_split\_type(MPI\_Comm comm, int split\_type, int key, MPI\_Info info, MPI\_Comm \*newcomm)

#### Fortran 2008 binding

MPI\_Comm\_split\_type(comm, split\_type, key, info, newcomm, ierror) TYPE(MPI\_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: split\_type, key TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_Comm), INTENT(OUT) :: newcomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror

## Fortran binding

MPI\_COMM\_SPLIT\_TYPE(COMM, SPLIT\_TYPE, KEY, INFO, NEWCOMM, IERROR) INTEGER COMM, SPLIT\_TYPE, KEY, INFO, NEWCOMM, IERROR

This function partitions the group associated with comm into disjoint subgroups such that each subgroup contains all MPI processes in the same grouping referred to by split\_type. Within each subgroup, the MPI processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in **newcomm**. This is a collective call. All MPI processes in the group associated with comm must provide the same split\_type, but each MPI process is permitted to provide different values for key. An exception to this rule is that an MPI process may supply the type value MPI\_UNDEFINED, in which case MPI\_COMM\_NULL is returned in newcomm for such MPI process. No cached information propagates from comm to newcomm and no virtual topology information is added to the created communicators.

For split\_type, the following values are defined by MPI:

MPI\_COMM\_TYPE\_SHARED—all MPI processes in newcomm can create a shared memory segment (e.g., with a successful call to MPI\_WIN\_ALLOCATE\_SHARED). This segment can subsequently be used for load/store accesses by all MPI processes in newcomm.

> Since the location of some of the MPI processes may change Advice to users. during the application execution, the communicators created with the value MPI\_COMM\_TYPE\_SHARED before this change may not reflect an actual ability to share memory between MPI processes after this change. (End of advice to users.)

MPI\_COMM\_TYPE\_HW\_GUIDED—this value specifies that the communicator comm is split according to a hardware resource type (for example a computing core or an L3

## **Unofficial Draft for Comment Only**

1 2

4 5 6

9

10

11

1213

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

4647

1 cache) specified by the "mpi\_hw\_resource\_type" info key. Each output communicator 2 newcomm corresponds to a single instance of the specified hardware resource type. 3 The MPI processes in the group associated with the output communicator newcomm 4 utilize that specific hardware resource type instance, and no other instance of the 5same hardware resource type. 6 If an MPI process does not meet the above criteria, then MPI\_COMM\_NULL is returned 7 in newcomm for such process. 8 MPI\_COMM\_NULL is also returned in **newcomm** in the following cases: 9 10 • No info key is provided. 11 • The info handle does not include the key "mpi\_hw\_resource\_type". 12• The MPI implementation neither recognizes nor supports the info key 13 14"mpi\_hw\_resource\_type". 15• The MPI implementation does not recognize the value associated with the info 16key "mpi\_hw\_resource\_type". 17 The MPI implementation will return in the group of the output communicator 18 newcomm the largest subset of MPI processes that match the splitting criterion. 19 20The processes in the group associated with newcomm are ranked in the order defined 21by the value of the argument key with ties broken according to their rank in the group 22 associated with comm. 2324Advice to users. The set of hardware resources that an MPI process is able to 25utilize may change during the application execution (e.g., because of the reloca-26tion of an MPI process), in which case the communicators created with the value 27MPI\_COMM\_TYPE\_HW\_GUIDED before this change may not reflect the utiliza-28 tion of hardware resources of such process at any time after the communicator 29 creation. (End of advice to users.) 30 31The user explicitly constrains with the info argument the splitting of the input communicator comm. To this end, the info key "mpi\_hw\_resource\_type" is reserved and 32 33 its associated value is an implementation-defined string designating the type of the 34 requested hardware resource (e.g., "NUMANode", "Package" or "L3Cache"). 35The value "mpi\_shared\_memory" is reserved and its use is equivalent to using 36 MPI\_COMM\_TYPE\_SHARED for the split\_type parameter. 37 38 *Rationale.* The value "mpi\_shared\_memory" is defined in order to ensure consis-39 tency between the use of MPI\_COMM\_TYPE\_SHARED and the use of 40 MPI\_COMM\_TYPE\_HW\_GUIDED. (*End of rationale.*) 41 All MPI processes must provide the same value for the info key "mpi\_hw\_resource\_type". 4243 **Example 7.3** Splitting MPI\_COMM\_WORLD into NUMANode subcommunicators. 44 45 46 47 48

MPI\_COMM\_TYPE\_HW\_UNGUIDED—the group of MPI processes associated with newcomm must be a *strict* subset of the group associated with comm and each newcomm corresponds to a single instance of a **hardware resource type** (for example a computing core or an L3 cache).

All MPI processes in the group associated with comm which utilize that specific hardware resource type instance – and no other instance of the same hardware resource type – are included in the group of newcomm.

If a given MPI process cannot be a member of a communicator that forms such a strict subset, or does not meet the above criteria, then MPI\_COMM\_NULL is returned in newcomm for this process.

Advice to implementors. In a high-quality MPI implementation, the number of different new valid communicators **newcomm** produced by this splitting operation should be minimal unless the user provides a key/value pair that modifies this behavior. The sets of hardware resource types used for the splitting operation are implementation-dependent, but should reflect the hardware of the actual system on which the application is currently executing. (*End of advice to implementors.*)

Rationale. If the hardware resources are hierarchically organized, calling this routine several times using as its input communicator comm the output communicator newcomm of the previous call creates a sequence of newcomm communicators in each MPI process, which exposes a hierarchical view of the hardware platform, as shown in Example 7.4. This sequence of returned newcomm communicators may differ from the sets of hardware resource types, as shown in the second splitting operation in Figure 7.3. (End of rationale.)

Advice to users. Each output communicator newcomm can represent a different hardware resource type (see Figure 7.3 for an example). The set of hardware resources an MPI process utilizes may change during the application execution (e.g., because of process relocation), in which case the communicators created with the value MPI\_COMM\_TYPE\_HW\_UNGUIDED before this change may not reflect the utilization of hardware resources for such process at any time after the communicator creation. (*End of advice to users.*)

If a valid info handle is provided as an argument, the MPI implementation sets the info key "mpi\_hw\_resource\_type" for each MPI process in the group associated with a

#### Unofficial Draft for Comment Only

a

 $46 \\ 47$ 

returned **newcomm** communicator and the info key value is an implementation-defined string that indicates the hardware resource type represented by **newcomm**. The same hardware resource type must be set in all MPI processes in the group associated with **newcomm**.

**Example 7.4** Recursive splitting of MPI\_COMM\_WORLD.

```
#define MAX_NUM_LEVELS 32
```

1

2

3

4

5

7 8

9 10

11

12 13

14 15

16

17

18

19

20

21

22

23

24

25 26 27

28

29

30

31

32

33 34

```
MPI_Comm hwcomm[MAX_NUM_LEVELS];
```

```
int rank, level_num = 0;
```

hwcomm[level\_num] = MPI\_COMM\_WORLD;

```
level_num++;
```

}

Advice to implementors. Implementations can define their own split\_type values, or use the info argument, to assist in creating communicators that help expose platformspecific information to the application. The concept of hardware-based communicators was first described by Träff [67] for SMP systems. Guided and unguided modes description as well as an implementation path are introduced by Goglin *et al.* [27]. (*End of advice to implementors.*)

&hwcomm[level\_num+1]);

```
35
36
      MPI_COMM_CREATE_FROM_GROUP(group, stringtag, info, errhandler, newcomm)
37
        IN
                  group
                                                group (handle)
38
        IN
                  stringtag
                                                unique identifier for this operation (string)
39
40
                  info
        IN
                                                info object (handle)
41
        IN
                  errhandler
                                                error handler to be attached to new
42
                                                intra-communicator (handle)
43
        OUT
                  newcomm
                                                new communicator (handle)
44
45
46
      C binding
47
      int MPI_Comm_create_from_group(MPI_Group group, const char *stringtag,
48
```

```
MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm)
```

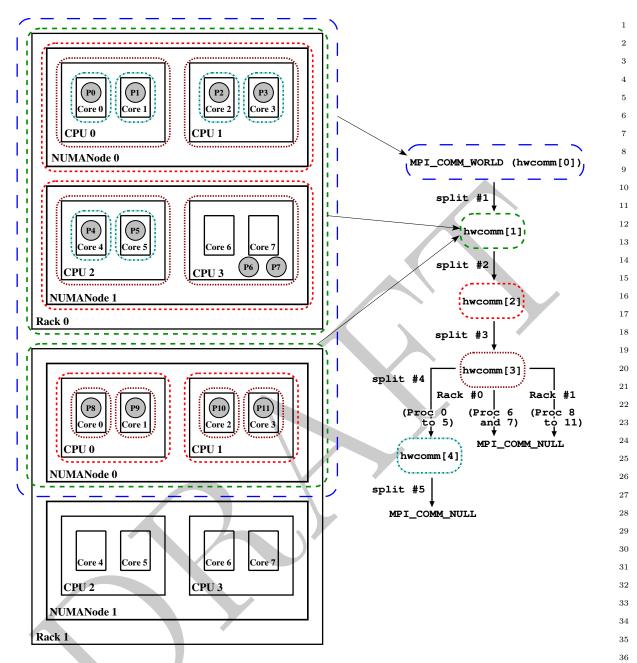


Figure 7.3: Recursive splitting of MPI\_COMM\_WORLD with MPI\_COMM\_SPLIT\_TYPE and MPI\_COMM\_TYPE\_HW\_UNGUIDED. Dashed lines represent communicators whilst solid lines represent hardware resources. MPI processes (P0 to P11) utilize exclusively their respective core, except for P6 and P7 which utilize CPU #3 of Rack #0 and can therefore use Cores #6 and #7 indifferently. The second splitting operation yields two subcommunicators corresponding to NUMANodes in Rack #0 and to CPUs in Rack #1 because Rack #1 features only one NUMANode which corresponds to the whole portion of the Rack that is included in MPI\_COMM\_WORLD and hwcomm[1]. For the first splitting operation, the hardware resource type returned in the info argument is "Rack" on the processes on Rack #0, whereas on Rack #1, it can be either "Rack" or "NUMANode".

```
1
              Fortran 2008 binding
         \mathbf{2}
              MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm,
         3
                              ierror)
         4
                   TYPE(MPI_Group), INTENT(IN) :: group
         5
                   CHARACTER(LEN=*), INTENT(IN) :: stringtag
         6
                   TYPE(MPI_Info), INTENT(IN) :: info
         7
                   TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
         8
                   TYPE(MPI_Comm), INTENT(OUT) :: newcomm
         9
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
         10
              Fortran binding
         11
              MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM,
         12
                              IERROR)
         13
                   INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR
         14
                   CHARACTER*(*) STRINGTAG
         15
         16
                   MPI_COMM_CREATE_FROM_GROUP is similar to MPI_COMM_CREATE_GROUP, ex-
         17
              cept that the set of MPI processes involved in the creation of the new intra-communicator
         18
              is specified by a group argument, rather than the group associated with a pre-existing com-
         19
              municator. If a non-empty group is specified, then all MPI processes in that group must call
         20
              the function and each of these MPI processes must provide the same arguments, including
         21
              a group that contains the same members with the same ordering, and identical stringtag
         22
              value. In the event that MPI_GROUP_EMPTY is supplied as the group argument, then the
         23
              call is a local operation and MPI_COMM_NULL is returned as newcomm. The stringtag argu-
         ^{24}
              ment is analogous to the tag used for MPI_COMM_CREATE_GROUP. If multiple threads at
         25
              a given MPI process perform concurrent MPI_COMM_CREATE_FROM_GROUP operations,
         26
              the user must distinguish these operations by providing different stringtag arguments. The
         27
              stringtag shall not exceed MPI_MAX_FROM_GROUP_TAG characters in length. For C, this
         28
              includes space for a null terminating character. The errhandler argument specifies an error
#-update<sup>5</sup>9
              handler to be attached to the new intra-communicator. This error handler will also be in-
         30
              voked if the MPI_COMM_CREATE_FROM_GROUP function encounters an error. The info
              argument provides hints and assertions, possibly MPI implementation dependent, which
              indicate desired characteristics and guide communicator creation.
                                    The stringtag argument is used to distinguish concurrent commu-
                    Advice to users.
incomplete?
                    nicator construction operations issued by different entities. As such, it is important
         36
                    to ensure that this argument is unique for each concurrent call to
         37
                    MPI_COMM_CREATE_FROM_GROUP. Reverse domain name notation convention [1]
         38
                    is one approach to constructing unique stringtag arguments. See also example 11.8.
                    (End of advice to users.)
         39
         40
         ^{41}
              7.4.3 Communicator Destructors
         42
         43
         44
              MPI_COMM_FREE(comm)
         45
```

```
INOUT
                comm
46
```

C binding

4748

#-other

#-error!

#-TODO

Merge of

Is PR400

PR400

communicator to be destroyed (handle)

int MPI\_Comm\_free(MPI\_Comm \*comm)

#### Fortran 2008 binding

MPI\_Comm\_free(comm, ierror)
 TYPE(MPI\_Comm), INTENT(INOUT) :: comm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_COMM\_FREE(COMM, IERROR) INTEGER COMM, IERROR

This collective operation marks the communication object for deallocation. The handle is set to MPI\_COMM\_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 7.7) are called in arbitrary order.

Advice to implementors. Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

#### 7.4.4 Communicator Info

Hints specified via info (see Chapter 10) 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\_SET\_INFO, MPI\_COMM\_SET\_INFO, MPI\_COMM\_SET\_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 47 on the given communicator. 48

#### Unofficial Draft for Comment Only

 $^{31}$ 

"mpi\_assert\_exact\_length" (boolean, default: "false"): If set to "true", then the imple mentation 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 group of the communicator provide a given info key with the same value. (End of advice to users.)

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

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

```
^{26}_{27} C binding
```

5

13

14

15 16

17

18

19 20 21

 $^{31}$ 

32

33

34

36

37

```
int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
```

```
<sup>29</sup> Fortran 2008 binding
```

```
MPI_Comm_set_info(comm, info, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
```

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

35 Fortran binding

```
MPI_COMM_SET_INFO(COMM, INFO, IERROR)
INTEGER COMM, INFO, IERROR
```

<sup>38</sup> MPI\_COMM\_SET\_INFO updates the hints of the communicator associated with comm <sup>39</sup> using the hints provided in info. This operation has no effect on previously set or defaulted <sup>40</sup> hints that are not specified by info. It also has no effect on previously set or defaulted <sup>41</sup> hints that are specified by info, but are ignored by the MPI implementation in this call to <sup>42</sup> MPI\_COMM\_SET\_INFO. MPI\_COMM\_SET\_INFO is a collective routine. The info object <sup>43</sup> may be different on each process, but any info entries that an implementation requires to <sup>44</sup> be the same on all processes must appear with the same value in each process's info object.

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

## Unofficial Draft for Comment Only

## 7.5. MOTIVATING EXAMPLES

accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI\_COMM\_SET\_INFO. MPI\_COMM\_GET\_INFO can be used to determine whether updates to existing info hints were ignored by the implementation. (*End of advice to users.*)

Advice to users. Setting info hints on the predefined communicators MPI\_COMM\_WORLD and MPI\_COMM\_SELF may have unintended effects, as changes to these global objects may affect all components of the application, including libraries and tools. Users must ensure that all components of the application that use a given communicator, including libraries and tools, can comply with any info hints associated with that communicator. (*End of advice to users.*)

object (handle)

MPI_CO	DMM_GET_INFO(comr	n, info_used)
IN	comm	$\operatorname{communicator}$

OUT	info_used	new info object (handle)
-----	-----------	--------------------------

#### C binding

int MPI\_Comm\_get\_info(MPI\_Comm comm, MPI\_Info \*info\_used)

#### Fortran 2008 binding

<pre>MPI_Comm_get_info(comm, info_used, ierror)</pre>	
TYPE(MPI_Comm), INTENT(IN) :: comm	
TYPE(MPI_Info), INTENT(OUT) :: info_used	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

## Fortran binding

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.

# 7.5 Motivating Examples

```
7.5.1 Current Practice #1
```

Example #1a:

```
int main(int argc, char *argv[])
{
    int me, size;
```

 $\overline{7}$ 

```
1
           . . .
\mathbf{2}
           MPI_Init(&argc, &argv);
3
           MPI_Comm_rank(MPI_COMM_WORLD, &me);
4
           MPI_Comm_size(MPI_COMM_WORLD, &size);
5
6
           (void)printf("Process %d size %d\n", me, size);
7
8
           MPI_Finalize();
9
           return 0;
10
         }
11
     Example #1a is a do-nothing program that initializes itself, and refers to the "all" commu-
12
     nicator, and prints a message. It terminates itself too. This example does not imply that
13
     MPI supports printf-like communication itself.
14
     Example #1b: Message exchange (supposing that size is even)
15
16
          int main(int argc, char *argv[])
17
          {
18
             int me, size;
19
             int SOME_TAG = 0;
20
              . . .
21
             MPI_Init(&argc, &argv);
22
23
             MPI_Comm_rank(MPI_COMM_WORLD, &me);
                                                        /* local */
24
             MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
25
26
             if((me % 2) == 0)
27
             ſ
28
                 /* send unless highest-numbered process */
29
                 if((me + 1) < size)
30
                    MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
31
             }
32
              else
33
                                     - 1, SOME_TAG, MPI_COMM_WORLD, &status);
                 MPI_Recv(..., me
34
35
36
             MPI_Finalize();
37
             return 0;
38
          }
39
40
     Example #1b schematically illustrates message exchanges between "even" and "odd" pro-
41
     cesses in the "all" communicator.
42
43
            Current Practice #2
     7.5.2
44
         int main(int argc, char *argv[])
45
         {
46
           int me, count;
47
           void *data;
48
```

```
1
     . . .
                                                                                        2
                                                                                        3
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &me);
                                                                                        4
                                                                                        5
     if(me == 0)
                                                                                        6
     ſ
                                                                                        7
          /* get input, create buffer ''data'' */
                                                                                        9
          . . .
     }
                                                                                        10
                                                                                        11
     MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
                                                                                        12
                                                                                        13
                                                                                        14
      . . .
                                                                                        15
     MPI_Finalize();
                                                                                        16
     return 0;
                                                                                        17
   }
                                                                                        18
This example illustrates the use of a collective communication.
                                                                                        19
                                                                                        20
7.5.3 (Approximate) Current Practice #3
                                                                                        21
                                                                                        22
  int main(int argc, char *argv[])
                                                                                        23
  {
                                                                                        ^{24}
    int me, count, count2;
                                                                                        25
    void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
                                                                                        26
    MPI_Group group_world, grprem;
                                                                                        27
    MPI_Comm commWorker;
                                                                                        28
    static int ranks[] = {0};
                                                                                        29
    . . .
                                                                                        30
    MPI_Init(&argc, &argv);
                                                                                        31
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);
                                                                                        32
    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
                                                                                        33
                                                                                        34
    MPI_Group_excl(group_world, 1, ranks, &grprem); /* local */
                                                                                        35
    MPI_Comm_create(MPI_COMM_WORLD, grprem, &commWorker);
                                                                                        36
                                                                                        37
    if(me != 0)
                                                                                        38
    {
                                                                                        39
      /* compute on worker */
                                                                                        40
       . . .
                                                                                        41
      MPI_Reduce(send_buf,recv_buf,count, MPI_INT, MPI_SUM, 1, commWorker);
                                                                                        42
      . . .
                                                                                        43
      MPI_Comm_free(&commWorker);
                                                                                        44
    }
                                                                                        45
    /* zero falls through immediately to this reduce, others do later... */
                                                                                        46
    MPI_Reduce(send_buf2, recv_buf2, count2,
                                                                                        47
                MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
                                                                                        48
```

```
1
\mathbf{2}
          MPI_Group_free(&group_world);
3
          MPI_Group_free(&grprem);
4
          MPI_Finalize();
5
          return 0;
6
        }
7
     This example illustrates how a group consisting of all but the zeroth process of the "all"
8
     group is created, and then how a communicator is formed (commWorker) for that new group.
9
     The new communicator is used in a collective call, and all processes execute a collective call
10
     in the MPI_COMM_WORLD context. This example illustrates how the two communicators
11
     (that inherently possess distinct contexts) protect communication. That is, communication
12
     in MPI_COMM_WORLD is insulated from communication in commWorker, and vice versa.
13
          In summary, "group safety" is achieved via communicators because distinct contexts
14
     within communicators are enforced to be unique on any process.
15
16
17
     7.5.4 Example \#4
18
     The following example is meant to illustrate "safety" between point-to-point and collective
19
     communication. MPI guarantees that a single communicator can do safe point-to-point and
20
     collective communication.
21
22
         #define TAG_ARBITRARY 12345
23
         #define SOME_COUNT
                                      50
24
25
         int main(int argc, char *argv[])
26
         ſ
27
           int me;
28
           MPI_Request request[2];
29
           MPI_Status status[2];
30
           MPI_Group group_world, subgroup;
31
           int ranks[] = \{2, 4, 6, 8\};
32
           MPI_Comm the_comm;
33
           . . .
34
           MPI_Init(&argc, &argv);
35
           MPI_Comm_group(MPI_COMM_WORLD, &group_world);
36
37
           MPI_Group_incl(group_world, 4, ranks, &subgroup); /* local */
38
           MPI_Group_rank(subgroup, &me);
                                                   /* local */
39
40
           MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
41
42
           if(me != MPI_UNDEFINED)
43
           {
44
                MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
45
                                     the_comm, request);
46
                MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
47
                                     the_comm, request+1);
48
                for(i = 0; i < SOME_COUNT; i++)</pre>
```

Unofficial Draft for Comment Only

```
1
            MPI_Reduce(..., the_comm);
                                                                                             \mathbf{2}
          MPI_Waitall(2, request, status);
                                                                                             3
          MPI_Comm_free(&the_comm);
                                                                                             4
     }
                                                                                             5
                                                                                             6
     MPI_Group_free(&group_world);
                                                                                             7
     MPI_Group_free(&subgroup);
                                                                                             8
     MPI_Finalize();
                                                                                             9
                                                                                             10
     return 0;
   }
                                                                                             11
                                                                                             12
                                                                                             13
7.5.5
       Library Example #1
                                                                                             14
The main program:
                                                                                             15
                                                                                             16
   int main(int argc, char *argv[])
                                                                                             17
   {
                                                                                             18
      int done = 0;
                                                                                             19
     user_lib_t *libh_a, *libh_b;
                                                                                             20
     void *dataset1, *dataset2;
                                                                                             21
      . . .
                                                                                             22
     MPI_Init(&argc, &argv);
                                                                                             23
      . . .
                                                                                             ^{24}
      init_user_lib(MPI_COMM_WORLD, &libh_a);
                                                                                             25
      init_user_lib(MPI_COMM_WORLD, &libh_b);
                                                                                             26
      . . .
                                                                                             27
     user_start_op(libh_a, dataset1);
                                                                                             28
     user_start_op(libh_b, dataset2);
                                                                                             29
      . . .
                                                                                             30
     while(!done)
                                                                                             ^{31}
      {
                                                                                             32
         /* work */
                                                                                             33
         . . .
                                                                                             34
         MPI_Reduce(..., MPI_COMM_WORLD);
                                                                                             35
                                                                                             36
         /* see if done */
                                                                                             37
          . .
                                                                                             38
      }
                                                                                             39
     user_end_op(libh_a);
                                                                                             40
     user_end_op(libh_b);
                                                                                             41
                                                                                             42
     uninit_user_lib(libh_a);
                                                                                             43
     uninit_user_lib(libh_b);
                                                                                             44
     MPI_Finalize();
                                                                                             45
     return 0;
                                                                                             46
   }
                                                                                             47
```

The user library initialization code:

```
1
        void init_user_lib(MPI_Comm comm, user_lib_t **handle)
\mathbf{2}
        {
3
           user_lib_t *save;
4
5
           user_lib_initsave(&save); /* local */
6
           MPI_Comm_dup(comm, &(save->comm));
7
8
           /* other inits */
9
           . . .
10
11
           *handle = save;
12
        }
13
     User start-up code:
14
15
        void user_start_op(user_lib_t *handle, void *data)
16
        {
17
           MPI_Irecv( ..., handle->comm, &(handle->irecv_handle) );
18
          MPI_Isend( ..., handle->comm, &(handle->isend_handle) );
19
        }
20
     User communication clean-up code:
21
22
        void user_end_op(user_lib_t *handle)
23
        {
24
          MPI_Status status;
25
          MPI_Wait(&handle->isend_handle, &status);
26
           MPI_Wait(&handle->irecv_handle, &status);
27
        }
28
29
     User object clean-up code:
30
        void uninit_user_lib(user_lib_t *handle)
^{31}
        {
32
          MPI_Comm_free(&(handle->comm));
33
           free(handle);
34
       }
35
36
     7.5.6 Library Example #2
37
38
     The main program:
39
40
         int main(int argc, char *argv[])
^{41}
         ſ
42
           int ma, mb;
43
          MPI_Group group_world, group_a, group_b;
44
          MPI_Comm comm_a, comm_b;
45
46
           static int list_a[] = {0, 1};
47
     #if defined(EXAMPLE_2B) || defined(EXAMPLE_2C)
48
           static int list_b[] = {0, 2,3};
```

Unofficial Draft for Comment Only

```
#else/* EXAMPLE_2A */
     static int list_b[] = \{0, 2\};
#endif
     int size_list_a = sizeof(list_a)/sizeof(int);
     int size_list_b = sizeof(list_b)/sizeof(int);
     MPI_Init(&argc, &argv);
     MPI_Comm_group(MPI_COMM_WORLD, &group_world);
     MPI_Group_incl(group_world, size_list_a, list_a, &group_a);
     MPI_Group_incl(group_world, size_list_b, list_b, &group_b);
     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
     if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
     if(comm_b != MPI_COMM_NULL)
        MPI_Comm_rank(comm_b, &mb);
     if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);
     if(comm_b != MPI_COMM_NULL)
     {
       lib_call(comm_b);
       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)
   ſ
     int me, done = 0;
     MPI_Status status;
     MPI_Comm_rank(comm, &me);
     if(me == 0)
```

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 44

 $45 \\ 46$ 

47

```
1
              while(!done)
2
              {
3
                 MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
4
5
              }
6
           else
7
           {
8
             /* work */
9
             MPI_Send(..., 0, ARBITRARY_TAG, comm);
10
11
           }
12
     #ifdef EXAMPLE_2C
13
           /* include (resp, exclude) for safety (resp, no safety): */
14
          MPI_Barrier(comm);
15
     #endif
16
        }
17
```

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 [64]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity—deleting either feature removes the guarantee that back-masking cannot be required.

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

All of the foregoing is a supposition of "collective calls" implemented with point-topoint operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 7.9.

39 40

41

# 7.6 Inter-Communication

This section introduces the concept of inter-communication and describes the portions of
 MPI that support it. It describes support for writing programs that contain user-level
 servers.

<sup>45</sup> All communication described thus far has involved communication between processes <sup>46</sup> that are members of the same group. This type of communication is called "intra-com-<sup>47</sup> munication" and the communicator used is called an "intra-communicator," as we have <sup>48</sup> noted earlier in the chapter.

## Unofficial Draft for Comment Only

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

An **inter-communication** is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the "local group," that is, the sender in a send and the receiver in a receive. The group containing the target process is called the "remote group," that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (**communicator**, **rank**) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking except for MPI\_COMM\_IDUP and require that the local and remote groups be disjoint.

Advice to users. The groups must be disjoint for several reasons. Primarily, this is the intent of the inter-communicators—to provide a communicator for communication between disjoint groups. This is reflected in the definition of MPI\_INTERCOMM\_MERGE, which allows the user to control the ranking of the processes in the created intra-communicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to inter-communicators makes the most sense when the groups are disjoint. (*End of advice to users.*)

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point and collective communication is the same for both interand intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- A communicator will provide either intra- or inter-communication, never both.

The routine MPI\_COMM\_TEST\_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI\_CART\_CREATE).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8 9

10 11

12

13

14 15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28 29

30

31

32 33

34

35

36 37

38

39

40 41

42

43

44

 $45 \\ 46$ 

47

1	grou	מו			
2	send_context				
3					
4	rece	receive_context			
5	soui	rce			
6	For i	inter-communica <sup>1</sup>	tors, <i>aroup</i> de	scribes the remote group, and <i>source</i> is the rank of	
7 8			, 0 1	r intra-communicators, group is the communicator	
9		=		e rank of the process in this group, and <i>send context</i>	
10	and	receive context a	re identical. A	A group can be represented by a rank-to-absolute-	
11	addr	ess translation ta	able.		
12	The	inter-communica	tor cannot be	discussed sensibly without considering processes in	
13				Imagine a process $\mathbf{P}$ in group $\mathcal{P}$ , which has an inter-	
14			and a process	${\bf Q}$ in group ${\cal Q},$ which has an inter-communicator	
15	$\mathbf{C}_{\mathcal{Q}}.$	Then			
16	•	C <sub>p</sub> .group desc	ribes the grou	p $\mathcal{Q}$ and $\mathbf{C}_{\mathcal{O}}$ .group describes the group $\mathcal{P}$ .	
17			_	ceive_context and the context is unique in $Q$ ;	
18 19	•			$\mathbf{\mathcal{L}}_{\mathbf{\mathcal{L}}}$ send_context and the context is unique in $\mathcal{P}$ .	
20					
21	•	$C_{\mathcal{P}}$ .source is ra	and of $\mathbf{P}$ in $P$	and $\mathbf{C}_{\mathcal{Q}}$ .source is rank of $\mathbf{Q}$ in $\mathcal{Q}$ .	
22	Assu	$\mathbf{P}$ ime that $\mathbf{P}$ send	s a message t	o $\mathbf{Q}$ using the inter-communicator. Then $\mathbf{P}$ uses	
23	the g	<b>group</b> table to f	find the absol	ute address of $\mathbf{Q}$ ; source and send_context are	
24	appe	ended to the mes	sage.		
25	Assu	Assume that $\mathbf{Q}$ posts a receive with an explicit source argument using the inter-			
26				eceive_context to the message context and source	
27	argument to the message source.				
28 29	The same algorithm is appropriate for intra-communicators as well.				
30	In order to support inter-communicator accessors and constructors, it is necessary to				
31	supplement this model with additional structures, that store information about the				
32	local communication group, and additional safe contexts. (End of advice to imple-				
33	men	tors.)			
34	7.6.1 Int	er-Communicato			
35 36	7.0.1 III	er-Communicate	or Accessors		
37					
38		IM_TEST_INTER	R(comm_flag)		
39			(comm, mag)		
40	IN	comm		communicator (handle)	
41	OUT	flag		true if $comm$ is an inter-communicator (logical)	
42					
43	C bindin	•			
44	<pre>int MPI_Comm_test_inter(MPI_Comm comm, int *flag)</pre>				
45 46	Fortran 2	2008 binding			
40	MPI_Comm_test_inter(comm, flag, ierror)				
48	TYPE	(MPI_Comm), IN	TENT(IN) ::	comm	

LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

```
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
INTEGER COMM, IERROR
LOGICAL FLAG
```

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

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

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

Table 7.1: 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.

The following accessors provide consistent access to the remote group of an intercommunicator. The following are all local operations.

MPI_COM	1M_REMOTE_SIZE(comm, size	e)	29
IN	comm	inter-communicator (handle)	30
	comm		31
OUT	size	number of processes in the remote group of comm	32
		(integer)	33
			34
C bindin	g		35
int MPI_	Comm_remote_size(MPI_Comm	comm, int *size)	36
<b>D</b> (			37
	2008 binding		38
MPI_Comm	_remote_size(comm, size, i	ierror)	39
TYPE	(MPI_Comm), INTENT(IN) ::	comm	40
INTE	GER, INTENT(OUT) :: size		41
INTE	GER, OPTIONAL, INTENT(OUT)	) :: ierror	42
Fortran	binding		43
	_REMOTE_SIZE(COMM, SIZE, ]	(ERROR)	44
_	GER COMM, SIZE, IERROR		45
			46
			47

1 MPI\_COMM\_REMOTE\_GROUP(comm, group) 2 IN inter-communicator (handle) comm 3 OUT remote group corresponding to comm (handle) group 4 56 C binding  $\overline{7}$ int MPI\_Comm\_remote\_group(MPI\_Comm comm, MPI\_Group \*group) 8 Fortran 2008 binding 9 MPI\_Comm\_remote\_group(comm, group, ierror) 10 TYPE(MPI\_Comm), INTENT(IN) :: comm 11 TYPE(MPI\_Group), INTENT(OUT) :: group 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14Fortran binding MPI\_COMM\_REMOTE\_GROUP(COMM, GROUP, IERROR) 1516INTEGER COMM, GROUP, IERROR 1718 Symmetric access to both the local and remote groups of an inter-Rationale. 19 communicator is important, so this function, as well as MPI\_COMM\_REMOTE\_SIZE 20have been provided. (End of rationale.) 2122Inter-Communicator Operations 7.6.2 23 $^{24}$ This section introduces five blocking inter-communicator operations. 25MPI\_INTERCOMM\_CREATE is used to bind two intra-communicators into an inter-com-26municator; the function MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS constructs an inter-27communicator from two previously defined disjoint groups; the function 28MPI\_INTERCOMM\_MERGE creates an intra-communicator by merging the local and remote 29groups of an inter-communicator. The functions MPI\_COMM\_DUP and MPI\_COMM\_FREE, 30 introduced previously, duplicate and free an inter-communicator, respectively.  $^{31}$ Overlap of local and remote groups that are bound into an inter-communicator is 32 prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If 33 a process is multithreaded, and MPI calls block only a thread, rather than a process, then 34"dual membership" can be supported. It is then the user's responsibility to make sure that 35 calls on behalf of the two "roles" of a process are executed by two independent threads.) 36 The function MPI\_INTERCOMM\_CREATE can be used to create an inter-communicator 37 from two existing intra-communicators, in the following situation: At least one selected 38 member from each group (the "group leader") has the ability to communicate with the 39 selected member from the other group; that is, a "peer" communicator exists to which both 40leaders belong, and each leader knows the rank of the other leader in this peer communicator.  $^{41}$ Furthermore, members of each group know the rank of their leader. 42Construction of an inter-communicator from two intra-communicators requires separate 43collective operations in the local group and in the remote group, as well as a point-to-point 44communication between a process in the local group and a process in the remote group. 45When using the World Model, the MPI\_COMM\_WORLD communicator (or preferably a 46dedicated duplicate thereof) can be this peer communicator. For applications that have used 47the Sessions Model, spawn, or join it may be necessary to first create an intra-communicator 48to be used as peer.

The application topology functions described in Chapter 8 do not apply to intercommunicators. Users that require this capability should utilize MPI\_INTERCOMM\_MERGE to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topologyoriented intra-communicator. Alternatively, it may be reasonable to devise one's own application topology mechanisms for this case, without loss of generality.

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

IN	local_comm	local intra-communicator (handle)	11
IN	local_leader	rank of local group leader in local_comm (integer)	12
IN	peer_comm	"peer" communicator; significant only at the	13
	peer_comm	local_leader (handle)	14
			15
IN	remote_leader	rank of remote group leader in peer_comm;	16
		significant only at the local_leader (integer)	17 18
IN	tag	tag (integer)	18
OUT	newintercomm	new inter-communicator (handle)	19 20
001			20

## C binding

## 

INTEGER, INTENT(IN) :: local\_leader, remote\_leader, tag
TYPE(MPI\_Comm), INTENT(OUT) :: newintercomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

## Fortran binding

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

This call creates an inter-communicator. It is collective over the union of the local and remote groups. MPI processes should provide identical local\_comm and local\_leader arguments within each group. Wildcards are not permitted for remote\_leader, local\_leader, and tag.

 $^{1}_{2}$ 

```
1
     MPI_INTERCOMM_CREATE_FROM_GROUPS(local_group, local_leader, remote_group,
\mathbf{2}
                     remote_leader, stringtag, info, errhandler, newintercomm)
3
       IN
                 local_group
                                             local group (handle)
4
       IN
                 local_leader
                                             rank of local group leader in local_group (integer)
5
6
       IN
                 remote_group
                                             remote group, significant only at local_leader (handle)
7
                 remote_leader
                                             rank of remote group leader in remote_group,
       IN
8
                                             significant only at local_leader (integer)
9
                 stringtag
                                             unique idenitifier for this operation (string)
       IN
10
11
                 info
       IN
                                             info object (handle)
12
       IN
                 errhandler
                                             error handler to be attached to new
13
                                             inter-communicator (handle)
14
       OUT
                 newintercomm
                                             new inter-communicator (handle)
15
16
17
     C binding
     int MPI_Intercomm_create_from_groups(MPI_Group local_group,
18
                     int local_leader, MPI_Group remote_group, int remote_leader,
19
                    const char *stringtag, MPI_Info info,
20
                    MPI_Errhandler errhandler, MPI_Comm *newintercomm)
21
22
     Fortran 2008 binding
23
     MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
24
                    remote_leader, stringtag, info, errhandler, newintercomm,
25
                     ierror)
26
          TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group
27
          INTEGER, INTENT(IN) :: local_leader, remote_leader
28
          CHARACTER(LEN=*), INTENT(IN) :: stringtag
29
          TYPE(MPI_Info), INTENT(IN) :: info
30
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
31
          TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     Fortran binding
     MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP,
35
                    REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM,
36
37
                     IERROR)
          INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, INFO,
38
                     ERRHANDLER, NEWINTERCOMM, IERROR
39
40
          CHARACTER*(*) STRINGTAG
41
     This call creates an inter-communicator. Unlike MPI_INTERCOMM_CREATE, this function
42
     uses as input previously defined, disjoint local and remote groups. The calling MPI process
43
     must be a member of the local group. The call is collective over the union of the local
44
     and remote groups. All involved MPI processes shall provide an identical value for the
45
     stringtag argument. Within each group, all MPI processes shall provide identical
46
     local_group, local_leader arguments. Wildcards are not permitted for the
47
     remote_leader or local_leader arguments. The stringtag argument serves the same purpose
48
```

as the stringtag used in the MPI\_COMM\_CREATE\_FROM\_GROUP function; it differentiates concurrent calls in a multithreaded environment. The stringtag shall not exceed MPI\_MAX\_FROM\_GROUP\_STRINGTAG characters in length. For C, this includes space for a null terminating character. In the event that MPI\_GROUP\_EMPTY is supplied as the local\_group or remote\_group or both, then the call is a local operation and MPI\_COMM\_NULL is returned as the newintercomm.

MPI\_INTERCOMM\_MERGE(intercomm, high, newintracomm)

IN	intercomm	inter-communicator (handle)
IN	high	ordering of the local and remote groups in the new intra-communicator (logical)
OUT	newintracomm	new intra-communicator (handle)

## C binding

int	MPI_Intercomm_merge(MPI_Comm	intercomm,	int	high,
	MPI_Comm *newintrac	omm)		

## Fortran 2008 binding

<pre>MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)</pre>
TYPE(MPI_Comm), INTENT(IN) :: intercomm
LOGICAL, INTENT(IN) :: high
TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

## Fortran binding

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 inter-communicator 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.*) Is PR400

incomplete?

8

9

27

28

29 30

31

32

33

34

35

36

37

38

39

40 41

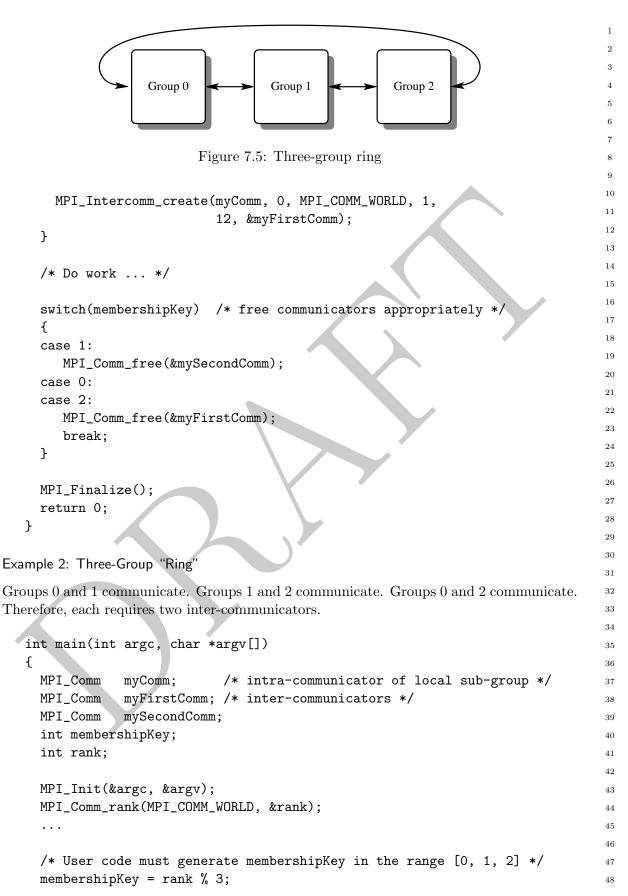
42

43

44

45

```
1
2
3
                      Group 0
                                           Group 1
                                                                Group 2
5
6
7
                               Figure 7.4: Three-group pipeline
8
9
            Inter-Communication Examples
     7.6.3
10
11
     Example 1: Three-Group "Pipeline"
12
     Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires
13
     one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1
14
     inter-communicator.
15
16
        int main(int argc, char *argv[])
17
        {
18
                                      /* intra-communicator of local sub-group */
          MPI_Comm
                       myComm;
19
          MPI_Comm
                       myFirstComm;
                                      /* inter-communicator */
20
          MPI_Comm
                       mySecondComm; /* second inter-communicator (group 1 only) */
21
           int membershipKey;
22
           int rank;
23
24
           MPI_Init(&argc, &argv);
25
           MPI_Comm_rank(MPI_COMM_WORLD, &rank);
26
27
           /* User code must generate membershipKey in the range [0, 1, 2] */
28
           membershipKey = rank % 3;
29
30
           /* Build intra-communicator for local sub-group */
31
           MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
32
33
           /* Build inter-communicators.
                                             Tags are hard-coded. */
34
           if (membershipKey == 0)
35
                                   /* Group 0 communicates with group 1. */
           {
36
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
37
                                    1, &myFirstComm);
38
           }
39
           else if (membershipKey == 1)
40
                           /* Group 1 communicates with groups 0 and 2. */
           Ł
41
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
42
                                    1, &myFirstComm);
43
             MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
44
                                    12, &mySecondComm);
45
           }
46
           else if (membershipKey == 2)
47
           {
                                   /* Group 2 communicates with group 1. */
48
```



```
1
2
          /* Build intra-communicator for local sub-group */
3
          MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
4
5
          /* Build inter-communicators. Tags are hard-coded. */
6
          if (membershipKey == 0)
7
          {
                         /* Group 0 communicates with groups 1 and 2. */
8
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
9
                                   1, &myFirstComm);
10
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
11
                                   2, &mySecondComm);
12
          }
13
          else if (membershipKey == 1)
14
                                                                     */
          {
                     /* Group 1 communicates with groups 0 and 2.
15
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
16
                                   1, &myFirstComm);
17
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
18
                                   12, &mySecondComm);
19
          }
20
          else if (membershipKey == 2)
21
                    /* Group 2 communicates with groups 0 and 1. */
          {
22
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
23
                                   2, &myFirstComm);
24
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
25
                                   12, &mySecondComm);
26
          }
27
28
          /* Do some work ...
29
          /* Then free communicators before terminating... */
30
31
          MPI_Comm_free(&myFirstComm);
32
          MPI_Comm_free(&mySecondComm);
33
          MPI_Comm_free(&myComm);
34
          MPI_Finalize();
35
          return 0;
36
        }
37
```

# 7.7 Caching

38

39 40

41

42

43

44 45

46

47

48

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects, communicators, windows, and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window, or datatype,
- quickly retrieve that information, and

## Unofficial Draft for Comment Only

• 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.*)

## 7.7.1 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, or datatype. Accessor functions include the following:

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

 $46 \\ 47$ 

1 • obtain a key value (used to identify an attribute); the user specifies "callback" func- $\mathbf{2}$ tions by which MPI informs the application when the communicator is destroyed or 3 copied. 4 • store and retrieve the value of an attribute; 56 Advice to implementors. Caching and callback functions are only called synchronously, 7 in response to explicit application requests. This avoids problems that result from re-8 peated crossings between user and system space. (This synchronous calling rule is a 9 general property of MPI.) 10 The choice of key values is under control of MPI. This allows MPI to optimize its 11 implementation of attribute sets. It also avoids conflict between independent modules 12caching information on the same communicators. 13 14A much smaller interface, consisting of just a callback facility, would allow the entire 15caching facility to be implemented by portable code. However, with the minimal call-16back interface, some form of table searching is implied by the need to handle arbitrary 17 communicators. In contrast, the more complete interface defined here permits rapid 18 access to attributes through the use of pointers in communicators (to find the attribute 19 table) and cleverly chosen key values (to retrieve individual attributes). In light of the 20efficiency "hit" inherent in the minimal interface, the more complete interface defined 21here is seen to be superior. (End of advice to implementors.) 22 23MPI provides the following services related to caching. They are all process local.  $^{24}$ 257.7.2 Communicators 26Functions for caching on communicators are: 272829MPI\_COMM\_CREATE\_KEYVAL(comm\_copy\_attr\_fn, comm\_delete\_attr\_fn, comm\_keyval, 30 extra\_state)  $^{31}$ IN comm\_copy\_attr\_fn copy callback function for comm\_keyval (function) 32 33 IN comm\_delete\_attr\_fn delete callback function for comm\_keyval (function) 34 OUT comm\_keyval key value for future access (integer) 35 IN extra\_state extra state for callback function 36 37 38 C binding 39 int MPI\_Comm\_create\_keyval(MPI\_Comm\_copy\_attr\_function \*comm\_copy\_attr\_fn, 40 MPI\_Comm\_delete\_attr\_function \*comm\_delete\_attr\_fn, 41 int \*comm\_keyval, void \*extra\_state) 42Fortran 2008 binding 43 MPI\_Comm\_create\_keyval(comm\_copy\_attr\_fn, comm\_delete\_attr\_fn, comm\_keyval, 44 extra\_state, ierror) 45PROCEDURE (MPI\_Comm\_copy\_attr\_function), INTENT(IN) :: comm\_copy\_attr\_fn 46 PROCEDURE(MPI\_Comm\_delete\_attr\_function), INTENT(IN) :: 47comm\_delete\_attr\_fn 48

Unofficial Draft for Comment Only

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

and

1	SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
2	EXTRA_STATE, IERROR)
3	INTEGER COMM, COMM_KEYVAL, IERROR
4	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
5	
6	The comm_copy_attr_fn function is invoked when a communicator is duplicated by
7	MPI_COMM_DUP or MPI_COMM_IDUP. comm_copy_attr_fn should be of type
8	MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in
9	oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its
10	corresponding attribute. If it returns $flag = 0$ or .FALSE., then the attribute is deleted in
11	the duplicated communicator. Otherwise ( $flag = 1$ or .TRUE.), the new attribute value is
12	set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success
13	and an error code on failure (in which case MPI_COMM_DUP or MPI_COMM_IDUP will
14	fail).
15	The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN
16	or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a
17	function that does nothing other than returning $flag = 0$ or .FALSE. (depending on whether
18	the keyval was created with a C or Fortran binding to MPI_COMM_CREATE_KEYVAL) and
19	MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1 or
20	.TRUE., returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.
21	These replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN,
22	whose use is deprecated.
23	
24	Advice to users. Even though both formal arguments attribute_val_in and
25	attribute_val_out are of type void*, their usage differs. The C copy function is passed
26	by MPI in attribute_val_in the <i>value</i> of the attribute, and in attribute_val_out the <i>address</i> of the attribute, so as to allow the function to return the (new) attribute
27	value. The use of type void* for both is to avoid messy type casts.
28	
29	A valid copy function is one that completely duplicates the information by making
30	a full duplicate copy of the data structures implied by an attribute; another might
31	just make another reference to that data structure, while using a reference-count
32	mechanism. Other types of attributes might not copy at all (they might be specific
33	to oldcomm only). (End of advice to users.)
34	
35	Advice to implementors. A C interface should be assumed for copy and delete
36	functions associated with key values created in C; a Fortran calling interface should
37	be assumed for key values created in Fortran. (End of advice to implementors.)
38	Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows.
39	The comm_delete_attr_fn function is invoked when a communicator is deleted by
40	MPI_COMM_FREE or when a call is made explicitly to MPI_COMM_DELETE_ATTR.
41	comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.
42	This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and
43	MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function
44	returns MPI_SUCCESS on success and an error code on failure (in which case
45	MPI_COMM_FREE will fail).
46	The argument comm_delete_attr_fn may be specified as
47	MPI_COMM_NULL_DELETE_FN from either C or Fortran.
48	

MPI\_COMM\_NULL\_DELETE\_FN is a function that does nothing, other than returning 1 MPI\_SUCCESS. MPI\_COMM\_NULL\_DELETE\_FN replaces MPI\_NULL\_DELETE\_FN, whose 2 3 use is deprecated. 4 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), 56 is erroneous. 7 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 8 9 values. 10 The predefined Fortran functions Advice to implementors. 11 MPI\_COMM\_NULL\_COPY\_FN, MPI\_COMM\_DUP\_FN, and 12MPI\_COMM\_NULL\_DELETE\_FN are defined in the mpi module (and mpif.h) and 13the mpi\_f08 module with the same name, but with different interfaces. Each function 14can coexist twice with the same name in the same MPI library, one routine as an 15implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other 16routine within mpi\_f08 declared with CONTAINS. These routines have different link 17names, which are also different to the link names used for the routines used in C. 18 (End of advice to implementors.) 1920Callbacks, including the predefined Fortran functions Advice to users. 21MPI\_COMM\_NULL\_COPY\_FN, MPI\_COMM\_DUP\_FN, and 22 MPI\_COMM\_NULL\_DELETE\_FN should not be passed from one application routine 23that uses the mpi\_f08 module to another application routine that uses the mpi module  $^{24}$ or mpif.h, and vice versa; see also the advice to users on page 836. (End of advice to 25users.) 262728 29 MPI\_COMM\_FREE\_KEYVAL(comm\_keyval) 30 comm\_keyval INOUT key value (integer) 31 32 C binding 33 int MPI\_Comm\_free\_keyval(int \*comm\_keyval) 34 35Fortran 2008 binding 36 MPI\_Comm\_free\_keyval(comm\_keyval, ierror) 37 INTEGER, INTENT(INOUT) :: comm\_keyval 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 Fortran binding 40 41 MPI\_COMM\_FREE\_KEYVAL(COMM\_KEYVAL, IERROR) 42INTEGER COMM\_KEYVAL, IERROR 43 Frees an extant attribute key. This function sets the value of keyval to 44MPI\_KEYVAL\_INVALID. Note that it is not erroneous to free an attribute key that is in use, 45because the actual free does not transpire until after all references (in other communicators 46on the process) to the key have been freed. These references need to be explicitly freed by the

program, either via calls to MPI\_COMM\_DELETE\_ATTR that free one attribute instance,

47

```
1
     or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed
\mathbf{2}
     communicator.
3
4
     MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)
5
6
       INOUT
                 comm
                                              communicator to which attribute will be attached
7
                                              (handle)
8
       IN
                 comm_keyval
                                              key value (integer)
9
       IN
                 attribute_val
                                              attribute value
10
11
     C binding
12
     int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
13
14
     Fortran 2008 binding
15
     MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
16
          TYPE(MPI_Comm), INTENT(IN) :: comm
17
          INTEGER, INTENT(IN) :: comm_keyval
18
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
19
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     Fortran binding
21
     MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
22
23
          INTEGER COMM, COMM_KEYVAL, IERROR
^{24}
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
25
          This function stores the stipulated attribute value attribute_val for subsequent retrieval
26
     by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if
27
     MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback
28
     function comm_delete_attr_fn was executed), and a new value was next stored. The call
29
     is erroneous if there is no key with value keyval; in particular MPI_KEYVAL_INVALID is an
30
     erroneous key value. The call will fail if the comm_delete_attr_fn function returned an error
^{31}
     code other than MPI_SUCCESS.
32
33
34
     MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)
35
       IN
                 comm
                                              communicator to which the attribute is attached
36
                                              (handle)
37
       IN
                                              key value (integer)
38
                 comm_keyval
39
       OUT
                 attribute_val
                                              attribute value, unless flag = false
40
       OUT
                 flag
                                              false if no attribute is associated with the key
41
                                              (logical)
42
43
     C binding
44
     int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
45
                     int *flag)
46
47
     Fortran 2008 binding
48
```

<pre>MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)</pre>
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: comm_keyval
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
INTEGER COMM, COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
LOGICAL FLAG
Retrieves attribute value by key. The call is erroneous if there is no key v

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

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

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

MPI_COMM_DELETE_ATTR(comm, comm_keyval)		
INOUT comm	communicator from which the attribute is deleted	32
	(handle)	33
IN comm_keyval	key value (integer)	34
		35
C binding		36
int MPI_Comm_delete_attr(MPI_Comm	comm int comm kouural)	37
Int MF1_Comm_delete_atti(MF1_Comm	comm, int comm_keyval)	38
Fortran 2008 binding		
MPI_Comm_delete_attr(comm, comm_keyval, ierror)		
TYPE(MPI_Comm), INTENT(IN) ::	comm	41
INTEGER, INTENT(IN) :: comm_ke	yval	42
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	43
Fortron binding		
Fortran binding MPI_COMM_DELETE_ATTR(COMM, COMM_KE		45
INTEGER COMM, COMM_KEYVAL, IER	•	46
INTEGER COMM, COMM_RETVAL, IEN		47

```
1
         Delete attribute from cache by key. This function invokes the attribute delete function
\mathbf{2}
     comm_delete_attr_fn specified when the keyval was created. The call will fail if the
3
     comm_delete_attr_fn function returns an error code other than MPI_SUCCESS.
4
         Whenever a communicator is replicated using the function MPI_COMM_DUP or
     MPI_COMM_IDUP, all call-back copy functions for attributes that are currently set are
\mathbf{5}
6
     invoked (in arbitrary order). Whenever a communicator is deleted using the function
7
     MPI_COMM_FREE all callback delete functions for attributes that are currently set are
8
     invoked.
9
10
     7.7.3 Windows
11
     The functions for caching on windows are:
12
13
14
     MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
15
                    extra_state)
16
       IN
                win_copy_attr_fn
                                             copy callback function for win_keyval (function)
17
18
                win_delete_attr_fn
                                             delete callback function for win_keyval (function)
       IN
19
       OUT
                win_keyval
                                             key value for future access (integer)
20
       IN
                 extra_state
                                             extra state for callback function
21
22
23
     C binding
     int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
^{24}
25
                    MPI_Win_delete_attr_function *win_delete_attr_fn,
26
                    int *win_keyval, void *extra_state)
27
     Fortran 2008 binding
28
     MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
29
                    extra_state, ierror)
30
         PROCEDURE(MPI_Win_copy_attr_function), INTENT(IN) :: win_copy_attr_fn
31
         PROCEDURE(MPI_Win_delete_attr_function), INTENT(IN) ::
32
                     win_delete_attr_fn
33
          INTEGER, INTENT(OUT) :: win_keyval
34
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     Fortran binding
38
     MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
39
                    EXTRA_STATE, IERROR)
40
         EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
41
          INTEGER WIN_KEYVAL, IERROR
42
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
43
         The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or
44
     MPI_WIN_DUP_FN from either C or Fortran. MPI_WIN_NULL_COPY_FN is a function
45
     that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_WIN_DUP_FN is
46
     a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in
47
```

```
<sub>48</sub> attribute_val_out, and returns MPI_SUCCESS.
```

```
The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN
                                                                                      1
                                                                                      \mathbf{2}
from either C or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing,
other than returning MPI_SUCCESS.
The C callback functions are:
                                                                                      4
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
                                                                                      5
                                                                                      6
              void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
                                                                                      7
and
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
                                                                                      10
              void *attribute_val, void *extra_state);
                                                                                      11
With the mpi_f08 module, the Fortran callback functions are:
                                                                                      12
                                                                                      13
ABSTRACT INTERFACE
                                                                                      14
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
                                                                                      15
               attribute_val_in, attribute_val_out, flag, ierror)
                                                                                      16
    TYPE(MPI_Win) :: oldwin
                                                                                      17
    INTEGER :: win_keyval, ierror
                                                                                      18
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                      19
               attribute_val_out
                                                                                      20
    LOGICAL :: flag
                                                                                      21
and
                                                                                      22
ABSTRACT INTERFACE
                                                                                      23
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
                                                                                      24
               extra_state, ierror)
                                                                                      25
    TYPE(MPI_Win) :: win
                                                                                      26
    INTEGER :: win_keyval, ierror
                                                                                      27
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                      28
                                                                                      29
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                      30
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
                                                                                      31
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                      32
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
                                                                                      33
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                      34
               ATTRIBUTE_VAL_OUT
                                                                                      35
    LOGICAL FLAG
                                                                                      36
and
                                                                                      37
SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
                                                                                      38
              EXTRA_STATE, IERROR)
                                                                                      39
    INTEGER WIN, WIN_KEYVAL, IERROR
                                                                                      40
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                      41
                                                                                      42
    If an attribute copy function or attribute delete function returns other than
                                                                                      43
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is
                                                                                      44
erroneous.
                                                                                      45
                                                                                      46
                                                                                      47
```

```
1
     MPI_WIN_FREE_KEYVAL(win_keyval)
\mathbf{2}
       INOUT
                 win_keyval
                                              key value (integer)
3
4
     C binding
5
     int MPI_Win_free_keyval(int *win_keyval)
6
\overline{7}
     Fortran 2008 binding
8
     MPI_Win_free_keyval(win_keyval, ierror)
9
          INTEGER, INTENT(INOUT) :: win_keyval
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     Fortran binding
12
     MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
13
          INTEGER WIN_KEYVAL, IERROR
14
15
16
     MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)
17
18
                                              window to which attribute will be attached (handle)
       INOUT
                 win
19
       IN
                 win_keyval
                                              key value (integer)
20
21
       IN
                 attribute_val
                                              attribute value
22
23
     C binding
^{24}
     int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
25
     Fortran 2008 binding
26
     MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
27
          TYPE(MPI_Win), INTENT(IN) :: win
28
          INTEGER, INTENT(IN) :: win_keyval
29
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     Fortran binding
33
     MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
34
          INTEGER WIN, WIN_KEYVAL, IERROR
35
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
36
37
38
     MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)
39
       IN
                 win
                                              window to which the attribute is attached (handle)
40
41
                 win_keyval
       IN
                                              key value (integer)
42
       OUT
                 attribute_val
                                              attribute value, unless flag = false
43
       OUT
                 flag
                                              false if no attribute is associated with the key
44
                                              (logical)
45
46
47
     C binding
48
```

## 7.7. CACHING

<pre>int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,</pre>	1 2
Fortran 2008 binding	3
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)	4
TYPE(MPI_Win), INTENT(IN) :: win	5
INTEGER, INTENT(IN) :: win_keyval	6
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val	7
LOGICAL, INTENT(OUT) :: flag	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
Fortran binding	11
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	12
INTEGER WIN, WIN_KEYVAL, IERROR	13
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	14
LOGICAL FLAG	15
	16
	17
MPI_WIN_DELETE_ATTR(win, win_keyval)	18
	19 1-)
	, -
IN win_keyval key value (integer)	21
	22
C binding	23
int MPI_Win_delete_attr(MPI_Win win, int win_keyval)	24 25
Fortran 2008 binding	25
MPI_Win_delete_attr(win, win_keyval, ierror)	27
TYPE(MPI_Win), INTENT(IN) :: win	28
INTEGER, INTENT(IN) :: win_keyval	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
Fortran binding	31
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)	32
INTEGER WIN, WIN_KEYVAL, IERROR	33
	34
	35
7.7.4 Datatypes	36
The new functions for caching on datatypes are:	37
	38
	39
*	40
	41 42
	42
	43
	45
	46
	47

MPI_TYF	PE_CREATE_KEYVAL(type_ extra_state)	_copy_attr_fn, type_delete_attr_fn, type_keyval,
IN	type_copy_attr_fn	copy callback function for <code>type_keyval</code> (function)
IN	type_delete_attr_fn	delete callback function for type_keyval (function)
OUT	type_keyval	key value for future access (integer)
IN	extra_state	extra state for callback function
C bindir int MPI_	Type_create_keyval(MPI MPI_Type_delete_a	_Type_copy_attr_function *type_copy_attr_fn attr_function *type_delete_attr_fn, , void *extra_state)
	2008 binding	
MPI_Type	<pre>_create_keyval(type_co extra_state, ierr</pre>	<pre>py_attr_fn, type_delete_attr_fn, type_keyval</pre>
PROC INTE INTE	EDURE(MPI_Type_delete_ type_delete_attr GER, INTENT(OUT) :: ty	pe_keyval IND), INTENT(IN) :: extra_state
Fortran	binding	
	0	PY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAI
	EXTRA_STATE, IERF	
EXTE	RNAL TYPE_COPY_ATTR_FN	
	GER TYPE_KEYVAL, IERRO	
INTE	GER(KIND=MPI_ADDRESS_K	IND) EXTRA_STATE
MPI_TYF that does is a simpl	<b>E_DUP_FN</b> from either C nothing other than return	in may be specified as MPI_TYPE_NULL_COPY_FN or Fortran. MPI_TYPE_NULL_COPY_FN is a functi- ning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_F at sets flag = 1, returns the value of attribute_val_in SUCCESS.
The from either	argument type_delete_attr_	fn may be specified as MPI_TYPE_NULL_DELETE_I E_NULL_DELETE_FN is a function that does nothin
	llback functions are:	
typedef	<pre>int MPI_Type_copy_attr</pre>	_function(MPI_Datatype oldtype,
		<pre>void *extra_state, void *attribute_val_in, val_out, int *flag);</pre>
and		
		<pre>tr_function(MPI_Datatype datatype, void *attribute_val, void *extra_state);</pre>
	<pre>mpi_f08 module, the Fort INTERFACE</pre>	ran callback functions are:

```
1
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
                                                                                     2
               attribute_val_in, attribute_val_out, flag, ierror)
                                                                                     3
    TYPE(MPI_Datatype) :: oldtype
    INTEGER :: type_keyval, ierror
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                     5
               attribute_val_out
                                                                                     6
    LOGICAL :: flag
and
                                                                                     9
ABSTRACT INTERFACE
                                                                                     10
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
                                                                                     11
               attribute_val, extra_state, ierror)
                                                                                     12
    TYPE(MPI_Datatype) :: datatype
                                                                                     13
    INTEGER :: type_keyval, ierror
                                                                                     14
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                     15
                                                                                     16
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                     17
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
                                                                                     18
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                     19
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                     20
                                                                                     21
               ATTRIBUTE_VAL_OUT
                                                                                     22
    LOGICAL FLAG
                                                                                     23
and
                                                                                     24
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
                                                                                     25
              EXTRA_STATE, IERROR)
                                                                                     26
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
                                                                                     27
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                     28
                                                                                     29
    If an attribute copy function or attribute delete function returns other than
                                                                                     30
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),
                                                                                     31
is erroneous.
                                                                                     32
                                                                                     33
MPI_TYPE_FREE_KEYVAL(type_keyval)
                                                                                     34
                                                                                     35
 INOUT type_keyval
                                     key value (integer)
                                                                                     36
                                                                                     37
C binding
                                                                                     38
int MPI_Type_free_keyval(int *type_keyval)
                                                                                     39
Fortran 2008 binding
                                                                                     40
                                                                                     41
MPI_Type_free_keyval(type_keyval, ierror)
    INTEGER, INTENT(INOUT) :: type_keyval
                                                                                     42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     43
                                                                                     44
Fortran binding
                                                                                     45
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
                                                                                     46
    INTEGER TYPE_KEYVAL, IERROR
                                                                                     47
                                                                                     48
```

```
1
     MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)
2
       INOUT
                datatype
                                            datatype to which attribute will be attached (handle)
3
                type_keyval
       IN
                                            key value (integer)
4
5
                 attribute_val
       IN
                                            attribute value
6
7
     C binding
8
     int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
9
                    void *attribute_val)
10
     Fortran 2008 binding
11
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
12
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
         INTEGER, INTENT(IN) :: type_keyval
14
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     Fortran binding
18
     MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
19
         INTEGER DATATYPE, TYPE_KEYVAL, IERROR
20
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
21
22
23
     MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)
24
                                            datatype to which the attribute is attached (handle)
       IN
                 datatype
25
26
       IN
                 type_keyval
                                            key value (integer)
27
       OUT
                 attribute_val
                                            attribute value, unless flag = false
28
       OUT
                flag
                                            false if no attribute is associated with the key
29
                                            (logical)
30
31
32
     C binding
33
     int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
34
                    void *attribute_val, int *flag)
35
     Fortran 2008 binding
36
     MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         INTEGER, INTENT(IN) :: type_keyval
39
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
40
         LOGICAL, INTENT(OUT) :: flag
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     Fortran binding
44
     MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
45
         INTEGER DATATYPE, TYPE_KEYVAL, IERROR
46
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
47
         LOGICAL FLAG
48
```

INOUT		tatype, type_keyval)
	datatype	datatype from which the attribute is deleted (handle)
IN	type_keyval	key value (integer)
C bindin		
.nt MPI_7	Type_delete_attr(M	PI_Datatype datatype, int type_keyval)
Fortran 2	2008 binding	
	0	ype, type_keyval, ierror)
• -		TENT(IN) :: datatype
INTEC	GER, INTENT(IN) ::	type_keyval
INTEC	ER, OPTIONAL, INT	ENT(OUT) :: ierror
Fortran h	ninding	
		YPE, TYPE_KEYVAL, IERROR)
	GER DATATYPE, TYPE	
	,	
7.7.5 Eri	ror Class for Invalid k	(eyval
Key values	s for attributes are sy	vstem-allocated, by
-		. Only such values can be passed to the functions that use
ey values	as input arguments.	. In order to signal that an erroneous key value has been
		s, there is a new $MPI$ error class: $MPI\_ERR\_KEYVAL.$ It can
	-	JT, MPI_ATTR_GET, MPI_ATTR_DELETE,
	VAL_FREE,	
	<pre>K}_DELETE_ATTR,</pre>	
	<pre>K}_SET_ATTR, </pre>	
<u>,</u>	<pre>K}_GET_ATTR, </pre>	MPI_COMM_DUP, MPI_COMM_IDUP,
C C	· · · · · · · · · · · · · · · · · · ·	nd MPI_COMM_FREE. The last four are included because
		py and delete functions for attributes.
	ir argument to the co	by and delete functions for attributes.
76 Δ+	tributes Example	
.1.0 At		
		example shows how to write a collective communication
_		ng to be more efficient after the first call. (End of advice to
user	s.)	
/* ket	for this module'	s stuff: */
-		
202020	; int gop kev = MP	
	c int gop_key = MP	
typede	c int gop_key = MP ef struct	
typede {		
{		/* reference count */
{ int	ef struct t ref_count;	
{ int /*	ef struct t ref_count;	/* reference count */

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

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

# 7.8 Naming Objects

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

			40
MPI_COM	M_SET_NAME(comm, comm_	_name)	41
INOUT	comm	communicator whose identifier is to be set (handle)	42 43
IN	comm_name	the character string which is remembered as the	44
		name (string)	45
			46
C binding	5		47
int MPI_C	comm_set_name(MPI_Comm cor	nm, const char *comm_name)	48

 $\mathbf{2}$ 

10 11

12

13 14

15

16

17 18

19

20

21

22

23 24

> 25 26

27

28 29

30 31 32

33 34

35

36

37

38

1		008 binding				
2	<pre>MPI_Comm_set_name(comm, comm_name, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm</pre>					
3						
4		CTER(LEN=*), INTENT(IN) :				
5 6	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror			
7	Fortran b	inding				
8	MPI_COMM_	SET_NAME(COMM, COMM_NAME,	IERROR)			
9	INTEG	ER COMM, IERROR				
10	CHARA	CTER*(*) COMM_NAME				
11	MPI C	OMM SET NAME allows a us	ser to associate a name string with a communicator.			
12			MPI_COMM_SET_NAME will be saved inside the			
13	MPI library	y (so it can be freed by the ca	ller immediately after the call, or allocated on the			
14	stack). Lea	ding spaces in name are signif	ficant but trailing ones are not.			
15	MPI_C	OMM_SET_NAME is a local	(non-collective) operation, which only affects the			
16	name of the	e communicator as seen in the	process which made the MPI_COMM_SET_NAME			
17	call. There	is no requirement that the sa	me (or any) name be assigned to a communicator			
18	in every pre-	ocess where it exists.				
19	1 days	a to warma Since MPL COM	M_SET_NAME is provided to help debug code, it			
20			to a communicator in all of the processes where it			
21		, to avoid confusion. (End of	-			
22 23	CAIDID					
23		-	be stored is limited to the value of			
25			MPI_MAX_OBJECT_NAME-1 in C to allow for the			
26			s longer than this will result in truncation of the			
27	name. MPL	_MAX_OBJECT_NAME must ha	ave a value of at least 64.			
28	Advia	e to users. Under circumstar	nces of store exhaustion an attempt to put a name			
29			the value of MPI_MAX_OBJECT_NAME should be			
30	viewe	d only as a strict upper bound	d on the name length, not a guarantee that setting			
31	name	s of less than this length will	always succeed. (End of advice to users.)			
32	Advic	e to implementors Implement	ntations which pre-allocate a fixed size space for a			
33		-	llocation as the value of MPI_MAX_OBJECT_NAME.			
34		Ū.	ace for the name from the heap should still define			
35 36			elatively small value, since the user has to allocate			
37			e when calling MPI_COMM_GET_NAME. (End of			
38		e to implementors.)				
39						
40						
41	MPI_COM	M_GET_NAME(comm, comm_	name, resultlen)			
42	IN	comm	communicator whose name is to be returned (handle)			
43 44	OUT	comm_name	the name previously stored on the communicator, or			
44 45			an empty string if no such name exists (string)			
46	OUT	resultlen	length of returned name (integer)			
47			0			
48	C binding					

int MPI\_Comm\_get\_name(MPI\_Comm comm, char \*comm\_name, int \*resultlen)

#### Fortran 2008 binding

```
MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
   TYPE(MPI_Comm), INTENT(IN) :: comm
   CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
   INTEGER, INTENT(OUT) :: resultlen
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### Fortran binding

```
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
INTEGER COMM, RESULTLEN, IERROR
CHARACTER*(*) COMM_NAME
```

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

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

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

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

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

(End of rationale.)

1

2

5

6

10

11

12 13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31 32

33

34

35 36

37

38

39

40

41

42

43 44

 $45 \\ 46$ 

```
1
           Advice to users. The above definition means that it is safe simply to print the string
\mathbf{2}
           returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was
3
           no name.
4
           Note that associating a name with a communicator has no effect on the semantics of
5
           an MPI program, and will (necessarily) increase the store requirement of the program,
6
           since the names must be saved. Therefore there is no requirement that users use these
7
           functions to associate names with communicators. However debugging and profiling
8
           MPI applications may be made easier if names are associated with communicators,
9
           since the debugger or profiler should then be able to present information in a less
10
           cryptic manner. (End of advice to users.)
11
12
          The following functions are used for setting and getting names of datatypes. The
13
     constant MPI_MAX_OBJECT_NAME also applies to these names.
14
15
16
     MPI_TYPE_SET_NAME(datatype, type_name)
17
       INOUT
                 datatype
                                              datatype whose identifier is to be set (handle)
18
       IN
                                              the character string which is remembered as the
                 type_name
19
                                              name (string)
20
21
     C binding
22
     int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)
23
^{24}
     Fortran 2008 binding
25
     MPI_Type_set_name(datatype, type_name, ierror)
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          CHARACTER(LEN=*), INTENT(IN) :: type_name
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     Fortran binding
30
     MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
^{31}
          INTEGER DATATYPE, IERROR
32
          CHARACTER*(*) TYPE_NAME
33
34
35
36
     MPI_TYPE_GET_NAME(datatype, type_name, resultlen)
37
       IN
                 datatype
                                              datatype whose name is to be returned (handle)
38
       OUT
                 type_name
                                              the name previously stored on the datatype, or an
39
                                              empty string if no such name exists (string)
40
41
       OUT
                 resultlen
                                              length of returned name (integer)
42
43
     C binding
44
     int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,
45
                     int *resultlen)
46
47
     Fortran 2008 binding
     MPI_Type_get_name(datatype, type_name, resultlen, ierror)
48
```

CHARA INTEG	TYPE(MPI_Datatype), INTENT(IN) :: datatype CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
MPI_TYPE_ INTEG	Fortran binding MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR) INTEGER DATATYPE, RESULTLEN, IERROR CHARACTER*(*) TYPE_NAME			
Named predefined datatypes have the default names of the datatype name. For example, MPI_WCHAR has the default name of "MPI_WCHAR". The following functions are used for setting and getting names of windows. The constant MPI_MAX_OBJECT_NAME also applies to these names.				
MPI_WIN_	SET_NAME(win, win_name)		15 16	
INOUT	win	window whose identifier is to be set (handle)	17	
IN	win_name	the character string which is remembered as the name (string)	18 19 20	
Chindin			21	
C binding	g /in_set_name(MPI_Win win,	const char *win name)	22	
			23 24	
	2008 binding set_name(win, win_name, io	arror)	25	
	MPI_Win), INTENT(IN) :: 1		26	
	CHARACTER(LEN=*), INTENT(IN) :: win_name			
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	28	
Fortran k	oinding		29 30	
	ET_NAME(WIN, WIN_NAME, I	ERROR)	31	
INTEG	ER WIN, IERROR		32	
CHARA	CTER*(*) WIN_NAME		33	
			34	
			35	
MPI_WIN_	.GET_NAME(win, win_name, i	resultlen)	36	
IN	win	window whose name is to be returned (handle)	37 38	
OUT	win_name	the name previously stored on the window, or an empty string if no such name exists (string)	39 40	
OUT	resultlen	length of returned name (integer)	41	
			42	
C binding	5		43	
int MPI_W	<pre>in_get_name(MPI_Win win,</pre>	char *win_name, int *resultlen)	44	
Fortran 2	008 binding		45 46	
	get_name(win, win_name, re	esultlen, ierror)	46 47	
	MPI_Win), INTENT(IN) :: 1		48	

CHARACTER(LEN=MPI\_MAX\_OBJECT\_NAME), INTENT(OUT) :: win\_name INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI\_WIN\_GET\_NAME(WIN, WIN\_NAME, RESULTLEN, IERROR) INTEGER WIN, RESULTLEN, IERROR CHARACTER\*(\*) WIN\_NAME

# 7.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

7.9.1 Basic Statements

17When a caller passes a communicator (that contains a context and group) to a callee, that 18 communicator must be free of side effects throughout execution of the subprogram: there 19should be no active operations on that communicator that might involve the process. This 20provides one model in which libraries can be written, and work "safely." For libraries 21so designated, the callee has permission to do whatever communication it likes with the 22communicator, and under the above guarantee knows that no other communications will 23interfere. Since we permit good implementations to create new communicators without  $^{24}$ synchronization (such as by preallocated contexts on communicators), this does not impose 25a significant overhead.

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

29 30

 $^{31}$ 

1

2

3

4

5

6

7

8 9 10

11 12

13

14 15

16

## 7.9.2 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.

 $^{41}$ 

### <sup>42</sup> Static Communicator Allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are singlethreaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

#### Dynamic Communicator Allocation

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

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

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

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

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

#### The General Case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, communicator creation is properly coordinated. 1

2

3

4

5 6 7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24 25

26

27

28 29 30

31

32

33

34

35



#-... all changes
 #-nnn by issue nnn
 #-PRnnn by pull request
 Non problematic fomatting-changes and typo-corrections are not listed
 Embiggening: only new \_c versions are marked, all other is not listed
 New or changed content
 Problematic fomatting-changes are listed
 #-error
 Errors in RC 40 - also marked with #-error
 B.m.n: i. - refers to Change-Log section B.m.n, Item i

# **Process Topologies**

## 8.1 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 7, 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 [49]. 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 [12, 13].

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 

 $^{31}$ 

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.*)

# 8.2 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-19 tion 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 example, the relation between group rank and coordinates for four processes in a  $(2 \times 2)$  grid is as follows.

# 8.3 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 7.

40Information representing an MPI virtual topology may be added to a communicator at 41 the time of its creation. If a communicator creation function adds information representing 42an MPI virtual topology to the output communicator it creates, then it either propagates 43the topology representation from the input communicator to the output communicator, or 44adds a new topology representation generated from the input parameters that describe a 45virtual topology. The description of every MPI communicator creation function explicitly 46 states how topology information is handled. Communicator creation functions that create 47new topology representations are described in Section 8.5. 48

1

2

3

4 5

> 6 7

> > 8

9

10

11

12

13

14

15

25

26

27

28

29

30

 $^{31}$ 

32

33 34

35 36

37

38

39

#-314

no such changes.	B.1.2.7
8.4 Overview of the Functions	1 2
MPI supports three topology types: <b>Cartesian</b> , <b>graph</b> , and <b>distributed graph</b> . The	3
function MPI_CART_CREATE can be used to create Cartesian topologies, the function	<sub>4</sub> #-314
MPI_GRAPH_CREATE can be used to create graph topologies, and the functions	5
MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE can be used to	6
create distributed graph topologies. These topology creation functions are collective. As	7
with other collective calls, the program must be written to work correctly, whether the call	8
synchronizes or not.	9
The above topology creation functions take as input an existing communicator	10
comm_old, which defines the set of processes on which the topology is to be mapped. For	11
MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical	12
values on all processes of the group of comm_old. When calling MPI_GRAPH_CREATE,	12
each process specifies all nodes and edges in the graph. In contrast, the functions	13
MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE are used to spec-	14
ify 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	16 17
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.	18
In all cases, a new communicator <b>comm_topol</b> is created that carries the topological struc-	19
	20
ture as cached information (see Chapter 7). In analogy to function MPI_COMM_CREATE,	21
no cached information propagates from comm_old to comm_topol.	22
MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimen-	23
sion. For each coordinate direction one specifies whether the process structure is periodic or	24
not. Note that an $n$ -dimensional hypercube is an $n$ -dimensional torus with 2 processes per	25
coordinate direction. Thus, special support for hypercube structures is not necessary. The	26
local auxiliary function MP1_DIMS_CREATE can be used to compute a balanced distribution	27
of processes among a given number of dimensions.	28
MPI defines functions to query a communicator for topology information. The function	29
MPI_TOPO_TEST is used to query for the type of topology associated with a communicator.	30
Depending on the topology type, different information can be extracted. For a graph	31
topology, the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-	32 <b>#-314</b>
topology information that is associated with the communicator. Additionally, the functions	33
MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to obtain	34
the neighbors of an arbitrary node in the graph. For a distributed graph topology, the	35
functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS	36
can be used to obtain the neighbors of the calling process. For a Cartesian topology, the	37
function MPI_CARTDIM_GET returns the number of dimensions and MPI	38 ## orr
MPI_CART_GET returns the numbers of processes in each dimension and periodicity of the	<sub>39</sub> #-erro
associated Cartesian topology. Additionally, the functions MPI_CART_RANK and	<sup>40</sup> Done
MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa.	41 in +
The function MPI_CART_SHIFT provides the information needed to communicate with	42 PR439
neighbors along a Cartesian dimension. All of these query functions are local.	43
For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Carte-	44
sian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input	45
communicator's group.	46
The two additional functions, $MPI\_GRAPH\_MAP$ and $MPI\_CART\_MAP,$ are, in gen-	47
eral, not called by the user directly. However, together with the communicator manipulation	48

Unofficial Draft for Comment Only

1		functions presented in Chapter 7, they are sufficient to implement all other topology func-					
2		tions. Section 8.5.8 outlines such an implementation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER.					
3 4		The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL,					
5	MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with t						
6		nearest neighbors on the topology associated with the communicator. The nonblocking					
7			R_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV,				
8			L, MPI_INEIGHBOR_ALLTOALLV, and				
9		IEIGHBOR_ALLTOAL					
10							
11	ог .						
12	8.5	Topology Constru	ctors				
13	8.5.1	Cartesian Constructo	r				
14	0.5.1						
15							
16		APT CPEATE(comm	old, ndims, dims, periods, reorder, comm_cart)				
17		,					
18 19	IN	comm_old	input communicator (handle)				
19 20	IN	ndims	number of dimensions of Cartesian grid (integer)				
21	IN	dims	integer array of size ndims specifying the number of				
22			processes in each dimension				
23	IN	periods	logical array of size ndims specifying whether the grid				
24			is periodic (true) or not (false) in each dimension				
25	IN	reorder	ranking may be reordered $(true)$ or not $(false)$				
26			(logical)				
27 28	OUT	comm_cart	communicator with new Cartesian topology (handle)				
29							
30	C bind	ding					
31	int MP	int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],					
32	<pre>const int periods[], int reorder, MPI_Comm *comm_cart)</pre>						
33	Fortra	in 2008 binding					
34		-	, ndims, dims, periods, reorder, comm_cart, ierror)				
35			NT(IN) :: comm_old				
36			:: ndims, dims(ndims)				
37			:: periods(ndims), reorder				
38			NT(OUT) :: comm_cart				
39			NTENT(OUT) :: ierror				
40 41	Fortro	n binding					
41 42		•	, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)				
42			IMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR				
43		GICAL PERIODS(*),					
45							
<b>#-277</b> 46			turns a handle to a new communicator to which the Cartesian				
47			ched. If reorder $=$ false then the rank of each process in the				
48	new gr	oup 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. MPI\_CART\_CREATE will associate information representing a Cartesian topology with the specified number of dimensions, numbers of processes in each coordinate direction, and periodicity with the new communicator. MPI

#### 8.5.2 Cartesian Convenience Function: MPI\_DIMS\_CREATE

For Cartesian topologies, the function MPI\_DIMS\_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI\_COMM\_WORLD's group) into an *n*-dimensional topology.

MPI\_DIMS\_CREATE(nnodes, ndims, dims)

			10
IN	nnodes	number of nodes in a grid (integer)	20
IN	ndims	number of Cartesian dimensions (integer)	21
INOUT	dims	integer array of size ndims specifying the number of	22
inte e i	unno		23
		nodes in each dimension	24
			25
C binding	r		26
, c		the second second states that dime (7)	26
int MPI_D	ims_create	int nnodes, int ndims, int dims[])	27
Fortran 2	008 binding		28
			29
MPI_Dims_	create(nnoc	les, ndims, dims, ierror)	
INTEG	ER, INTENT	IN) :: nnodes, ndims	30
	-	INOUT) :: dims(ndims)	31
			32
INTEG	ER, OPTIONA	L, INTENT(OUT) :: ierror	
			33
Fortran b	oinding		34
MDT DTMC	ODEATE (NINOT	EC NOTED DIAG TEDDOD)	

MPI\_DIMS\_CREATE(NNODES, NDIMS, DIMS, IERROR) INTEGER NNODES, NDIMS, DIMS(\*), IERROR

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions 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

$\prod_{\substack{i,dims[i]\neq 0}} dims[i].$	
\$\$\prod_{i, {\sf dims}[i]\neq 0} \mpicode{dims[\$i\$]}.\$\$	

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

387

1

 $\mathbf{2}$ 

3

4

5

9

10

11

12

13

14

15

1617 18

19

35

36 37

38

39

40

41

42

43

44

45

46

47lin PR439

48

#-error

Done

#-314

#-error

Done

in ... +

PR439

1  $\mathbf{2}$ #-72

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  $MPI_SUCCESS$ .

### B.2.2: 10<sup>4</sup>

3

 $\mathbf{5}$ 

48

## Example 8.1

6	-			
7		dims	function call	dims
8		before call		on return
9		(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
10		(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
11		(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
12		(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call
13				
14	8.5.3 Grap	ph Constructor		
15 16				
17	MPI_GRAPI	H_CREATE(cor	nm_old, nnodes, index, edges, reord	er, comm_graph)
18 19	IN	comm_old	input communicator	(handle)
20	IN	nnodes	number of nodes in g	graph (integer)
21	IN	index	array of integers desc	cribing node degrees (see below)
22 23	IN	edges	array of integers desc	cribing graph edges (see below)
24	IN	reorder	ranking may be reord	dered (true) or not (false)
25			(logical)	
26	OUT	comm_graph	communicator with g	graph topology added (handle)
27 28				
29	C binding		PI_Comm comm_old, int nnodes,	const int index[]
30	IIIC MFI_GI	-	c edges[], int reorder, MPI_C	
31	_			
32		008 binding		
33	MP1_Graph_	_create(comm_ ierror)	old, nnodes, index, edges, re	eorder, comm_graph,
34 35	TYPE(M		TENT(IN) :: comm_old	
36			) :: nnodes, index(nnodes), e	edges(*)
37	LOGICAL, INTENT(IN) :: reorder			
38	TYPE(MPI_Comm), INTENT(OUT) :: comm_graph			
39	INTEGE	ER, OPTIONAL,	INTENT(OUT) :: ierror	
40	Fortran bi	nding		
41	MPI_GRAPH_	CREATE (COMM_	OLD, NNODES, INDEX, EDGES, RE	EORDER, COMM_GRAPH,
42 43		IERROR)		
43	INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR			
45	LOGICA	AL REORDER		
46	MPI_G	RAPH_CREAT	E returns a handle to a new comm	nunicator to which the graph
47	topology in	formation is at	tached. If reorder = false then the	e rank of each process in the

new group is identical to its rank in the old group. Otherwise, the function may reorder the

processes. If the size, nnodes, of the graph is smaller than the size of the group of comm\_old, then some processes are returned MPI\_COMM\_NULL, in analogy to MPI\_CART\_CREATE and MPI\_COMM\_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI\_COMM\_NULL is returned in all processes. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

**Example 8.2** Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

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

Then, the input arguments are:

$$\begin{array}{ll} \text{nnodes} = & 4 \\ \text{ndex} = & 2, 3, 4, 6 \end{array}$$

edges = 1, 3, 0, 3, 0, 2

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

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

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{31}$ 

 $45 \\ 46$ 

1	• Type of topology (Cartesian/graph),
2	• For a Cartesian topology:
3	1. ndims (number of dimensions),
4 5	2. dims (numbers of processes per coordinate direction),
6	3. periods (periodicity information),
7	4. own_position (own position in grid, could also be computed from rank and
8	dims)
9	• For a graph topology:
10	1. index,
11	2. edges,
12 13	
14	which are the vectors defining the graph structure.
15	For a graph structure the number of nodes is equal to the number of processes in
16	the group. Therefore, the number of nodes does not have to be stored explicitly.
17	An additional zero entry at the start of array index simplifies access to the topology
18	information. (End of advice to implementors.)
19	9.5.4 Distributed Greek Constructor
20 21	8.5.4 Distributed Graph Constructor
22	MPI_GRAPH_CREATE requires that each process passes the full (global) communication
23	graph to the call. This limits the scalability of this constructor. With the distributed graph
24	interface, the communication graph is specified in a fully distributed fashion. Each process specifies only the part of the communication graph of which it is aware. Typically, this
25	could be the set of processes from which the process will eventually receive or get data,
26	or the set of processes to which the process will send or put data, or some combination of
27	such edges. Two different interfaces can be used to create a distributed graph topology.
28 29	MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with
29 30	each process specifying each of its incoming and outgoing (adjacent) edges in the logical
31	communication graph and thus requires minimal communication during creation.

<sup>31</sup> MPI\_DIST\_GRAPH\_CREATE provides full flexibility such that any process can indicate that <sup>32</sup> communication will occur between any pair of processes in the graph.

To provide better possibilities for optimization by the MPI library, the distributed graph constructors permit weighted communication edges and take an info argument that can further influence process reordering or other optimizations performed by the MPI library. For example, hints can be provided on how edge weights are to be interpreted, the quality of the reordering, and/or the time permitted for the MPI library to process the graph.

- 39
- 40 41
- 42

- 44 45
- 46
- 47
- 48

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT(comm\_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm\_dist\_graph)

	outdegree, destinations, d	estweights, into, reolder, comm_dist_graph)	
IN	comm_old	input communicator (handle)	3
IN	indegree	size of sources and sourceweights arrays (non-negative integer)	4 5 6
IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)	7 8
IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)	9 10 11
IN	outdegree	size of destinations and destweights arrays (non-negative integer)	11 12 13
IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)	14 15
IN	destweights	weights of the edges out of the calling process (array of non-negative integers)	16 17 18
IN	info	hints on optimization and interpretation of weights (handle)	19 20
IN	reorder	the ranks may be reordered (true) or not (false) (logical)	21 22
OUT	comm_dist_graph	communicator with distributed graph topology (handle)	23 24
C binding			25 26 27

## C binding

int M	<pre>IPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,</pre>
	<pre>const int sources[], const int sourceweights[], int outdegree,</pre>
	<pre>const int destinations[], const int destweights[],</pre>
	<pre>MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)</pre>

# Fortran 2008 binding

MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,	33
outdegree, destinations, destweights, info, reorder,	34
comm_dist_graph, ierror)	35
TYPE(MPI_Comm), INTENT(IN) :: comm_old	36
<pre>INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),</pre>	37
<pre>outdegree, destinations(outdegree), destweights(*)</pre>	38
TYPE(MPI_Info), INTENT(IN) :: info	39
LOGICAL, INTENT(IN) :: reorder	40
TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
	43
Fortran binding	44
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,	45
OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,	46

COMM\_DIST\_GRAPH, IERROR)

 $\mathbf{2}$ 

#### INTEGER COMM\_OLD, INDEGREE, SOURCES(\*), SOURCEWEIGHTS(\*), OUTDEGREE, DESTINATIONS(\*), DESTWEIGHTS(\*), INFO, COMM\_DIST\_GRAPH, IERROR LOGICAL REORDER

5MPI\_DIST\_GRAPH\_CREATE\_ADJACENT returns a handle to a new communicator 6 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. 8 The calling processes must ensure that each edge of the graph is described in the source 9 and in the destination process with the same weights. If there are multiple edges for a given 10 (source,dest) pair, then the sequence of the weights of these edges does not matter. The 11 complete communication topology is the combination of all edges shown in the sources arrays 12of all processes in **comm\_old**, which must be identical to the combination of all edges shown 13 in the destinations arrays. Source and destination ranks must be process ranks of comm\_old. 14 This allows a fully distributed specification of the communication graph. Isolated processes 15(i.e., processes with no outgoing or incoming edges, that is, processes that have specified 16indegree and outdegree as zero and thus do not occur as source or destination rank in the 17graph specification) are allowed. 18

The call creates a new communicator comm\_dist\_graph of distributed graph topology 19type to which topology information has been attached. The number of processes in 20comm\_dist\_graph is identical to the number of processes in comm\_old. The call to 21MPI\_DIST\_GRAPH\_CREATE\_ADJACENT is collective. 22

Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between pairs of processes. However, the exact meaning of edge weights is not specified by the MPI standard and is left to the implementation. In C or Fortran, an application can supply the special value MPL\_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. It is erroneous to supply MPI\_UNWEIGHTED for some but not all processes of comm\_old. If the graph is weighted but indegree or outdegree is zero, then MPI\_WEIGHTS\_EMPTY or any arbitrary array may be passed to sourceweights or destweights respectively. Note that MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are not special weight values; rather they are special values for the total array argument. In Fortran, MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are objects like MPI\_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

Advice to users. In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI\_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI\_UNWEIGHTED to the implementation. In this case MPI\_WEIGHTS\_EMPTY should be used instead. (*End of advice to users.*)

Advice to implementors. It is recommended that MPI\_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)

Rationale. To ensure backward compatibility, MPI\_UNWEIGHTED may still be implemented as NULL. See Annex B.4. (End of rationale.)

\constmain{MPI\\_UNWEIGHTED}

\constmain{MPI\\_WEIGHTS\\_EMPTY}

\code{NULL}

1

2

3

4

7

23

24

25

26

27

28

30

31

32

33

34

35

36 37

38

39

40

 $^{41}$ 

42 $^{43}$ 

44

4546

47

 $^{48}$ 

The meaning of the info and reorder arguments is defined in the description of the following routine.

MPI_DIST_GRAPH_CREATE(comm_old, n, source	es, degrees, destinations, weights, info,
reorder, comm_dist_graph)	

	·0 1	,	
IN	comm_old	input communicator (handle)	7
IN	n	number of source nodes for which this process	8
		specifies edges (non-negative integer)	9
IN	sources	array containing the $n$ source nodes for which this	10 11
	5041000	process specifies edges (array of non-negative	11
		integers)	13
IN	degrees	array specifying the number of destinations for each	14
		source node in the source node array (array of	15
		non-negative integers)	16
IN	destinations	destination nodes for the source nodes in the source	17
		node array (array of non-negative integers)	18
IN	weights	weights for source to destination edges (array of	19
IIN	weights	non-negative integers)	20
			21 22
IN	info	hints on optimization and interpretation of weights	22
		(handle)	24
IN	reorder	the ranks may be reordered (true) or not (false)	25
		(logical)	26
OUT	comm_dist_graph	communicator with distributed graph topology	27
		added (handle)	28
			29
C binding	-		30
int MPI_D		n comm_old, int n, const int sources[],	31
		const int destinations[],	32 33
	MPI_Comm *comm_dist_	MPI_Info info, int reorder,	33 34
	MF1_COMM *COMM_dist_	grapu)	35
	008 binding		36
MPI_Dist_	-	, sources, degrees, destinations, weights,	37
	info, reorder, comm_		38
	MPI_Comm), INTENT(IN) ::		39
INTEG	weights(*)	<pre>rces(n), degrees(n), destinations(*),</pre>	40
TVPF (	MPI_Info), INTENT(IN) ::	info	41
	AL, INTENT(IN) :: reorder		42
	MPI_Comm), INTENT(OUT) :		43
	ER, OPTIONAL, INTENT(OUT)		44
			45
Fortran b	0	COMPARE DEADERS DESTINATIONS METAUTS	46 47
MET_DISI_	INFO, REORDER, COMM_	, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,	48

**Unofficial Draft for Comment Only** 

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 5distributed graph topology information is attached. Concretely, each process calls the con-6 structor with a set of directed (source.destination) communication edges as described below. 7 Every process passes an array of n source nodes in the sources array. For each source node, a 8 non-negative number of destination nodes is specified in the degrees array. The destination 9 nodes are stored in the corresponding consecutive segment of the destinations array. More 10 precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the 11 j-th such edge stored in destinations[degrees[0]+ $\dots$ +degrees[i-1]+j]. The weight of this edge 12is stored in weights[degrees[0]+ $\ldots$ +degrees[i-1]+i]. Both the sources and the destinations 13 arrays may contain the same node more than once, and the order in which nodes are listed 14as destinations or sources is not significant. Similarly, different processes may specify edges 15with the same source and destination nodes. Source and destination nodes must be pro-16cess ranks of comm\_old. Different processes may specify different numbers of source and 17destination nodes, as well as different source to destination edges. This allows a fully dis-18 tributed specification of the communication graph. Isolated processes (i.e., processes with 19no outgoing or incoming edges, that is, processes that do not occur as source or destination 20node in the graph specification) are allowed. 21

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  $^{31}$ remapping strategy and other internal MPI optimizations. For instance, approximate count 32 arguments of later communication calls along specific edges could be used as their edge 33 weights. Multiplicity of edges can likewise indicate more intense communication between 34pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 35 standard and is left to the implementation. In C or Fortran, an application can supply 36 the special value MPI\_UNWEIGHTED for the weight array to indicate that all edges have the 37 same (effectively no) weight. It is erroneous to supply MPI\_UNWEIGHTED for some but not 38 all processes of comm\_old. If the graph is weighted but n = 0, then MPI\_WEIGHTS\_EMPTY 39 or any arbitrary array may be passed to weights. Note that MPI\_UNWEIGHTED and 40 MPI\_WEIGHTS\_EMPTY are not special weight values; rather they are special values for the 41 total array argument. In Fortran, MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are objects 42like MPI\_BOTTOM (not usable for initialization or assignment). See Section 2.5.4. 43

Advice to users. In the case of an empty weights array argument passed while
 constructing a weighted graph, one should not pass NULL because the value of
 MPI\_UNWEIGHTED may be equal to NULL. The value of this argument would then
 be indistinguishable from MPI\_UNWEIGHTED to the implementation.
 MPI\_WEIGHTS\_EMPTY should be used instead. (End of advice to users.)

# Unofficial Draft for Comment Only

1

2

3

Advice to implementors. It is recommended that MPI\_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)

Rationale. To ensure backward compatibility, MPI\_UNWEIGHTED may still be implelecode{NULL} mented as NULL. See Annex B.4. (*End of rationale.*)

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is valid for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

Advice to implementors. MPI implementations must document any additionally supported key-value info pairs. MPI\_INFO\_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI\_GRAPH\_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors*.)

**Example 8.3** As for Example 8.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

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

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

process	n	sources	degrees	destinations	weights
0	1	0	2	1,3	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

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

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

## Unofficial Draft for Comment Only

PR439

6 lin

In both cases above, the application could supply MPI\_UNWEIGHTED instead of explicitly providing identical weights.

MPI_DIST_GRAPH_CREATE_ADJACENT	could be used to specify this graph using the
following arguments:	

proce	ss 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 8.4 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:

```
18
```

13

14

15

16

17

```
/*
19
     Input:
                 dimensions P, Q
20
     Condition: number of processes equal to P*Q; otherwise only
21
                ranks smaller than P*Q participate
22
     */
23
     int rank, x, y;
^{24}
     int sources[1], degrees[1];
25
     int destinations[8], weights[8];
26
     MPI_Comm comm_dist_graph;
27
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
28
29
30
     /* get x and y dimension */
^{31}
     y=rank/P; x=rank%P;
32
33
     /* get my communication partners along x dimension */
34
     destinations[0] = P*y+(x+1)%P; weights[0] = 2;
     destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
35
36
37
     /* get my communication partners along y dimension */
38
     destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
39
     destinations[3] = P*((Q+y-1))(Q)+x; weights[3] = 2;
40
41
     /* get my communication partners along diagonals */
42
     destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[4] = 1;
43
     destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
     destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[6] = 1;
44
45
     destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
46
47
     sources[0] = rank;
48
     degrees [0] = 8;
```

**Unofficial Draft for Comment Only** 

1

 $\mathbf{2}$ 

	MPI_Dist_	graph_create(MPI_COMM_WO	RLD, 1, sources, degrees, destinations,	1
		2		
				3
	8.5.5 To		4	
	If a tap ala	mukaghaan dafinad mith ana at	f the shows functions, then the ten elegen information	5
		ked up using inquiry function	f the above functions, then the topology information	6
	can be loo	ked up using inquiry function	s. They all are local calls.	7
				8
	MPI_TOP	O_TEST(comm, status)		9
	IN	comm	communicator (handle)	10
				11
	OUT	status	topology type of communicator <b>comm</b> (state)	12
				13
	C binding	g		14 15
	int MPI_7	Copo_test(MPI_Comm comm,	int *status)	16
	Fortran 2	2008 binding		17
		test(comm, status, ierro:	r)	18
	-	(MPI_Comm), INTENT(IN) ::		19
		ER, INTENT(OUT) :: statu		20
		ER, OPTIONAL, INTENT(OUT)		20
			,	22
	Fortran k	8		23
		TEST (COMM, STATUS, IERRO	R)	24
	INTEG	ER COMM, STATUS, IERROR		25
	The f	unction MPI_TOPO_TEST re	eturns the type of topology that is assigned to a	26
	communic			27
	The o	utput value <b>status</b> is one of th	ne following:	28
\constmainitem	ז{}			29
-	MPI_GRA		graph topology	30 #-error
	MPI_CAF		Cartesian topology	<sub>31</sub> #-update 3
		T_GRAPH	distributed graph topology	32 Done
	MPI_UNI		no topology	<sup>33</sup> in
	There seem	is to be no main index entry for MPI	_UNDEFINED	<sup>34</sup> PR439
				35
	MPI_GRAI	PHDIMS_GET(comm, nnodes,	nedges)	36
	IN	comm	communicator for group with graph structure	37
			(handle)	38
	OUT	nnodes	number of nodes in graph (same as number of	39
	001	modes	processes in the group) (integer)	40
	0.UT		, ,,	41
	OUT	nedges	number of edges in graph (integer)	42
				43
	C bindin	-		44
<pre>int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)</pre>			mm, int *nnodes, int *nedges)	45
	Fortran 2	2008 binding		46
		ndims_get(comm, nnodes, n	edges, ierror)	47
	- 1			48

398
-----

lin

```
1
                  TYPE(MPI_Comm), INTENT(IN) :: comm
        2
                  INTEGER, INTENT(OUT) :: nnodes, nedges
         3
                  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
         4
              Fortran binding
        5
              MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
         6
                  INTEGER COMM, NNODES, NEDGES, IERROR
         7
                  The functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topol-
#-314
        9
             ogy information that is associated with the communicator. No linebreak
#-error
        10
                  The information provided by MPI_GRAPHDIMS_GET can be used to dimension the
Done
        11
              vectors index and edges correctly for the following call to MPI_GRAPH_GET.
        12
              \mpiarg{index} and \mpiarg{edges}
PR439
              (instead of \mpicode)
        13
              MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)
        14
        15
                IN
                         comm
                                                     communicator with graph structure (handle)
        16
                IN
                         maxindex
                                                      length of vector index in the calling program (integer)
        17
                IN
                                                     length of vector edges in the calling program (integer)
                         maxedges
        18
        19
                OUT
                         index
                                                      array of integers containing the graph structure (for
        20
                                                      details see the definition of MPI_GRAPH_CREATE)
        21
                OUT
                         edges
                                                     array of integers containing the graph structure
        22
        23
              C binding
        24
              int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],
        25
                             int edges[])
        26
        27
              Fortran 2008 binding
        28
              MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
        29
                  TYPE(MPI_Comm), INTENT(IN) :: comm
        30
                  INTEGER, INTENT(IN) :: maxindex, maxedges
        ^{31}
                  INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
        32
                  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        33
              Fortran binding
        34
              MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
        35
                  INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
        36
        37
        38
              MPI_CARTDIM_GET(comm, ndims)
        39
        40
                IN
                                                     communicator with Cartesian structure (handle)
                         comm
        41
                OUT
                         ndims
                                                      number of dimensions of the Cartesian structure
        42
                                                      (integer)
        43
        44
              C binding
        45
              int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
        46
        47
              Fortran 2008 binding
        48
```

<pre>MPI_Cartdim_get(comm, ndims, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm     INTEGER, INTENT(OUT) :: ndims</pre>				
INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
Fortran	binding		5 6	
	DIM_GET(COMM, NI		7	
INTE	GER COMM, NDIMS	, IERROR	8	
The f	unctions MPI_CA	RTDIM_GET and MPI_CART_GET return the Cartesian topol-	<sup>9</sup> #-314	
00		ociated with the communicator. If comm is associated with a	10	
		topology, $MPI_CARTDIM_GET$ returns $ndims = 0$ and	11 12	
MPI_CAR	I_GEI will keep a	Il output arguments unchanged.	13	
			14	
MPI_CAR	T_GET(comm, ma	xdims, dims, periods, coords)	15	
IN	comm	communicator with Cartesian structure (handle)	16	
IN	maxdims	length of vectors dims, periods, and coords in the	17	
		calling program (integer)	18 19	
OUT	dims	number of processes for each Cartesian dimension	20	
		(array of integers)	21	
OUT	periods	periodicity (true/false) for each Cartesian dimension	22	
		(array of logicals)	23	
OUT	coords	coordinates of calling process in Cartesian structure	24	
		(array of integers)	25 26	
			20	
C binding				
int MPI_0		<pre>nm comm, int maxdims, int dims[], int periods[],</pre>	29	
	int coords		30	
Fortran 2008 binding				
		ims, dims, periods, coords, ierror)	32	
	(MPI_Comm), INT		33 34	
	GER, INTENT(IN)		35	
		) :: dims(maxdims), coords(maxdims) ) :: periods(maxdims)	36	
		INTENT(OUT) :: ierror	37	
			38	
Fortran	9	IMS, DIMS, PERIODS, COORDS, IERROR)	39	
_	- · ·	MS, DIMS, PERIODS, COORDS, IERROR) MS, DIMS(*), COORDS(*), IERROR	40	
	CAL PERIODS(*)		41 42	
			43	
			44	
			45	
			46	
			47	
			48	

```
1
     MPI_CART_RANK(comm, coords, rank)
2
       IN
                  comm
                                               communicator with Cartesian structure (handle)
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
8
      C binding
9
      int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
10
      Fortran 2008 binding
11
     MPI_Cart_rank(comm, coords, rank, ierror)
12
          TYPE(MPI_Comm), INTENT(IN) :: comm
13
          INTEGER, INTENT(IN) :: coords(*)
14
          INTEGER, INTENT(OUT) :: rank
15
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
      Fortran binding
18
     MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
19
          INTEGER COMM, COORDS(*), RANK, IERROR
20
          For a communicator with an associated Cartesian topology, the function
21
      MPI_CART_RANK translates the logical process coordinates to process ranks as they are
22
     used by the point-to-point routines. No linebreak
23
^{24}
          For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that
     is, coords(i) < 0 or coords(i) \ge dims(i), it is shifted back to the interval [\mbox{...formula...} instead of \flushline
25
      0 \leq coords(i) < dims(i) automatically. Out-of-range coordinates are erroneous for non-
26
      periodic dimensions.
27
          If comm is associated with a zero-dimensional Cartesian topology, coords is not signif-
28
      icant and 0 is returned in rank.
29
30
^{31}
      MPI_CART_COORDS(comm, rank, maxdims, coords)
32
       IN
                                               communicator with Cartesian structure (handle)
33
                  comm
34
        IN
                  rank
                                               rank of a process within group of comm (integer)
35
       IN
                  maxdims
                                               length of vector coords in the calling program
36
                                               (integer)
37
       OUT
                                               integer array (of size maxdims) containing the
38
                  coords
                                               Cartesian coordinates of specified process (array of
39
40
                                               integers)
41
42
     C binding
43
      int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
44
      Fortran 2008 binding
45
      MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
46
          TYPE(MPI_Comm), INTENT(IN) :: comm
47
          INTEGER, INTENT(IN) :: rank, maxdims
48
```

Unofficial Draft for Comment Only

#-311

#-error

#-update3

Done

PR439

in

т мете/			1
	GER, INTENT(OUT) :: coord GER, OPTIONAL, INTENT(OUT		2
Fortran l	binding		$\frac{3}{4}$
	_COORDS(COMM, RANK, MAXDI	MS, COORDS, IERROR)	4 5
INTE	GER COMM, RANK, MAXDIMS,	COORDS(*), IERROR	6
Tho i	nucrea mapping rank to good	rdinates translation is provided by	7
	T_COORDS. No linebreak	tullates translation is provided by	<sup>*</sup> #-error <sup>*</sup> #-353
		-dimensional Cartesian topology,	<sup>9</sup> (PR332)
	be unchanged.		
	0		<sup>11</sup> in
			12 PR439
MPI_GRA	PH_NEIGHBORS_COUNT(cor	nm, rank, nneighbors)	13
IN	comm	communicator with graph topology (handle)	14 15
IN	rank	rank of process in group of comm (integer)	16
OUT	nneighbors	number of neighbors of specified process (integer)	17
			18
C bindin	g		19
int MPI_(	Graph_neighbors_count(MPI	_Comm comm, int rank, int *nneighbors)	20
Fortran 3	2008 binding		21
	n_neighbors_count(comm, r	ank, nneighbors, ierror)	22
-	(MPI_Comm), INTENT(IN) ::	-	23
	GER, INTENT(IN) :: rank		24 25
INTE	GER, INTENT(OUT) :: nneig	hbors	26
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	27
Fortran l	binding		28
	29		
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR) INTEGER COMM, RANK, NNEIGHBORS, IERROR			
			31
			32
MPI_GRA	PH_NEIGHBORS(comm, rank,	maxneighbors, neighbors)	33
IN	comm	communicator with graph topology (handle)	34
			35
IN	rank	rank of process in group of comm (integer)	36 37
IN	maxneighbors	size of array neighbors (integer)	38
OUT	neighbors	ranks of processes that are neighbors to specified	39
		process (array of integers)	40
			41
C bindin	g		42
int MPI_(		comm, int rank, int maxneighbors,	43
	<pre>int neighbors[])</pre>		44
Fortran 2	2008 binding		45
	0	axneighbors, neighbors, ierror)	46
-	(MPI_Comm), INTENT(IN) ::		47
			48

1	<pre>INTEGER, INTENT(IN) :: rank, maxneighbors</pre>
2	INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
3 4	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5	Fortran binding
6	MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
7	INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
8	MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency
9	information for a graph topology. The returned count and array of neighbors for the queried
10	rank will both include all neighbors and reflect the same edge ordering as was specified by
11	the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT
12	and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array
13 14	passed to MPI_GRAPH_CREATE (for the purpose of this example, we assume that index[-1]
14	is zero):
16	• The number of neighbors (nneighbors) returned from
17	MPI_GRAPH_NEIGHBORS_COUNT will be (index[rank] - index[rank-1]).
#-error 18	• The neighbors array returned from MPI_GRAPH_NEIGHBORS will be edges[index[rank-
#-353 <sub>19</sub>	1]] through edges[index[rank]-1].
(PR332) <sub>20</sub>	l] mostour by [maximum] -].
in 21	<b>Example 8.5</b> Assume there are four processes 0, 1, 2, 3 with the following adjacency
PR439 22 23	matrix (note that some neighbors are listed multiple times):
Done 24	
PR439 25	process neighbors
26	$egin{array}{cccccccccccccccccccccccccccccccccccc$
27	$egin{array}{c c} 1 & 0, 0 \ 2 & 3 \end{array}$
28	$\begin{bmatrix} 2 \\ 3 \end{bmatrix} \begin{bmatrix} 0 \\ 0, 2, 2 \end{bmatrix}$
29 30	
31	Thus, the input arguments to MPI_GRAPH_CREATE are:
32	nnodes = 4
33	index = 3, 5, 6, 9
34	edges = 1, 1, 3, 0, 0, 3, 0, 2, 2
35	Therefore, calling MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS for
36	each of the 4 processes will return:
37 38	Input rank Count Neighbors
39	
40	120, 0I would prefere to use this "Lightgray lines on top and bottom, with vertical ticks:"
41	2 1 3 https://github.com/mpi-forum/mpi-issues/
42	<u>3</u> <u>3</u> <u>0</u> , <u>2</u> , <u>2</u> (page 341[printed], 381[pdf])
43	(page 34 [[printed], 36 [[pdi]])
#-error 44	<b>Example 8.6</b> Suppose that comm is a communicator with a shuffle-exchange topology.
#-343 <sup>45</sup>	The group has $2^n$ members. Each process is labeled by $a_1, \ldots, a_n$ with $a_i \in \{0, 1\}$ , and has
#-285 <sup>46</sup> #-PR417 <sup>47</sup>	three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n \ (\bar{a} = 1 - a), \text{ shuffle}(a_1, \ldots, a_n) =$
	$a_2, \ldots, a_n, a_1$ , and unshuffle $(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$ . The graph adjacency list is illustrated below for $n = 3$ .
Done <sup>48</sup> in	$\frac{1}{10} \frac{1}{10} \frac{1}{10} \frac{1}{10} \frac{1}{10} \frac{1}{10} = 0.$
PR417	Unofficial Draft for Comment Only

# Unofficial Draft for Comment Only

https://github.com/mpi-forum/mpi-issues/files/4697966/mpi40-report-extopbottom.pdf#page=381

node		exchange	shuffle	unshuffle	
		neighbors(1)	neighbors(2)	neighbors(3)	
0	(000)	1	0	0	
1	(001)	0	2	4	
2	(010)	3	4	1	
3	(011)	2	6	5	
4	(100)	5	1	2	
5	(101)	4	3	6	
6	(110)	7	5	3	
7	(111)	6	7	7	

Suppose that the communicator **comm** has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

! assume: each process has stored a real number A.
! extract neighborhood information
CALL MPI_COMM_RANK(comm, myrank, ierr)
CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
! perform exchange permutation
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0, &
neighbors(1), 0, comm, status, ierr)
! perform shuffle permutation
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, &
<pre>neighbors(3), 0, comm, status, ierr)</pre>
! perform unshuffle permutation
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0, &
neighbors(2), 0, comm, status, ierr)

 $\mathsf{MPI}_\mathsf{DIST}_\mathsf{GRAPH}_\mathsf{NEIGHBORS}_\mathsf{COUNT}$  and  $\mathsf{MPI}_\mathsf{DIST}_\mathsf{GRAPH}_\mathsf{NEIGHBORS}$  provide adjacency information for a distributed graph topology.

MPI_DIST	_GRAPH_NEIGHBORS_COUN	T(comm, indegree, outdegree, weighted)	33		
IN	comm	communicator with distributed graph topology (handle)	$34 \\ 35$		
Ουτ	indegree	number of edges into this process (non-negative integer)	36 37 38		
OUT	outdegree	number of edges out of this process (non-negative integer)	39 40		
OUT	weighted	false if MPI_UNWEIGHTED was supplied during creation, true otherwise (logical)	41 42 43		
C binding					
int MPI_D	<pre>int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>				

int \*outdegree, int \*weighted)

Fortran 2008 binding

#-error #-343 #-285

1 2 3 4 5 6	<pre>MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm     INTEGER, INTENT(OUT) :: indegree, outdegree     LOGICAL, INTENT(OUT) :: weighted     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>							
7 8 9 10 11	INTEC	8	MM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) GREE, IERROR					
12 13 14	MPI_DIST	GRAPH_NEIGHBORS(comm	n, maxindegree, sources, sourceweights, ons, destweights)					
15 16	IN	comm	communicator with distributed graph topology (handle)					
17 18 19	IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)					
20 21	OUT	sources	processes for which the calling process is a destination (array of non-negative integers)					
22 23	OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)					
24 25 26	IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)					
20 27 28	OUT	destinations	processes for which the calling process is a source (array of non-negative integers)					
29 30	OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)					
31 32 33 34 35	C bindin int MPI_I	) Dist_graph_neighbors(MPI_	Comm comm, int maxindegree, int sources[], , int maxoutdegree, int destinations[],					
36 37 38 39 40 41 42 43 44 45	<pre>Fortran 2008 binding MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,</pre>							
46 47 48	Fortran k MPI_DIST_	_GRAPH_NEIGHBORS(COMM, MA	XINDEGREE, SOURCES, SOURCEWEIGHTS, NATIONS, 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 (potentially by processes other than the calling process in the case of MPI\_DIST\_GRAPH\_CREATE). Multiply-defined edges are all counted and returned by MPI\_DIST\_GRAPH\_NEIGHBORS in some order. If MPI\_UNWEIGHTED is supplied for sourceweights or destweights or both, or if MPI\_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays. If the communicator was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT then for each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in **comm** old in the creation call. If the communicator was created with MPI\_DIST\_GRAPH\_CREATE then the only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI\_DIST\_GRAPH\_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.*)

# 8.5.6 Cartesian Shift Coordinates

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

MPI\_CART\_SHIFT(comm, direction, disp, rank\_source, rank\_dest) IN comm communicator with Cartesian structure (handle) IN direction coordinate dimension of shift (integer) IN disp displacement (> 0: upwards shift, < 0: downwards shift) (integer) OUT rank\_source rank of source process (integer) OUT rank\_dest rank of destination process (integer) 46 C binding 47

```
1
     int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
\mathbf{2}
                     int *rank_source, int *rank_dest)
3
     Fortran 2008 binding
4
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
5
          TYPE(MPI_Comm), INTENT(IN) :: comm
6
          INTEGER, INTENT(IN) :: direction, disp
7
          INTEGER, INTENT(OUT) :: rank_source, rank_dest
8
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     Fortran binding
11
     MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
12
          INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
13
          The direction argument indicates the coordinate dimension to be traversed by the shift.
14
     The dimensions are numbered from 0 to ndims-1, where ndims is the number of dimensions.
15
          Depending on the periodicity of the Cartesian group in the specified coordinate direc-
16
     tion, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case
17
     of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest,
18
     indicating that the source or the destination for the shift is out of range.
19
          It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or
20
     greater than or equal to the number of dimensions in the Cartesian communicator. This
21
     implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with
22
     a zero-dimensional Cartesian topology.
23
^{24}
     Example 8.7 The communicator, comm, has a two-dimensional, periodic, Cartesian topol-
25
     ogy associated with it. A two-dimensional array of REALs is stored one element per process,
26
     in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the
27
     column) by i steps.
28
29
30
     ! find process rank
^{31}
     CALL MPI_COMM_RANK(comm, rank, ierr)
32
     ! find Cartesian coordinates
33
     CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
34
     ! compute shift source and destination
35
     CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
36
     ! skew array
37
     CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
38
                                   status, ierr)
39
40
           Advice to users. In Fortran, the dimension indicated by DIRECTION = i has
41
           DIMS(i+1) nodes, where DIMS is the array that was used to create the grid. In C, the
42
           dimension indicated by direction = i is the dimension specified by dims[i]. (End
           of advice to users.)
43
44
45
46
47
48
```

## 8.5.7 Partitioning of Cartesian Structures

MPI_CART_SUB(comm, I	remain_dims,	newcomm)	1
----------------------	--------------	----------	---

				5
	IN	comm	communicator with Cartesian structure (handle)	6
	IN	remain_dims	the i-th entry of remain_dims specifies whether the	7
			i-th dimension is kept in the subgrid (true) or is	8
			dropped (false) (array of logicals)	9
	OUT	newcomm	communicator containing the subgrid that includes	10
001			the calling process (handle)	11
				12

#### C binding

int MPI\_Cart\_sub(MPI\_Comm comm, const int remain\_dims[], MPI\_Comm \*newcomm)

### Fortran 2008 binding

MPI_Cart_sub(comm, remain_dims, newcomm, ierror)	17
TYPE(MPI_Comm), INTENT(IN) :: comm	18
LOGICAL, INTENT(IN) :: remain_dims(*)	19
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
	22

### Fortran binding

MPI_	_CART_SUB	B(COMM,	, REMAIN_I	DIMS,	NEWC	DMM,	IERROR)
	INTEGER	COMM,	NEWCOMM,	IERRO	)R		
	LOGICAL	REMAIN	I_DIMS(*)				

MPI\_CART\_SUB can be used to partition the group associated with a communicator that has an associated Cartesian topology into subgroups that form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid Cartesian topology. The topologies of the new communicators describe the subgrids. The number of dimensions of the subgrids is the number of remaining dimensions, i.e., the number of true values in remain\_dims. The numbers of processes in each coordinate direction of the subgrids are the remaining numbers of processes in each coordinate direction of the grid associated with the original communicator, i.e., the values of the original grid dimensions for which the corresponding entry in remain\_dims is true. The periodicity for the remaining dimensions in the new communicator is preserved from the original communicator. 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. (This function is closely related to MPI\_COMM\_SPLIT.)

**Example 8.8** Assume that MPI\_Cart\_create(..., comm) has defined a  $(2 \times 3 \times 4)$  grid. Let remain\_dims = (true, false, true). Then a call to

MPI\_Cart\_sub(comm, remain\_dims, newcomm)

will create three communicators each with eight processes in a  $2 \times 4$  Cartesian topology. If remain\_dims = (false, false, true) then the call to

13

14

15 16

23 24 25

26

 $^{28}$ 

29

 $^{31}$ 

32

33

34

35 36

37

38

39

40

41

42 43

44 45

46

47 48

<sup>27</sup> #-314

30 #-315

MPI_0	MPI_Cart_sub(comm, remain_dims, newcomm)						
	will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.						
8.5.8 L	ow-Level Topology F	unctions					
topology is creatin	functions. In general	introduced in this section can be used to implement all other I they will not be called by the user directly, unless he or she opology capability other than that provided by MPI. The two					
MPI_CAF	RT_MAP(comm, ndim	ns, dims, periods, newrank)					
IN	comm	input communicator (handle)					
IN	ndims	number of dimensions of Cartesian structure (integer)					
IN	dims	integer array of size <b>ndims</b> specifying the number of processes in each coordinate direction					
IN	periods	logical array of size ndims specifying the periodicity specification in each coordinate direction					
OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)					
C bindinint MPI	_Cart_map(MPI_Comm	n comm, int ndims, const int dims[], eriods[], int *newrank)					
MPI_Cart TYPE INTE LOGI INTE	E(MPI_Comm), INTEN EGER, INTENT(IN) : ICAL, INTENT(IN) : EGER, INTENT(OUT)	<pre>: ndims, dims(ndims) : periods(ndims)</pre>					
MPI_CART INTE		DIMS, PERIODS, NEWRANK, IERROR) DIMS(*), NEWRANK, IERROR					
ical mach	nine. A possible imple	tes an "optimal" placement for the calling process on the phys- ementation of this function is to always return the rank of the perform any reordering.					
per		a. The function MPI_CART_CREATE(comm, ndims, dims, cart), with reorder = true can be implemented by calling a, ndims, dims, periods, newrank), then calling					

MPI\_COMM\_SPLIT(comm, color, key, comm\_cart), with color = 0 if newrank  $\neq$ MPI\_UNDEFINED, color = MPI\_UNDEFINED otherwise, and key = newrank. If ndims is zero then a zero-dimensional Cartesian topology is created.

The function MPI\_CART\_SUB(comm, remain\_dims, comm\_new) can be implemented by a call to MPI\_COMM\_SPLIT(comm, color, key, comm\_new), using a single number encoding of the lost dimensions as color and a single number encoding of the preserved dimensions as key.

All other Cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (End of advice to implementors.)

The corresponding function for graph structures is as follows.

MPI\_GRAPH\_MAP(comm, nnodes, index, edges, newrank)

			15
IN	comm	input communicator (handle)	16
IN	nnodes	number of graph nodes (integer)	17
IN	index	integer array specifying the graph structure, see	18
		MPI_GRAPH_CREATE	19
			20
IN	edges	integer array specifying the graph structure	21
OUT	newrank	reordered rank of the calling process;	22
		MPI_UNDEFINED if the calling process does not	23
		belong to graph (integer)	24

# C binding

int MPI\_Graph\_map(MPI\_Comm comm, int nnodes, const int index[], const int edges[], int \*newrank)

# Fortran 2008 binding

MPI\_Graph\_map(comm, nnodes, index, edges, newrank, ierror) TYPE(MPI\_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(\*) INTEGER, INTENT(OUT) :: newrank INTEGER, OPTIONAL, INTENT(OUT) :: ierror

# Fortran binding

MPI\_GRAPH\_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) INTEGER COMM, NNODES, INDEX(\*), EDGES(\*), NEWRANK, IERROR

Advice to implementors. The function MPI\_GRAPH\_CREATE(comm, nnodes, index, edges, reorder, comm\_graph), with reorder = true can be implemented by calling MPI\_GRAPH\_MAP(comm, nnodes, index, edges, newrank), then calling MPI\_COMM\_SPLIT(comm, color, key, comm\_graph), with color = 0 if newrank  $\neq$ MPI\_UNDEFINED, color = MPI\_UNDEFINED otherwise, and key = newrank.

All other graph topology functions can be implemented locally, using the topology 46information that is cached with the communicator. (End of advice to implementors.) 47

1 2

3

4

5

6

7

8

9

10 11

1213 14

242526

27

28

29

30

31

32

33

34

3536

37

38

39 40

41

42

43

44

45

1 2

3

4

5

6

7 8

9

17

18

19

20

21

22

23

# 8.6 Neighborhood Collective Communication on Process Topologies

MPI process topologies specify a communication graph, but they implement no communication function themselves. Many applications require sparse nearest neighbor communications that can be expressed as graph topologies. We now describe several collective operations that perform communication along the edges of a process topology. All of these functions are collective; i.e., they must be called by all processes in the specified communicator. See Section 6 for an overview of other dense (global) collective communication operations and the semantics of collective operations.

If the graph was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT with sources and destinations containing 0, ..., n-1, where n is the number of processes in the group of comm\_old (i.e., the graph is fully connected and also includes an edge from each node to itself), then the sparse neighborhood communication routine performs the same data exchange as the corresponding dense (fully-connected) collective operation. In the case of a Cartesian communicator, only nearest neighbor communication is provided, corresponding to rank\_source and rank\_dest in MPI\_CART\_SHIFT with input disp = 1.

Rationale. Neighborhood collective communications enable communication on a process topology. This high-level specification of data exchange among neighboring processes enables optimizations in the MPI library because the communication pattern is known statically (the topology). Thus, the implementation can compute optimized message schedules during creation of the topology [39]. This functionality can significantly simplify the implementation of neighbor exchanges [35]. (End of rationale.)

 $^{24}$ For a distributed graph topology, created with MPI\_DIST\_GRAPH\_CREATE, the se-25quence of neighbors in the send and receive buffers at each process is defined as the sequence 26returned by MPI\_DIST\_GRAPH\_NEIGHBORS for destinations and sources, respectively. For 27a general graph topology, created with MPI\_GRAPH\_CREATE, the use of neighborhood col-28lective communication is restricted to adjacency matrices, where the number of edges be-29tween any two processes is defined to be the same for both processes (i.e., with a symmetric 30 adjacency matrix). In this case, the order of neighbors in the send and receive buffers is  $^{31}$ defined as the sequence of neighbors as returned by MPI\_GRAPH\_NEIGHBORS. Note that 32 general graph topologies should generally be replaced by the distributed graph topologies. 33

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

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

44 45

46

# 8.6.1 Neighborhood Gather

<sup>47</sup> In this function, each process *i* gathers data items from each process *j* if an edge (j, i) exists <sup>48</sup> in the topology graph, and each process *i* sends the same data items to all processes *j* where

# Unofficial Draft for Comment Only

an edge (i, j) exists. The send buffer is sent to each neighboring process and the *l*-th block in the receive buffer is received from the *l*-th neighbor.

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

	comm)		6
IN	sendbuf	starting address of send buffer (choice)	7
IN	sendcount	number of elements sent to each neighbor	8 9
		(non-negative integer)	9 10
IN	sendtype	data type of send buffer elements (handle)	11
OUT	recvbuf	starting address of receive buffer (choice)	12
IN	recvcount	number of elements received from each neighbor	13
		(non-negative integer)	14
			15
IN	recvtype	data type of receive buffer elements (handle)	16
IN	comm	communicator with topology structure (handle)	17
			18

# C binding

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

int MPI\_Neighbor\_allgather\_c(const void \*sendbuf, MPI\_Count sendcount, MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, MPI\_Datatype recvtype, MPI\_Comm comm)

# Fortran 2008 binding

MPI\_Neighbor\_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(\*), DIMENSION(..) :: recvbuf TYPE(MPI\_Comm), INTENT(IN) :: comm

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI\_Neighbor\_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(\*), DIMENSION(...) :: recvbuf TYPE(MPI\_Comm), INTENT(IN) :: comm

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

# Fortran binding

MPI\_NEIGHBOR\_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR)

1

 $\mathbf{2}$ 

3 4

5

19

20

21

22

23<sub>24</sub> #-...

25

26 27

28

29

30

31

32

33

34

35

36

38

39

40

41

42

43

44 45

46

47

48

37 #-...

```
412
                                                                CHAPTER 8. PROCESS TOPOLOGIES
            1
                      <type> SENDBUF(*), RECVBUF(*)
           2
                      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
            3
                      This function supports Cartesian communicators, graph communicators, and distributed
            4
                 graph communicators as described in Section 8.6. If comm is a distributed graph commu-
            5
                 nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
question in 6
                 and receives from each of its incoming neighbors:
#-343
              This is not an example: it is part of the specification of this MPI
            8
                 MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
           9
                 int *srcs=(int*)malloc(indegree*sizeof(int));
           10
                 int *dsts=(int*)malloc(outdegree*sizeof(int));
           11
                 MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
           12
                                               outdegree, dsts, MPI_UNWEIGHTED);
           13
     #-...
                 int k;
           14
           15
                 /* assume sendbuf and recvbuf are of type (char*) */
           16
                 for(k=0; k<outdegree; ++k)</pre>
           17
                   MPI_Isend(sendbuf, sendcount, sendtype,dsts[k],...);
           18
           19
                 for(k=0; k<indegree; ++k)</pre>
           20
                   MPI_Irecv(recvbuf+k*recvcount*extent(recvtype), recvcount, recvtype,
           21
                               srcs[k],...);
           22
           23
                 MPI_Waitall(...);
           24
           25
                      Figure 8.1 shows the neighborhood gather communication of one process with outgoing
           26
                 neighbors d_0 \ldots d_3 and incoming neighbors s_0 \ldots s_5. The process will send its sendbuf to
           27
                 all four destinations (outgoing neighbors) and it will receive the contribution from all six
           28
                 sources (incoming neighbors) into separate locations of its receive buffer.
           29
           30
                                                         d_0
           31
                                                                       d_2, s_4
           32
                                                     s_0
           33
           34
                                                                   s_1
           35
           36
           37
                                                                           s_3
                                                     s_2
           38
                                                                d_{3}, s_{5}
           39
           40
                                  sendbuf
           41
           42
           43
                                                                            s_4
                                                                                    s_5
                                              s_0
                                                      s_1
                                                             s_2
                                                                     s_3
           44
                                  recvbuf
           45
           46
```

Figure 8.1: Neighborhood gather communication example.

47 48

**Unofficial Draft for Comment Only** 

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

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

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

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

**Example 8.9** On a Cartesian virtual grid, the buffer usage in a given direction d with dims[d] == 3 and 1, respectively during creation of the communicator is described in Figure 8.2.

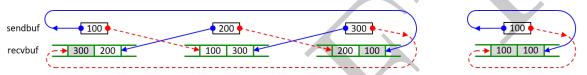


Figure 8.2: Cartesian neighborhood allgather example for 3 and 1 processes in a dimension.

The figure may apply to any (or multiple) directions in the Cartesian topology. The grey buffers are required in all cases but are only accessed if during creation of the communicator, periods[d] was defined as 1 (in C) or .TRUE. (in Fortran).

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

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

IN	sendbuf	starting address of send buffer (choice)	32
IN	sendcount	number of elements sent to each neighbor	33
		(non-negative integer)	34
IN	sendtype	data type of send buffer elements (handle)	35
			36
OUT	recvbuf	starting address of receive buffer (choice)	37
IN	recvcounts	non-negative integer array (of length indegree)	38
		containing the number of elements that are received	39
	*	from each neighbor	40
IN	displs	integer array (of length indegree). Entry i specifies	41
		the displacement (relative to $recvbuf$ ) at which to	42
		place the incoming data from neighbor i	43
IN	recvtype	data type of receive buffer elements (handle)	44
	51		45
IN	comm	communicator with topology structure (handle)	46
			47

# C binding

non-zero

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

 $^{31}$ 

48

#-error!

(PR332)

Done in

PR439

#-343

#-153

B.2.1:1

1int MPI\_Neighbor\_allgatherv(const void \*sendbuf, int sendcount,  $\mathbf{2}$ MPI\_Datatype sendtype, void \*recvbuf, const int recvcounts[], 3 const int displs[], MPI\_Datatype recvtype, MPI\_Comm comm) 4 int MPI\_Neighbor\_allgatherv\_c(const void \*sendbuf, MPI\_Count sendcount, #-... 5 MPI\_Datatype sendtype, void \*recvbuf, 6 const MPI\_Count recvcounts[], const MPI\_Aint displs[],  $\overline{7}$ MPI\_Datatype recvtype, MPI\_Comm comm) 8 9 Fortran 2008 binding 10 MPI\_Neighbor\_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 11 displs, recvtype, comm, ierror) 12TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 13 INTEGER, INTENT(IN) :: sendcount, recvcounts(\*), displs(\*) 14TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 15TYPE(\*), DIMENSION(..) :: recvbuf 16TYPE(MPI\_Comm), INTENT(IN) :: comm 17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 MPI\_Neighbor\_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 19 #-... displs, recvtype, comm, ierror) 20TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 21INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcounts(\*) 22 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 23TYPE(\*), DIMENSION(..) :: recvbuf 24INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: displs(\*) 25TYPE(MPI\_Comm), INTENT(IN) :: comm 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728Fortran binding 29MPI\_NEIGHBOR\_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 30 DISPLS, RECVTYPE, COMM, IERROR) 31<type> SENDBUF(\*), RECVBUF(\*) 32 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 33 IERROR 34 This function supports Cartesian communicators, graph communicators, and distributed 35graph communicators as described in Section 8.6. If comm is a distributed graph commu-36 nicator, the outcome is as if each process executed sends to each of its outgoing neighbors 37 and receives from each of its incoming neighbors: 38 39 MPI\_Dist\_graph\_neighbors\_count(comm, &indegree, &outdegree, &weighted); 40 int \*srcs=(int\*)malloc(indegree\*sizeof(int)); 41 int \*dsts=(int\*)malloc(outdegree\*sizeof(int)); 42MPI\_Dist\_graph\_neighbors(comm, indegree, srcs, MPI\_UNWEIGHTED, 43 outdegree, dsts, MPI\_UNWEIGHTED); 44 int k; 45 46 /\* assume sendbuf and recvbuf are of type (char\*) \*/ 47 for(k=0; k<outdegree; ++k)</pre> 48

```
MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],...);
```

MPI\_Waitall(...);

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[I], recvtype at any other process with srcs[I]==j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data received from the I-th neighbor is placed into recvbuf beginning at offset displs[I] elements (in terms of the recvtype).

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

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

# 8.6.2 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 MPI\_NEIGHBOR\_ALLGATHER in that different data items can be sent to each neighbor. The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in the receive buffer is received from the l-th neighbor.

MPI_NEIGHBOR	L_ALLTOALL(sendbu	f, sendcount, sendtype,	recvbuf, recvcount, recvtype,
--------------	-------------------	-------------------------	-------------------------------

	comm)		29
	,		30
IN	sendbuf	starting address of send buffer (choice)	31
IN	sendcount	number of elements sent to each neighbor	32
		(non-negative integer)	33
IN	sendtype	data type of send buffer elements (handle)	34
<u></u>			35
00-	T recvbuf	starting address of receive buffer (choice)	36
IN	recvcount	number of elements received from each neighbor	37
		(non-negative integer)	38
IN	recvtype	data type of receive buffer elements (handle)	39
	leeveype		40
IN	comm	communicator with topology structure (handle)	41
			42

#### C binding

<sup>з</sup> **#-...** 

```
int MPI_Neighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,
#-...
     \mathbf{2}
                        MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
     3
                        MPI_Datatype recvtype, MPI_Comm comm)
     4
          Fortran 2008 binding
     \mathbf{5}
          MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
     6
                        recvtype, comm, ierror)
     7
              TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
     8
              INTEGER, INTENT(IN) :: sendcount, recvcount
     9
              TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
     10
              TYPE(*), DIMENSION(...) :: recvbuf
     11
              TYPE(MPI_Comm), INTENT(IN) :: comm
     12
              INTEGER, OPTIONAL, INTENT(OUT) :: ierror
     13
    14
          MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
#-...
     15
                        recvtype, comm, ierror)
     16
              TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
     17
              INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
     18
              TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
     19
              TYPE(*), DIMENSION(..) :: recvbuf
     20
              TYPE(MPI_Comm), INTENT(IN) :: comm
     21
              INTEGER, OPTIONAL, INTENT(OUT) :: ierror
     22
          Fortran binding
     23
          MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
     24
                        RECVTYPE, COMM, IERROR)
     25
              <type> SENDBUF(*), RECVBUF(*)
     26
              INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
     27
     28
              This function supports Cartesian communicators, graph communicators, and distributed
     29
          graph communicators as described in Section 8.6. If comm is a distributed graph commu-
     30
          nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
     ^{31}
          and receives from each of its incoming neighbors:
     32
     33
          MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
     34
          int *srcs=(int*)malloc(indegree*sizeof(int));
     35
          int *dsts=(int*)malloc(outdegree*sizeof(int));
     36
          MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
     37
                                     outdegree, dsts, MPI_UNWEIGHTED);
     38
          int k;
     39
     40
          /* assume sendbuf and recvbuf are of type (char*) */
     41
          for(k=0; k<outdegree; ++k)</pre>
     42
            MPI_Isend(sendbuf+k*sendcount*extent(sendtype), sendcount, sendtype,
     43
                       dsts[k],...);
     44
     45
          for(k=0; k<indegree; ++k)</pre>
            MPI_Irecv(recvbuf+k*recvcount*extent(recvtype), recvcount, recvtype,
     46
     47
                       srcs[k],...);
     48
```

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

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of 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.

**Example 8.10** For a halo communication on a Cartesian grid, the buffer usage in a given direction d with dims[d] == 3 and 1, respectively during creation of the communicator is described in Figure 8.3.

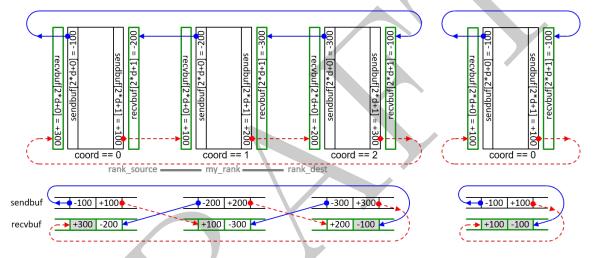


Figure 8.3: Cartesian neighborhood alltoall example for 3 and 1 processes in a dimension.

The figure may apply to any (or multiple) directions in the Cartesian topology. The grey buffers are required in all cases but are only accessed if during creation of the communicator, periods[d] was defined as  $\mathcal{I}(in C)$  or .TRUE. (in Fortran).

If each array element of sendbuf and recvbuf are described by sendcount,sendtype and mpiarg recvbuf,recvtype, then after MPI\_NEIGHBOR\_ALLTOALL on a Cartesian communicator returned, the content of the recvbuf is as if the following code is executed:

```
in
MPI_Cartdim_get(comm, &ndims);
                                                                                            38
                                                                                               PR439
for( /*direction*/ d=0; d < ndims; d++) {</pre>
                                                                                            39
 MPI_Cart_shift(comm, /*direction*/ d, /*disp*/ 1, &rank_source, &rank_dest);
                                                                                            40
 MPI_Sendrecv(sendbuf[d*2+0], sendcount, sendtype, rank_source, /*sendtag*/d*2,
                                                                                            41
                recvbuf[d*2+1],recvcount,recvtype,rank_dest, /*recvtag*/ d*2,
                                                                                            42
                comm,&status);/*communication in direction of displacment -1*/
                                                                                            43
 MPI_Sendrecv(sendbuf[d*2+1],sendcount,sendtype,rank_dest, /*sendtag*/ d*2+1,
                                                                                            44
                recvbuf[d*2+0],recvcount,recvtype,rank_source,/*recvtag*/d*2+1,
                                                                                            45
                comm,&status);/*communication in direction of displacment +1*/
                                                                                            46
}
                                                                                            47
    The first call to MPI_Sendrecv implements the upper (solid arrows) communication
                                                                                               #-error!
                                                                                            48
                                                     solid arrows'
                                                                                               #-343
```

```
Unofficial Draft for Comment Only
```

2

3

4

5

6

8

9

10 11

13

18

19

20 21 22

23

24 25

26

27 28 29

30 31

32

33

34

35

36

37

#-error!

(PR332)

#-343

Done

Done in PR439

12 #-153

B.2.1:1

#-153 B.2.1: 1

# #-error! #-343 Done in PR439 #-error!

#-343 (PR332) Done in

PR439

9

10

11 12

30

31

32

33

34

35

36

3738

418		CHAPTER 8. PROCESS TOPOLOGIES			
pattern i arrows) p		3, whereas the second call is for the lower (dashed dashed arrows'			
Advice to implementors. For a Cartesian topology, if the virtual grid in a direction d 1 or 2 is periodic and dims[d] is equal to f or f, then rank_source and rank_dest are identical, but still all ndims send and ndims receive operations use different buffers. If in this case, the two send and receive operations per direction or of all directions are internally parallelized, then the several send and receive operations for the same sender-receiver shall process pair must be initiated in the same sequence on sender and receiver side or they shall be distinguished by different tags. The code above shows a valid sequence of operations and tags. (End of advice to implementors.)					
	The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different numbers of elements to and from each neighbor.				
MPI_NEI	GHBOR_ALLTOALLV(sendbuf rdispls, recvtype, comm	, sendcounts, sdispls, sendtype, recvbuf, recvcounts, )			
IN	sendbuf	starting address of send buffer (choice)			
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor			
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which to 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) specifying the number of elements that are received			

from each neighbor

integer array (of length indegree). Entry i specifies

the displacement (relative to recvbuf) at which to

place the incoming data from neighbor i

data type of receive buffer elements (handle)

communicator with topology structure (handle)

# C binding

IN

IN

IN

rdispls

recvtype

comm

	39	C binding	
	int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcou		
	41	<pre>const int sdispls[], MPI_Datatype sendtype, void *recvbuf,</pre>	
	42	<pre>const int recvcounts[], const int rdispls[],</pre>	
	43	MPI_Datatype recvtype, MPI_Comm comm)	
#	44	int MPI_Neighbor_alltoallv_c(const void *sendbuf,	
#		<b>o</b>	
	45	<pre>const MPI_Count sendcounts[], const MPI_Aint sdispls[],</pre>	
	46	MPI_Datatype sendtype, void *recvbuf,	
	47	<pre>const MPI_Count recvcounts[], const MPI_Aint rdispls[],</pre>	
	48	MPI_Datatype recvtype, MPI_Comm comm)	

**Unofficial Draft for Comment Only** 

```
Fortran 2008 binding
                                                                                     2
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
              recvcounts, rdispls, recvtype, comm, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
               rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(..) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     10
                                                                                     11
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                        #-...
                                                                                     12
              recvcounts, rdispls, recvtype, comm, ierror)
                                                                                     13
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                     14
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
                                                                                     15
               recvcounts(*)
                                                                                     16
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                     17
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                     18
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                     19
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     21
                                                                                     22
Fortran binding
                                                                                     23
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
                                                                                     ^{24}
              RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
                                                                                     25
    <type> SENDBUF(*), RECVBUF(*)
                                                                                     26
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
               RECVTYPE, COMM, IERROR
                                                                                     27
                                                                                     28
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                     29
graph communicators as described in Section 8.6. If comm is a distributed graph commu-
                                                                                     30
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
                                                                                     ^{31}
and receives from each of its incoming neighbors:
                                                                                     32
                                                                                     33
```

```
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;
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)</pre>
```

34

35

36

37

38

<sup>39</sup> **#-..**.

40

41

42

43

 $\frac{44}{45}$ 

46

47

1					
2	<pre>MPI_Waitall();</pre>				
3					
4	-		sendcounts[k], sendtype with $dsts[k] == j$ at process		
5			sociated with recvcounts[I], recvtype with $srcs[I] = = i$		
6 7	—		ount of data sent must be equal to the amount of ir of communicating processes. Distinct type maps		
8			owed. The data in the sendbuf beginning at offset		
9			<b>pe</b> ) is sent to the k-th outgoing neighbor. The data		
10		· · · · · · · · · · · · · · · · · · ·	r is placed into recvbuf beginning at offset rdispls[1]		
11		in terms of the recvtype).			
12	The "	in place" option is not meaning	ngful for this operation.		
13			processes and the argument comm must have iden-		
14		s on all processes.			
15			we one to send and receive with different datatypes		
16 17	to and from	n each neighbor.			
18					
19	MPI_NEIG	HBOR_ALLTOALLW(sendbuf,	sendcounts, sdispls, sendtypes, recvbuf, recvcounts,		
20		rdispls, recvtypes, comm)			
21	IN	sendbuf	starting address of send buffer (choice)		
22 23	IN	sendcounts	non-negative integer array (of length outdegree)		
24			specifying the number of elements to send to each		
25			neighbor		
26	IN	sdispls	integer array (of length outdegree). Entry j specifies		
27			the displacement in bytes (relative to sendbuf) from		
28			which to take the outgoing data destined for		
29			neighbor j (array of integers)		
30	IN	sendtypes	array of datatypes (of length outdegree). Entry j		
31 32			specifies the type of data to send to neighbor j (array		
33			of handles)		
34	OUT	recvbuf	starting address of receive buffer (choice)		
35	IN	recvcounts	non-negative integer array (of length indegree)		
36			specifying the number of elements that are received		
37			from each neighbor		
38	IN	rdispls	integer array (of length indegree). Entry i specifies		
39			the displacement in bytes (relative to recvbuf) at		
40 41		Ť	which to place the incoming data from neighbor i (array of integers)		
42			· · · · · · · · · · · · · · · · · · ·		
43	IN	recvtypes	array of datatypes (of length indegree). Entry i		
44			specifies the type of data received from neighbor i (array of handles)		
45	184				
46	IN	comm	communicator with topology structure (handle)		
47		_			
48	C binding	r S			

```
1
int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[],
                                                                                     2
              const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
              void *recvbuf, const int recvcounts[],
              const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
                                                                                     4
                                                                                     5
              MPI Comm comm)
                                                                                     6
int MPI_Neighbor_alltoallw_c(const void *sendbuf,
                                                                                      #-...
                                                                                     \overline{7}
              const MPI_Count sendcounts[], const MPI_Aint sdispls[],
                                                                                     8
              const MPI_Datatype sendtypes[], void *recvbuf,
                                                                                     9
              const MPI_Count recvcounts[], const MPI_Aint rdispls[],
                                                                                     10
              const MPI_Datatype recvtypes[], MPI_Comm comm)
                                                                                     11
Fortran 2008 binding
                                                                                     12
                                                                                     13
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                     14
              recvcounts, rdispls, recvtypes, comm, ierror)
                                                                                     15
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
                                                                                     16
                                                                                     17
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                     18
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                     19
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                     20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    22
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                    23 #-...
              recvcounts, rdispls, recvtypes, comm, ierror)
                                                                                     ^{24}
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                     25
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
                                                                                     26
               recvcounts(*)
                                                                                     27
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                     28
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                     29
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                     30
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     ^{31}
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     32
                                                                                     33
Fortran binding
                                                                                    34
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                                                                                     35
              RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
                                                                                     36
    <type> SENDBUF(*), RECVBUF(*)
                                                                                    37
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
                                                                                     38
               IERROR
                                                                                     39
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
                                                                                     40
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                    41
graph communicators as described in Section 8.6. If comm is a distributed graph commu-
                                                                                     42
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
                                                                                     43
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));
```

44 45

46

47

```
1
          MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
     2
                                         outdegree, dsts, MPI_UNWEIGHTED);
     3
#-...
           int k;
     \mathbf{5}
          /* assume sendbuf and recvbuf are of type (char*) */
     6
          for(k=0; k<outdegree; ++k)</pre>
     \overline{7}
             MPI_Isend(sendbuf+sdispls[k], sendcounts[k], sendtypes[k], dsts[k],...);
     8
     9
          for(k=0; k<indegree; ++k)</pre>
     10
             MPI_Irecv(recvbuf+rdispls[k], recvcounts[k], recvtypes[k], srcs[k],...);
    11
    12
          MPI_Waitall(...);
     13
               The type signature associated with sendcounts[k], sendtypes[k] with dsts[k] == j at pro-
     14
           cess i must be equal to the type signature associated with recvcounts[1], recvtypes[1] with
     15
           srcs[I] == i at process j. This implies that the amount of data sent must be equal to the
     16
           amount of data received, pairwise between every pair of communicating processes. Distinct
     17
           type maps between sender and receiver are still allowed.
     18
               The "in place" option is not meaningful for this operation.
     19
               All arguments are significant on all processes and the argument comm must have iden-
     20
           tical values on all processes.
     21
     22
                 Nonblocking Neighborhood Communication on Process Topologies
    23
           8.7
     24
     25
           Nonblocking variants of the neighborhood collective operations allow relaxed synchroniza-
     26
           tion and overlapping of computation and communication. The semantics are similar to
    27
           nonblocking collective operations as described in Section 6.12.
     28
     29
                  Nonblocking Neighborhood Gather
           8.7.1
     30
     ^{31}
     32
           MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
     33
                          comm, request)
     34
            1N
                       sendbuf
                                                    starting address of send buffer (choice)
     35
     36
            IN
                       sendcount
                                                     number of elements sent to each neighbor
     37
                                                     (non-negative integer)
     38
            IN
                       sendtype
                                                     data type of send buffer elements (handle)
     39
            OUT
                       recvbuf
                                                     starting address of receive buffer (choice)
     40
     41
            IN
                                                     number of elements received from each neighbor
                       recvcount
     42
                                                     (non-negative integer)
     43
            IN
                                                     data type of receive buffer elements (handle)
                       recvtype
     44
            IN
                                                     communicator with topology structure (handle)
                       comm
     45
            OUT
                                                    communication request (handle)
                       request
     46
     47
```

<sup>48</sup> C binding

```
1
int MPI_Ineighbor_allgather(const void *sendbuf, int sendcount,
                                                                                     2
              MPI_Datatype sendtype, void *recvbuf, int recvcount,
                                                                                     3
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
                                                                                     4
int MPI_Ineighbor_allgather_c(const void *sendbuf, MPI_Count sendcount,
                                                                                     5 #-...
              MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
                                                                                     6
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
                                                                                     \overline{7}
                                                                                     8
Fortran 2008 binding
                                                                                     9
MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                    10
              recvtype, comm, request, ierror)
                                                                                    11
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                    12
                                                                                    13
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                    14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    16
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    18
MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                    <sup>19</sup> #-...
              recvtype, comm, request, ierror)
                                                                                    20
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    21
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                    22
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                    23
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                    24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    27
                                                                                    28
Fortran binding
                                                                                    29
MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                    30
              RECVTYPE, COMM, REQUEST, IERROR)
                                                                                    31
    <type> SENDBUF(*), RECVBUF(*)
                                                                                    32
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
                                                                                    33
    This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHER.
                                                                                    34
                                                                                    35
                                                                                    36
                                                                                    37
                                                                                    38
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
                                                                                    46
                                                                                    47
                                                                                    48
```

	1 2	MPI_INEI	GHBOR_ALLGATHER\ recvtype, comm,	/(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, request)
	3	IN	sendbuf	starting address of send buffer (choice)
	4 5 6	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
	7	IN	sendtype	data type of send buffer elements (handle)
	8	OUT	recvbuf	starting address of receive buffer (choice)
	9 10 11 12	IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor
	13 14 15	IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to <b>recvbuf</b> ) at which to place the incoming data from neighbor i
	16	IN	recvtype	data type of receive buffer elements (handle)
	17 18	IN	comm	communicator with topology structure (handle)
	19	OUT	request	communication request (handle)
	20			
	21 22 23 24 25 26	C binding int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)		
#	<pre># 27 int MPI_Ineighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount 28 MPI_Datatype sendtype, void *recvbuf, 29 const MPI_Count recvcounts[], const MPI_Aint displs[], 30 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request</pre>			<pre>sendtype, void *recvbuf, it recvcounts[], const MPI_Aint displs[],</pre>
#	<ul> <li>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</li> <li>Fortran 2008 binding</li> <li>MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror)</li> <li>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf</li> <li>INTEGER, INTENT(IN) :: sendcount</li> <li>TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvtype</li> <li>TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvcounts(*), displs(*)</li> <li>INTEGER, INTENT(IN) ASYNCHRONOUS :: recvcounts(*), displs(*)</li> <li>TYPE(MPI_Comm), INTENT(OUT) :: request</li> <li>INTEGER, OPTIONAL, INTENT(OUT) :: ierror</li> <li>MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror)</li> <li>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf</li> <li>INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount</li> <li>TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype</li> <li>TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype</li> </ul>			
			Unoffi	cial Draft for Comment Only

INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(\*) 1  $\mathbf{2}$ INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN), ASYNCHRONOUS :: displs(\*) 3 TYPE(MPI\_Comm), INTENT(IN) :: comm 4 TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 56 Fortran binding MPI\_INEIGHBOR\_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(\*), RECVBUF(\*) 10 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 11 REQUEST, IERROR 1213 This call starts a nonblocking variant of MPI\_NEIGHBOR\_ALLGATHERV. 1415Nonblocking Neighborhood Alltoall 8.7.2 161718 MPI\_INEIGHBOR\_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 19 comm, request) 20IN sendbuf starting address of send buffer (choice) 2122 IN sendcount number of elements sent to each neighbor 23(non-negative integer) 24IN sendtype data type of send buffer elements (handle) 25OUT recvbuf starting address of receive buffer (choice) 2627IN number of elements received from each neighbor recvcount 28 (non-negative integer) 29IN recvtype data type of receive buffer elements (handle) 30 IN communicator with topology structure (handle) comm 3132 OUT communication request (handle) request 33 34 C binding 35int MPI\_Ineighbor\_alltoall(const void \*sendbuf, int sendcount, 36 MPI\_Datatype sendtype, void \*recvbuf, int recvcount, 37 MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Request \*request) 38 int MPI\_Ineighbor\_alltoall\_c(const void \*sendbuf, MPI\_Count sendcount, 39 **#-..** MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 40 MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Request \*request) 4142Fortran 2008 binding 43 MPI\_Ineighbor\_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, 44recvtype, comm, request, ierror) 45TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 46INTEGER, INTENT(IN) :: sendcount, recvcount 47TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 48

```
1
                TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
      2
                TYPE(MPI_Comm), INTENT(IN) :: comm
      3
                TYPE(MPI_Request), INTENT(OUT) :: request
      4
                INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       5
            MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
#-...
      6
                           recvtype, comm, request, ierror)
      \overline{7}
                TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
      8
                INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
      9
                TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
      10
                TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
      11
                TYPE(MPI_Comm), INTENT(IN) :: comm
      12
                TYPE(MPI_Request), INTENT(OUT) :: request
      13
                INTEGER, OPTIONAL, INTENT(OUT) :: ierror
      14
      15
            Fortran binding
      16
            MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
      17
                           RECVTYPE, COMM, REQUEST, IERROR)
      18
                <type> SENDBUF(*), RECVBUF(*)
      19
                INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
      20
                This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALL.
      21
      22
      23
            MPI_INEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
      24
                            rdispls, recvtype, comm, request)
      25
              IN
                        sendbuf
                                                     starting address of send buffer (choice)
      26
      27
              IN
                        sendcounts
                                                     non-negative integer array (of length outdegree)
                                                     specifying the number of elements to send to each
      28
      29
                                                     neighbor
      30
              IN
                        sdispls
                                                     integer array (of length outdegree). Entry j specifies
      ^{31}
                                                     the displacement (relative to sendbuf) from which
      32
                                                     send the outgoing data to neighbor j
      33
              IN
                        sendtype
                                                     data type of send buffer elements (handle)
      34
              OUT
      35
                        recvbuf
                                                     starting address of receive buffer (choice)
      36
              IN
                        recvcounts
                                                     non-negative integer array (of length indegree)
      37
                                                     specifying the number of elements that are received
      38
                                                     from each neighbor
      39
              IN
                        rdispls
                                                     integer array (of length indegree). Entry i specifies
      40
                                                     the displacement (relative to recvbuf) at which to
      41
                                                     place the incoming data from neighbor i
      42
              IN
      43
                        recvtype
                                                     data type of receive buffer elements (handle)
      44
              IN
                        comm
                                                     communicator with topology structure (handle)
      45
              OUT
                                                     communication request (handle)
                        request
      46
      47
            C binding
      48
```

```
1
int MPI_Ineighbor_alltoallv(const void *sendbuf, const int sendcounts[],
                                                                                   2
              const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
                                                                                    3
              const int recvcounts[], const int rdispls[],
                                                                                   4
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
                                                                                   5
int MPI_Ineighbor_alltoallv_c(const void *sendbuf,
                                                                                   6 #-...
              const MPI_Count sendcounts[], const MPI_Aint sdispls[],
                                                                                   \overline{7}
              MPI_Datatype sendtype, void *recvbuf,
                                                                                   8
              const MPI_Count recvcounts[], const MPI_Aint rdispls[],
                                                                                   9
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
                                                                                   10
                                                                                   11
Fortran 2008 binding
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                   12
              recvcounts, rdispls, recvtype, comm, request, ierror)
                                                                                   13
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   14
                                                                                   15
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                   16
              recvcounts(*), rdispls(*)
                                                                                   17
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   18
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   19
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   20
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   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
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                   26
              sendcounts(*), recvcounts(*)
                                                                                   27
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                   28
              rdispls(*)
                                                                                   29
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   30
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                   ^{31}
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   32
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   34
                                                                                   35
Fortran binding
                                                                                   36
MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
                                                                                   37
             RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
                                                                                   38
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   39
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                   40
              RECVTYPE, COMM, REQUEST, IERROR
                                                                                   41
    This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.
                                                                                   42
                                                                                   43
                                                                                   44
                                                                                   45
                                                                                   46
                                                                                   47
                                                                                   48
```

1 2	MPI_INEI	GHBOR_ALLTOALLW(send rdispls, recvtypes, con	buf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, nm, request)
3 4	IN	sendbuf	starting address of send buffer (choice)
5 6 7	IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
8 9 10 11	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)
12 13 14 15	IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)
16	OUT	recvbuf	starting address of receive buffer (choice)
17 18 19	IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor
20 21 22 23 24	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to <b>recvbuf</b> ) at which to place the incoming data from neighbor i (array of integers)
25 26 27	IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)
28	IN	comm	communicator with topology structure (handle)
29 30	OUT	request	communication request (handle)
31 32	C bindir	ıg	
33			st void *sendbuf, const int sendcounts[],
34			<pre>ispls[], const MPI_Datatype sendtypes[],</pre>
35			<pre>nst int recvcounts[], ispls[], const MPI_Datatype recvtypes[],</pre>
$\frac{36}{37}$			I_Request *request)
38			
39	<pre>int MPI_Ineighbor_alltoallw_c(const void *sendbuf,</pre>		
40			e sendtypes[], void *recvbuf,
41 42			<pre>ecvcounts[], const MPI_Aint rdispls[], e recvtypes[], MPI_Comm comm,</pre>
43		MPI_Request *reque	
44	Fortron	2008 binding	
45 46		6	, sendcounts, sdispls, sendtypes, recvbuf,
40	_ 1	-	ls, recvtypes, comm, request, ierror)
48	TYPE	(*), DIMENSION(), INT	TENT(IN), ASYNCHRONOUS :: sendbuf

#-...

```
1
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                      \mathbf{2}
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
               rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                      4
               recvtypes(*)
                                                                                      5
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                      6
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      9
                                                                                      10
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                     <sup>11</sup> #-...
              recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                      12
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                     13
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                     14
               sendcounts(*), recvcounts(*)
                                                                                      15
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                      16
               rdispls(*)
                                                                                      17
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                      18
               recvtypes(*)
                                                                                      19
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                      20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     21
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                      22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     23
                                                                                      ^{24}
Fortran binding
MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                                                                                      25
                                                                                      26
              RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
                                                                                     27
    <type> SENDBUF(*), RECVBUF(*)
                                                                                     28
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
                                                                                     29
               REQUEST, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
                                                                                      30
                                                                                      31
    This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW.
                                                                                      32
                                                                                      33
                                                                                     34
8.8 Persistent Neighborhood Communication on Process Topologies
                                                                                        #-25
                                                                                      35
                                                                                        B.2.2: 3
Persistent variants of the neighborhood collective operations can offer significant perfor-
                                                                                     36
mance benefits for programs with repetitive communication patterns. The semantics are
                                                                                     37
similar to persistent collective operations as described in Section 6.13.
                                                                                     38
                                                                                      39
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
                                                                                      45
                                                                                      46
                                                                                      47
```

	430		CHAPTER 8. PROCESS TOPOLOGIES	
1 2 3	8.8.1 Per	sistent Neighborhood Gather		
4 5	MPI_NEIG	HBOR_ALLGATHER_INIT(sen recvtype, comm, info, req	dbuf, sendcount, sendtype, recvbuf, recvcount, uest)	
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	IN	info	info argument (handle)	
19 20	OUT	request	communication request (handle)	
24 25 26 27 28 29 30	<pre>MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request) int MPI_Neighbor_allgather_init_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>			
<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> <li>43</li> <li>44</li> <li>45</li> </ol>	MPI_Neigh TYPE( INTEG TYPE( TYPE( TYPE( TYPE( INTEG MPI_Neigh	recvcount, recvtype, *), DIMENSION(), INTENT ER, INTENT(IN) :: sendcou MPI_Datatype), INTENT(IN) *), DIMENSION(), ASYNCH MPI_Comm), INTENT(IN) :: MPI_Info), INTENT(IN) :: MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT) bor_allgather_init(sendbu recvcount, recvtype,	<pre>:: sendtype, recvtype RONOUS :: recvbuf comm info     :: request     :: ierror f, sendcount, sendtype, recvbuf,     comm, info, request, ierror)</pre>	
46 47 48	INTEG TYPE()		VI VI	

#-25

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                              #-25
                                                                                            2
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                            3
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                            4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                            5
Fortran binding
                                                                                            6
MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                                                                                            7
               RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
                                                                                            8
    <type> SENDBUF(*), RECVBUF(*)
                                                                                            9
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
                                                                                            10
                IERROR
                                                                                            11
                                                                                            12
    Creates a persistent collective communication request for the neighborhood allgather
                                                                                            13
operation.
                                                                                            14
                                                                                            15
MPI_NEIGHBOR_ALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                            16
               displs, recvtype, comm, info, request)
                                                                                            17
                                                                                            18
  IN
           sendbuf
                                        starting address of send buffer (choice)
                                                                                            19
  IN
           sendcount
                                        number of elements sent to each neighbor
                                                                                            20
                                        (non-negative integer)
                                                                                            21
  IN
           sendtype
                                        data type of send buffer elements (handle)
                                                                                            22
  OUT
           recvbuf
                                        starting address of receive buffer (choice)
                                                                                            23
                                                                                            24
  IN
            recvcounts
                                        non-negative integer array (of length indegree)
                                                                                            25
                                        containing the number of elements that are received
                                                                                            26
                                        from each neighbor
                                                                                            27
  IN
            displs
                                        integer array (of length indegree). Entry i specifies
                                                                                            28
                                        the displacement (relative to recvbuf) at which to
                                                                                            29
                                        place the incoming data from neighbor i
                                                                                            30
  IN
                                        data type of receive buffer elements (handle)
            recvtype
                                                                                            ^{31}
                                                                                            32
  IN
            comm
                                        communicator with topology structure (handle)
                                                                                            33
  IN
            info
                                        info argument (handle)
                                                                                            34
  OUT
            request
                                        communication request (handle)
                                                                                            35
                                                                                            36
C binding
                                                                                            37
int MPI_Neighbor_allgatherv_init(const void *sendbuf, int sendcount,
                                                                                            38
                                                                                            39
               MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
                                                                                            40
               const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
                                                                                            41
               MPI_Info info, MPI_Request *request)
                                                                                            42
int MPI_Neighbor_allgatherv_init_c(const void *sendbuf,
                                                                                            43
               MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf,
                                                                                            44
               const MPI_Count recvcounts[], const MPI_Aint displs[],
                                                                                            45
               MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
                                                                                            46
               MPI_Request *request)
                                                                                            47
```

```
Fortran 2008 binding
```

Unofficial Draft for Comment Only

```
MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
#-25<sup>1</sup>
     \mathbf{2}
                        recvcounts, displs, recvtype, comm, info, request, ierror)
     3
              TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
     4
              INTEGER, INTENT(IN) :: sendcount, displs(*)
     5
              TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
     6
              TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
     7
              INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
     8
              TYPE(MPI_Comm), INTENT(IN) :: comm
     9
              TYPE(MPI_Info), INTENT(IN) :: info
     10
              TYPE(MPI_Request), INTENT(OUT) :: request
     11
              INTEGER, OPTIONAL, INTENT(OUT) :: ierror
     12
          MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
     13
                        recvcounts, displs, recvtype, comm, info, request, ierror)
     14
              TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
     15
              INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
     16
              TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
     17
              TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
     18
              INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
     19
              INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
     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
     25
          Fortran binding
     26
          MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
     27
                        RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
     ^{28}
              <type> SENDBUF(*), RECVBUF(*)
     29
              INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
     30
                         INFO, REQUEST, IERROR
     31
              Creates a persistent collective communication request for the neighborhood allgathery
     32
          operation.
     33
     34
     35
     36
     37
     38
     39
     40
     41
     42
     43
     44
     45
     46
     47
    ^{48}
```

### 8.8.2 Persistent Neighborhood Alltoall

# MPI\_NEIGHBOR\_ALLTOALL\_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	number of elements received from each neighbor (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

## C binding

#### Fortran 2008 binding

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

#-25

#-25 <sup>1</sup>	TYPI	E(MPI_Comm), INT	ENT(IN) :: comm				
2	TYPI	E(MPI_Info), INT	ENT(IN) :: info				
3	TYPI	E(MPI_Request),	INTENT(OUT) :: request				
4		INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
5	Fortran	Fortran binding					
7		MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,					
8		RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)					
9	<ty]< td=""><td>pe&gt; SENDBUF(*), 1</td><td>RECVBUF(*)</td></ty]<>	pe> SENDBUF(*), 1	RECVBUF(*)				
10		INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR					
11							
12	Ulea	Creates a persistent collective communication request for the neighborhood alltoall					
13 14	operation	n.					
15							
16		IGHBOR_ALLTOAL	LV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf,				
17	-	recvcounts, i	rdispls, recvtype, comm, info, request)				
18	IIN	sendbuf	starting address of send buffer (choice)				
19	IN	sendcounts	non-negative integer array (of length outdegree)				
20 21			specifying the number of elements to send to each				
22			neighbor				
23	3 IN	sdispls	integer array (of length outdegree). Entry j specifies				
24	L		the displacement (relative to sendbuf) from which				
25	5		send the outgoing data to neighbor j				
26	IIN	sendtype	data type of send buffer elements (handle)				
27 28	OUT	recvbuf	starting address of receive buffer (choice)				
29	) IN	recvcounts	non-negative integer array (of length indegree)				
30 31			specifying the number of elements that are received from each neighbor				
32	<sup>2</sup> IN	rdispls	integer array (of length indegree). Entry i specifies				
33		l'alopio	the displacement (relative to recvbuf) at which to				
34			place the incoming data from neighbor i				
35 36	IN	recvtype	data type of receive buffer elements (handle)				
37	INI	comm	communicator with topology structure (handle)				
38		info	info argument (handle)				
39 40	OUT	request	communication request (handle)				
41							
42		-	lle init (court maid the endbut				
43		-	<pre>llv_init(const void *sendbuf, sendcounts[], const int sdispls[],</pre>				
44			<pre>re sendtype, void *recvbuf, const int recvcounts[],</pre>				
45 46		•	rdispls[], MPI_Datatype recvtype, MPI_Comm comm,				
40			.nfo, MPI_Request *request)				
48							

```
int MPI_Neighbor_alltoallv_init_c(const void *sendbuf,
                                                                                     #-25
                                                                                    \mathbf{2}
              const MPI_Count sendcounts[], const MPI_Aint sdispls[],
                                                                                   3
              MPI_Datatype sendtype, void *recvbuf,
                                                                                   4
              const MPI_Count recvcounts[], const MPI_Aint rdispls[],
                                                                                   5
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
                                                                                    6
              MPI_Request *request)
                                                                                    \overline{7}
Fortran 2008 binding
                                                                                    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
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                   13
              recvcounts(*), rdispls(*)
                                                                                   14
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   15
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   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_init(sendbuf, sendcounts, sdispls, sendtype,
                                                                                   22
              recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
                                                                                   23
              ierror)
                                                                                   24
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   25
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                   26
              sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                   27
                                                                                   28
              rdispls(*)
                                                                                   29
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   30
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   32
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   35
Fortran binding
                                                                                   36
MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
                                                                                   37
              RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
                                                                                   38
              IERROR)
                                                                                   39
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   40
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                   41
              RECVTYPE, COMM, INFO, REQUEST, IERROR
                                                                                   42
                                                                                   43
    Creates a persistent collective communication request for the neighborhood alloally
                                                                                   44
operation.
                                                                                   45
```

1	MPI NFI	CHROR ALLTOALLW	INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
2			ls, recvtypes, comm, info, request)
3 4	IN	sendbuf	starting address of send buffer (choice)
5 6 7	IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to ea neighbor
8 9 10 11	IN	sdispls	integer array (of length outdegree). Entry j spectule displacement in bytes (relative to sendbuf) is which to take the outgoing data destined for neighbor j (array of integers)
12 13 14 15	IN	sendtypes	array of datatypes (of length outdegree). Entry specifies the type of data to send to neighbor j of handles)
.6	OUT	recvbuf	starting address of receive buffer (choice)
17 18 19	IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are rece from each neighbor
20 21 22 23 24	IN	rdispls	integer array (of length indegree). Entry i speci the displacement in bytes (relative to recvbuf) a which to place the incoming data from neighbor (array of integers)
25 26 27	IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbo (array of handles)
28	IN	comm	communicator with topology structure (handle)
9 0	IN	info	info argument (handle)
1	OUT	request	communication request (handle)
2 3 4	C bindini int MPI	U U	_init(const void *sendbuf,
35 36 37 38 39		const MPI_Dat const int rec	<pre>dcounts[], const MPI_Aint sdispls[], atype sendtypes[], void *recvbuf, vcounts[], const MPI_Aint rdispls[], atype recvtypes[], MPI_Comm comm, MPI_Info in request)</pre>
	int MPI_	const MPI_Cou const MPI_Dat const MPI_Cou	_init_c(const void *sendbuf, nt sendcounts[], const MPI_Aint sdispls[], atype sendtypes[], void *recvbuf, nt recvcounts[], const MPI_Aint rdispls[], atype recvtypes[], MPI_Comm comm, MPI_Info in request)
45 46 47	Fortran		

```
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
                                                                                      #-25
                                                                                    \mathbf{2}
              recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
                                                                                    3
              ierror)
                                                                                    4
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    5
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                    6
                                                                                    7
               rdispls(*)
                                                                                    8
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                    9
               recvtypes(*)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    10
                                                                                    11
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                    12
                                                                                    13
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    15
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
                                                                                    16
              recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
                                                                                    17
              ierror)
                                                                                    18
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    19
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                    20
               sendcounts(*), recvcounts(*)
                                                                                    21
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                    22
              rdispls(*)
                                                                                    23
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                    24
               recvtypes(*)
                                                                                    25
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    26
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    27
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                    28
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    30
                                                                                    ^{31}
Fortran binding
                                                                                    32
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
                                                                                    33
              RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
                                                                                    34
              IERROR)
                                                                                    35
    <type> SENDBUF(*), RECVBUF(*)
                                                                                    36
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
                                                                                    37
               INFO, REQUEST, IERROR
                                                                                    38
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
                                                                                    39
    Creates a persistent collective communication request for the neighborhood alltoallw
                                                                                    40
```

operation.

#### 8.9 An Application Example

**Example 8.11** The example in Figures 8.4-8.7 shows how the grid definition and inquiry 46 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

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

41 42

43 44 45

47

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.

<sup>4</sup> In each relaxation step each process computes new values for the solution grid function <sup>5</sup> at the points u(1:100,1:100) owned by the process. Then the values at inter-process <sup>6</sup> boundaries have to be exchanged with neighboring processes. For example, the newly <sup>7</sup> calculated values in u(1,1:100) must be sent into the halo cells u(101,1:100) of the <sup>8</sup> left-hand neighbor with coordinates (own\_coord(1)-1,own\_coord(2)).

 $^{24}$  $^{31}$ 

```
INTEGER ndims, num_neigh
                                                                                      1
LOGICAL reorder
                                                                                      \mathbf{2}
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
                                                                                      3
INTEGER comm, comm_size, comm_cart, dims(ndims), ierr
                                                                                      4 #-...
INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
                                                                                      5
LOGICAL periods(ndims)
                                                                                      6
REAL u(0:101,0:101), f(0:101,0:101)
                                                                                      7
DATA dims / ndims * 0 /
                                                                                      8
comm = MPI_COMM_WORLD
                                                                                      9
CALL MPI_COMM_SIZE(comm, comm_size, ierr)
                                                                                      10
    Set process grid size and periodicity
!
                                                                                      11
CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
                                                                                      12
periods(1) = .TRUE.
                                                                                      13
periods(2) = .TRUE.
                                                                                      14
    Create a grid structure in WORLD group and inquire about own position
1
                                                                                      15
CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
                                                                                      16
                       comm_cart, ierr)
                                                                                      17
CALL MPI_CART_GET(comm_cart, ndims, dims, periods, own_coords, ierr)
                                                                                      18
i = own_coords(1)
                                                                                      19
j = own_coords(2)
                                                                                      20
! Look up the ranks for the neighbors. Own process coordinates are (i,j).
                                                                                      21
! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
                                                                                      22
CALL MPI_CART_SHIFT(comm_cart, 0,1, neigh_rank(1), neigh_rank(2), ierr)
                                                                                      23
CALL MPI_CART_SHIFT(comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
                                                                                      24
! Initialize the grid functions and start the iteration
                                                                                      25
CALL init(u, f)
                                                                                      26
DO it=1,100
                                                                                      27
   CALL relax(u, f)
                                                                                      28
       Exchange data with neighbor processes
!
                                                                                      29
   CALL exchange(u, comm_cart, neigh_rank, num_neigh)
                                                                                      30
END DO
                                                                                      31
CALL output(u)
                                                                                      32
                                                                                      33
                                                                                      34
  Figure 8.4: Set-up of process structure for two-dimensional parallel Poisson solver.
                                                                                      35
                                                                                      36
                                                                                      37
                                                                                      38
                                                                                      39
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
                                                                                      45
                                                                                      46
```

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
       1
            REAL u(0:101,0:101)
       \mathbf{2}
            INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
       3
            REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
       4
            INTEGER ierr
       5
            sndbuf(1:100,1) = u(1,1:100)
       6
            sndbuf(1:100,2) = u(100,1:100)
       7
            sndbuf(1:100,3) = u(1:100, 1)
       8
            sndbuf(1:100,4) = u(1:100,100)
       9
            CALL MPI_NEIGHBOR_ALLTOALL(sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
       10
                                        comm_cart, ierr)
       11
            ! instead of
      12
#-153 13
           ! CALL MPI_IRECV(rcvbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(1), ierr)
           ! CALL MPI_ISEND(sndbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(2), ierr)
B.2.1: 1<sup>14</sup>
                Always pairing a receive from rank_source with a send to rank_dest
           .
       15
                of the same direction in MPI_CART_SHIFT!
           .
       16
           ! CALL MPI_IRECV(rcvbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(3), ierr)
       17
           ! CALL MPI_ISEND(sndbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(4), ierr)
       18
           ! CALL MPI_IRECV(rcvbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(5), ierr)
       19
           ! CALL MPI_ISEND(sndbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(6), ierr)
       20
           ! CALL MPI_IRECV(rcvbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(7), ierr)
       21
           ! CALL MPI_ISEND(sndbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(8), ierr)
       22
                Of course, one can first start all four IRECV and then all four ISEND,
           •
       23
           .
               Or vice versa, but both in the sequence shown above. Otherwise, the
       ^{24}
               matching would be wrong for 2 or only 1 processes in a direction.
           (!)
      25
            ! CALL MPI_WAITALL(2*num_neigh, rq, statuses, ierr)
       26
            u(0,1:100) = rcvbuf(1:100,1)
      27
            u(101,1:100) = rcvbuf(1:100,2)
      28
            u(1:100, 0) = rcvbuf(1:100,3)
      29
            u(1:100,101) = rcvbuf(1:100,4)
      30
            END
      ^{31}
      32
      33
            Figure 8.5: Communication routine with local data copying and sparse neighborhood all-
      34
            to-all.
      35
      36
      37
       38
       39
       40
      41
      42
      43
      44
       45
       46
       47
       48
```

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
                                                                                          1
IMPLICIT NONE
                                                                                          2
USE MPI
REAL u(0:101,0:101)
                                                                                          4
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                          5
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
                                                                                          6
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
INTEGER type_vec, ierr
                                                                                          9
! The following initialization need to be done only once
                                                                                          10
! before the first call of exchange.
                                                                                          11
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
                                                                                          12
CALL MPI_TYPE_VECTOR(100, 1, 102, MPI_REAL, type_vec, ierr)
                                                                                          13
CALL MPI_TYPE_COMMIT(type_vec, ierr)
                                                                                          14
sndtypes(1:2) = type_vec
sndcounts(1:2) = 1
                                                                                          15
sndtypes(3:4) = MPI_REAL
                                                                                          16
sndcounts(3:4) = 100
                                                                                          17
rcvtypes = sndtypes
                                                                                          18
rcvcounts = sndcounts
                                                                                          19
sdispls(1) = ( 1 + 1*102) * sizeofreal ! first element of u( 1
                                                                           1:100)
                                                                                          20
                     1*102) * sizeofreal ! first element of u(100
sdispls(2) = (100 +
                                                                           1:100
                                                                                         21
sdispls(3) = (1 +
                     1*102) * sizeofreal ! first element of u( 1:100,
                                                                                 )
                                                                           1
sdispls(4) = (1 + 100*102) * size of real ! first element of u(
                                                                  1:100,100
                                                                                 )
                                                                                         22
rdispls(1) = (0 +
                     1*102) * sizeofreal ! first element of u( 0
                                                                         , 1:100)
                                                                                         23
                      1*102) * sizeofreal ! first element of u(101
rdispls(2) = (101 +
                                                                           1:100)
                                                                                          24
rdispls(3) = ( 1 +
                      0*102) * sizeofreal ! first element of u( 1:100, 0
                                                                                 )
                                                                                          25
rdispls(4) = (1 + 101*102) * sizeofreal ! first element of u( 1:100,101
                                                                                 )
                                                                                          26
! the following communication has to be done in each call of exchange
                                                                                          27
CALL MPI_NEIGHBOR_ALLTOALLW(u, sndcounts, sdispls, sndtypes, &
                                                                                          28
                            u, rcvcounts, rdispls, rcvtypes, &
                                                                                          29
                             comm_cart, ierr)
! The following finalizing need to be done only once
                                                                                          30
! after the last call of exchange.
                                                                                          31
CALL MPI_TYPE_FREE(type_vec, ierr)
                                                                                          32
END
                                                                                          33
                                                                                         34
                                                                                          35
Figure 8.6: Communication routine with sparse neighborhood all-to-all-w and without local
                                                                                          36
data copying.
                                                                                          37
                                                                                          38
                                                                                          39
                                                                                          40
                                                                                          41
                                                                                          42
                                                                                          43
                                                                                          44
                                                                                          45
                                                                                          46
                                                                                          47
```

```
INTEGER ndims, num_neigh
#-25
         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)
     \mathbf{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
          1
     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
          1
     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 8.7: Two-dimensional parallel Poisson solver with persistent sparse neighborhood
```

<sup>48</sup> all-to-all-w and without local data copying.

## Chapter 9

# **MPI** Environmental Management

 $\frac{24}{25}$ 

 $41 \\ 42$ 

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

## 9.1 Implementation Information

### 9.1.1 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,

```
INTEGER :: MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION = 3)
PARAMETER (MPI_SUBVERSION = 1)
```

For runtime determination,

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

OUT	version	version number (integer)
OUT	subversion	subversion number (integer)

C binding
-----------

int MPI_Get_version(int	*version, in	t *subversion)	
Fortran 2008 binding			
MPI_Get_version(version,	subversion,	ierror)	

```
1
          INTEGER, INTENT(OUT) :: version, subversion
\mathbf{2}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     Fortran binding
4
     MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
5
          INTEGER VERSION, SUBVERSION, IERROR
6
7
          MPI_GET_VERSION can be called at any time in an MPI program. This function must
8
     always be thread-safe, as defined in Section 11.6. Valid (MPI_VERSION, MPI_SUBVERSION)
9
     pairs in this and previous versions of the MPI standard are (4,0), (3,1), (3,0), (2,2), (2,1),
10
     (2,0), and (1,2).
11
12
     MPI_GET_LIBRARY_VERSION(version, resultlen)
13
14
       OUT
                 version
                                              version number (string)
15
       OUT
                 resultlen
                                              Length (in printable characters) of the result
16
                                              returned in version (integer)
17
18
     C binding
19
     int MPI_Get_library_version(char *version, int *resultlen)
20
21
     Fortran 2008 binding
22
     MPI_Get_library_version(version, resultlen, ierror)
23
          CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
^{24}
          INTEGER, INTENT(OUT) :: resultlen
25
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
28
          CHARACTER*(*) VERSION
29
          INTEGER RESULTLEN, IERROR
30
^{31}
          This routine returns a string representing the version of the MPI library. The version
32
     argument is a character string for maximum flexibility.
33
34
           Advice to implementors. An implementation of MPI should return a different string
35
           for every change to its source code or build that could be visible to the user. (End of
36
           advice to implementors.)
37
38
          The argument version must represent storage that is
39
     MPI_MAX_LIBRARY_VERSION_STRING characters long. MPI_GET_LIBRARY_VERSION may
40
     write up to this many characters into version.
41
          The number of characters actually written is returned in the output argument, resultlen.
42
     In C, a null character is additionally stored at version[resultlen]. The value of resultlen cannot
43
     be larger than MPI_MAX_LIBRARY_VERSION_STRING - 1. In Fortran, version is padded on
44
     the right with blank characters. The value of resultlen cannot be larger than
45
     MPI_MAX_LIBRARY_VERSION_STRING.
46
          MPI_GET_LIBRARY_VERSION can be called at any time in an MPI program. This
47
     function must always be thread-safe, as defined in Section 11.6.
```

CHAPTER 9. MPI ENVIRONMENTAL MANAGEMENT

```
48
```

9.1.2 Environmental Inquiries	1
When using the World Model (Section 11.2), a set of attributes that describe the execution	2 3
environment is attached to the communicator MPI_COMM_WORLD when MPI is initialized.	4
The values of these attributes can be inquired by using the function	5
MPI_COMM_GET_ATTR described in Section 7.7 and in Section 19.3.7. It is erroneous to	6
delete these attributes, free their keys, or change their values.	7
The list of predefined attribute keys include	8
<b>MPI_TAG_UB</b> Upper bound for tag value.	9 10
<b>MPI_HOST</b> Host process rank, if such exists, MPI_PROC_NULL, otherwise.	11
<b>MPI_IO</b> rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same	12
communicator may return different values for this parameter.	13 14
<b>MPI_WTIME_IS_GLOBAL</b> Boolean variable that indicates whether clocks are synchronized.	15 16
When using the Sessions Model (Section 11.3), only the MPI_TAG_UB attribute is avail-	17
able.	18
Vendors may add implementation-specific parameters (such as node number, real mem-	19
ory size, virtual memory size, etc.)	20
These predefined attributes do not change value between MPI initialization (MPI_INIT)	21
and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.	22
Advice to users. Note that in the C binding, the value returned by these attributes	23
is a <i>pointer</i> to an int containing the requested value. ( <i>End of advice to users.</i> )	24 25
	26
The required parameter values are discussed in more detail below:	27
	28
Tag Values	29
Tag values range from 0 to the value returned for MPI_TAG_UB, inclusive. These values are	30
guaranteed to be unchanging during the execution of an MPI program. In addition, the tag	31
upper bound value must be at least 32767. An MPI implementation is free to make the	32
value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a valid value	33
for MPI_TAG_UB.	34
In the Sessions Model, the attribute MPI_TAG_UB is attached to all communicators	35 36
created by MPI_COMM_CREATE_FROM_GROUP and	37
MPI_INTERCOMM_CREATE_FROM_GROUPS, with the same value on all MPI processes	38
in the communicator. In the World Model, the attribute MPI_TAG_UB has the same value	39
on all processes of MPI_COMM_WORLD.	40
	41
Host Rank	42
The value returned for MPI_HOST gets the rank of the <i>HOST</i> process in the group associated	43
with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if	44
there is no host. MPI does not specify what it means for a process to be a <i>HOST</i> , nor does	45
it requires that a $HOST$ exists.	46
The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.	47

## Unofficial Draft for Comment Only

1	IO Rank		
2 3 4 5	I/O facilities. For Fortra	an, this means th ITE). For C, this	k of a processor that can provide language-standard nat all of the Fortran I/O operations are supported s means that all of the ISO C I/O operations are
6 7 8 9	If every process car will be returned. Othe then its rank will be re	provide languagerwise, if the cal turned. Otherwise	ge-standard I/O, then the value MPI_ANY_SOURCE lling process can provide language-standard I/O se, if some process can provide language-standard s will be returned. The same value need not be
10 11 12	,	s. If no process of	can provide language-standard $I/O$ , then the value
13 14 15		-	s not collective, and this attribute does <i>not</i> indicate input. ( <i>End of advice to users.</i> )
16 17	Clock Synchronization		
18 19 20 21 22 23 24 25 26 27 28 29 30 31	MPI_COMM_WORLD are synchronized if explicit the variation in time, as round-trip time for an before a send and at an be always higher than t The attribute MPI_ synchronized (however, attribute may be associ	e synchronized, ( effort has been ta s measured by ca MPI message of l other process jus he first one. .WTIME_IS_GLOB the attribute ke ated with commu	GLOBAL is 1 if clocks at all processes in 0 otherwise. A collection of clocks is considered aken to synchronize them. The expectation is that alls to MPI_WTIME, will be less then one half the length zero. If time is measured at a process just st after a matching receive, the second time should BAL need not be present when the clocks are not ey MPI_WTIME_IS_GLOBAL is always valid). This unicators other then MPI_COMM_WORLD. BAL has the same value on all processes of
32	Inquire Processor Name		
33			
34	MPI_GET_PROCESSOR	_NAME(name, re	esultlen)
35 36 37	OUT name	)	A unique specifier for the actual (as opposed to virtual) node.
38 39 40	OUT resultlen		Length (in printable characters) of the result returned in $name$
41 42 43	C binding int MPI_Get_processo	r_name(char *r	name, int *resultlen)
44 45 46 47 48	Fortran 2008 binding MPI_Get_processor_na CHARACTER(LEN=MF INTEGER, INTENT( INTEGER, OPTIONA	me(name, resul I_MAX_PROCESSC OUT) :: result	DR_NAME), INTENT(OUT) :: name tlen

## Unofficial Draft for Comment Only

## Fortran binding MPI\_GET\_PROCESSOR\_NAME(NAME, RESULTLEN, IERROR) CHARACTER\*(\*) NAME INTEGER RESULTLEN, IERROR

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI\_MAX\_PROCESSOR\_NAME characters long. MPI\_GET\_PROCESSOR\_NAME may write up to this many characters into name.

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

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

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

## 9.2 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 12.5.3.

MPI\_ALLOC\_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative	40
		$\operatorname{integer})$	41
IN	info	info argument (handle)	42
OUT	baseptr	pointer to beginning of memory segment allocated	43
001		Pointer to seguring of memory pegment anotated	44

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

### Fortran 2008 binding

Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4 5

6

7

8

9

10

11

12

13

14

15

16 17

18

19

20

21 22

23

 $^{24}$ 

25

26 27

28 29

30

 $^{31}$ 

32

33

34

35

36 37 38

39

45

46

```
1
     MPI_Alloc_mem(size, info, baseptr, ierror)
\mathbf{2}
          USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
3
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
4
          TYPE(MPI_Info), INTENT(IN) :: info
5
          TYPE(C_PTR), INTENT(OUT) :: baseptr
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     Fortran binding
8
     MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
9
          INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
10
          INTEGER INFO, IERROR
11
12
         If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
13
     be provided in the mpi module and should be provided in mpif.h through overloading,
14
     i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
15
     BASEPTR, but with a different specific procedure name:
16
17
     INTERFACE MPI_ALLOC_MEM
18
          SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
19
              IMPORT :: MPI_ADDRESS_KIND
              INTEGER INFO, IERROR
20
21
              INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
22
          END SUBROUTINE
23
          SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
^{24}
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
25
              IMPORT :: MPI_ADDRESS_KIND
26
              INTEGER :: INFO, IERROR
27
              INTEGER(KIND=MPI_ADDRESS_KIND) ::
                                                      SIZE
              TYPE(C_PTR) ::
                                BASEPTR
28
29
          END SUBROUTINE
30
     END INTERFACE
31
          The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The
32
     implied specific procedure names are described in Section 19.1.5.
33
          By default, the allocated memory shall be aligned to at least the alignment required
34
     for load/store accesses of any datatype corresponding to a predefined MPI datatype. The
35
     info argument may be used to specify a desired alternative minimum alignment in bytes for
36
     the allocated memory by setting the value of the key "mpi_minimum_memory_alignment" to an
37
     integral number equal to a power of two. An implementation may ignore values smaller than
38
     the default required alignment. The info argument can also be used to provide directives
39
     that control the desired location of the allocated memory. Such a directive does not affect
40
     the semantics of the call. The corresponding info values are implementation-dependent. A
41
     null directive value of info = MPI_INFO_NULL is always valid.
42
          The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
43
     to indicate it failed because memory is exhausted.
44
45
46
47
48
```

MPI_FRE	EE_MEM(base)	initial address of memory segment allocated by			
		MPI_ALLOC_MEM (choice)			
C bindi int MPI	ng _Free_mem(void *b	ase)			
MPI_Free TYPI	<pre>Fortran 2008 binding MPI_Free_mem(base, ierror)    TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: base    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
MPI_FREI <tyj< td=""><td>binding E_MEM(BASE, IERRO pe&gt; BASE(*) EGER IERROR</td><td>R)</td></tyj<>	binding E_MEM(BASE, IERRO pe> BASE(*) EGER IERROR	R)			
	function MPI_FREE an invalid base argu	<b>E_MEM</b> may return an error code of class MPI_ERR_BASE to ment.			
to MF less as	the bindings for the PI_Alloc_mem(, &l s level of indirection to facilitate type cas	dings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar e malloc and free C library calls: a call to base) should be paired with a call to MPI_Free_mem(base) (one ). Both arguments are declared to be of same type void* so sting. The Fortran binding is consistent with the C bindings: DC_MEM call returns in baseptr the TYPE(C_PTR) pointer or			

MPI\_FREE\_MEM is a choice argument, which passes (a reference to) the variable stored at that location. (End of rationale.) Advice to implementors. If MPI\_ALLOC\_MEM allocates special memory, then a design similar to the design of C malloc and free functions has to be used, in order

the (integer valued) address of the allocated memory. The base argument of

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. (End of advice to implementors.)

**Example 9.1** Example of use of MPI\_ALLOC\_MEM, in Fortran with TYPE(C\_PTR) pointers. We assume 4-byte REALs.

USE mpi\_f08 ! or USE mpi (not guaranteed with INCLUDE 'mpif.h') USE, INTRINSIC :: ISO\_C\_BINDING TYPE(C\_PTR) :: p REAL, DIMENSION(:,:), POINTER :: a ! no memory is allocated INTEGER, DIMENSION(2) :: shape INTEGER(KIND=MPI\_ADDRESS\_KIND) :: size shape = (/100, 100/)

 $\mathbf{2}$ 

 $^{24}$ 

```
1
       size = 4 * \text{shape}(1) * \text{shape}(2)
                                                             ! assuming 4 bytes per REAL
2
       CALL MPI_Alloc_mem(size, MPI_INFO_NULL, p, ierr) ! memory is allocated and
3
       CALL C_F_POINTER(p, a, shape) ! intrinsic
                                                             ! now accessible via a(i,j)
4
                                          ! in ISO_C_BINDING
        . . .
5
       a(3,5) = 2.71
6
        . . .
7
       CALL MPI_Free_mem(a, ierr)
                                                             ! memory is freed
8
9
     Example 9.2 Example of use of MPI_ALLOC_MEM, in Fortran with non-standard Cray-
10
     pointers. We assume 4-byte REALS, and assume that these pointers are address-sized.
11
12
       REAL A
13
                                     ! no memory is allocated
       POINTER (P, A(100,100))
14
       INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
15
       SIZE = 4*100*100
16
       CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
17
        ! memory is allocated
18
        . . .
19
       A(3,5) = 2.71
20
        . . .
21
       CALL MPI_FREE_MEM(A, IERR) ! memory is freed
22
         This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this
23
     code or need a special option, e.g., the GNU gFortran compiler needs -fcray-pointer.
24
25
           Advice to implementors. Some compilers map Cray-pointers to address-sized integers,
26
           some to TYPE(C_PTR) pointers (e.g., Cray Fortran, version 7.3.3). From the user's
27
           viewpoint, this mapping is irrelevant because Examples 9.2 should work correctly
28
           with an MPI-3.0 (or later) library if Cray-pointers are available. (End of advice to
29
           implementors.)
30
31
32
     Example 9.3 Same example, in C.
33
       float (* f)[100][100];
34
       /* no memory is allocated */
35
       MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
36
       /* memory allocated */
37
        . . .
38
        (*f)[5][3] = 2.71;
39
        . . .
40
       MPI_Free_mem(f);
41
42
43
            Error Handling
     9.3
44
```

An MPI implementation may be unable or choose not to handle some failures that occur
 during MPI calls. These can include failures that generate exceptions or traps, such as
 floating point errors or access violations. The set of failures that are handled by MPI is
 implementation-dependent. Each such failure causes an error to be raised.

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

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

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 four types of objects: communicators, windows, files, and sessions. The specified error handling routine will be used for any error that occurs during a call to MPI for the respective object. MPI calls that are not related to any MPI objects are considered to be attached to the communicator MPI\_COMM\_SELF. When MPI\_COMM\_SELF is not initialized (i.e., before MPI\_INIT / MPI\_INIT\_THREAD or after MPI\_FINALIZE) the error raises the initial error handler (set during the launch operation, see 11.8.4). 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 MPI 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 file, it is similar to calling MPI\_ABORT using a communicator containing the group of MPI processes associated with the window or file, respectively. If the error handler is invoked on a session, the operation aborts only the local MPI process. In all cases, 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.

Unless otherwise requested, the error handler MPI\_ERRORS\_ARE\_FATAL is set as the 36 default initial error handler and associated with predefined communicators. Thus, if the 37 user chooses not to control error handling, every error that MPI handles is treated as fatal. 38 Since (almost) all MPI calls return an error code, a user may choose to handle errors in its 39 main code, by testing the return code of MPI calls and executing a suitable recovery code 40 when the call was not successful. In this case, the error handler MPI\_ERRORS\_RETURN will 41 be used. Usually it is more convenient and more efficient not to test for errors after each 42MPI call, and have such error handled by a non-trivial MPI error handler. Note that unlike 43 predefined communicators, windows and files do not inherit from the initial error handler, 44as defined in Sections 12.6 and 14.7 respectively. 45

When an error is raised, MPI will provide the user information about that error using an error code. 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 48

#### Unofficial Draft for Comment Only

or the user terminates the application. The user can make the determination of whether or not to attempt to continue when handling 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, window, and session arguments. In Fortran there are four user routines.

<sup>19</sup> An error handler object is created by a call to MPI\_XXX\_CREATE\_ERRHANDLER, <sup>20</sup> where XXX is, respectively, COMM, WIN, FILE, or SESSION.

An error handler is attached to a communicator, window, file, or session 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. An error handler can also be attached to a session using the errorhandler argument to MPI\_SESSION\_INIT. The predefined error handlers MPI\_ERRORS\_RETURN and MPI\_ERRORS\_ARE\_FATAL can be attached to communicators, windows, files, or sessions.

The error handler currently associated with a communicator, window, file, or session 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.

<sup>31</sup> MPI\_XXX\_GET\_ERRHANDLER behave as if a new error handler object is created. That <sup>32</sup> is, once the error handler is no longer needed, MPI\_ERRHANDLER\_FREE should be called <sup>33</sup> with the error handler returned from MPI\_XXX\_GET\_ERRHANDLER to mark the error <sup>34</sup> handler for deallocation. This provides behavior similar to that of MPI\_COMM\_GROUP <sup>35</sup> 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 attached to an object of the wrong type with a call to MPI\_YYY\_SET\_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

- The syntax for these calls is given below.
- 43 44

36

37

38

39

40

41 42

- 45
- 46
- 47
- 48

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

9.3. ERROR HANDLING	453
9.3.1 Error Handlers for Communicators	1
	2
	3
MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)	4
IN comm_errhandler_fn user defined error handling procedure (function	
OUTerrhandlerMPI error handler (handle)	7
	8
C binding	9
<pre>int MPI_Comm_create_errhandler(</pre>	10
MPI_Comm_errhandler_function *comm_errhandler_fn,	11 12
MPI_Errhandler *errhandler)	12
Fortran 2008 binding	14
<pre>MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)</pre>	15
PROCEDURE(MPI_Comm_errhandler_function), INTENT(IN) ::	16
comm_errhandler_fn	17
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
	19 20
Fortran binding	20
MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL COMM_ERRHANDLER_FN	22
INTEGER ERRHANDLER, IERROR	23
	24
Creates an error handler that can be attached to communicators.	25
The user routine should be, in C, a function of type MPI_Comm_errhandler_function, is defined as	
typedef void MPI_Comm_errhandler_function(MPI_Comm *comm, int *error_comm	ode, 27
);	29 29
	20
The first argument is the communicator in use. The second is the error code 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 of	20
the error handler to be invoked. The remaining arguments are "varargs" arguments	
number and meaning is implementation-dependent. An implementation should clearly	
ument these arguments. Addresses are used so that the handler may be written in Fo	26
With the Fortran mpi_f08 module, the user routine comm_errhandler_fn should be	of the $37$
	38
ABSTRACT INTERFACE SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)	39
TYPE(MPI_Comm) :: comm	40
INTEGER :: error_code	41
	42 - D - EN
With the Fortran mpi module and mpif.h, the user routine COMM_ERRHANDLE should be of the form:	
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)	44 45
INTEGER COMM, ERROR_CODE	45 46
· _	47

	454		CHAPTER	9. MPI ENVIRONMENTAL MANAGEMENT
1 2 3 4	S	standard hook for provid	ing additions	t list is provided because it provides an ISO- al information to the error handler; without this uments. ( <i>End of rationale.</i> )
5 6 7 8 9	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.)			
10				
.2	MPI_C	COMM_SET_ERRHANDL	.ER(comm, e	rrhandler)
4	INOU	JT comm	С	ommunicator (handle)
15 16 17	IN	errhandler	n	ew error handler for communicator (handle)
18	C bin int M	0	er(MPI_Comm	n comm, MPI_Errhandler errhandler)
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34	<pre>Fortran 2008 binding MPI_Comm_set_errhandler(comm, errhandler, ierror)    TYPE(MPI_Comm), INTENT(IN) :: comm    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler    INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)    INTEGER COMM, ERRHANDLER, IERROR    Attaches a new error handler to a communicator. The error handler must be either    a predefined error handler, or an error handler created by a call to    MPI_COMM_CREATE_ERRHANDLER. MPI_COMM_GET_ERRHANDLER(comm, errhandler)</pre>			
35	IN	comm		ommunicator (handle)
36 37 38	OUT	errhandler	e	rror handler currently associated with ommunicator (handle)
89 10 11	C bin int M	<b>S</b>	er(MPI_Comm	n comm, MPI_Errhandler *errhandler)
42 43 44 45 46 47	MPI_C T T I	an 2008 binding omm_get_errhandler(co YPE(MPI_Comm), INTEN YPE(MPI_Errhandler), NTEGER, OPTIONAL, INT an binding	F(IN) :: cc INTENT(OUT	omm ") :: errhandler
18	rortra	an binding		

MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)			
INTEGER COMM, ERRHANDLER, IERROR			
Retriev	es the error handler current!	ly associated with a communicator.	3 4
For exa	mple, a library function may	register at its entry point the current error handler	5
for a comm	unicator, set its own private	e error handler for this communicator, and restore	6
before exitin	ng the previous error handles	r.	7
			8
9.3.2 Error Handlers for Windows			
			9 10
			11
MPI_WIN_C	CREATE_ERRHANDLER(win	_errhandler_fn, errhandler)	12
IN	win_errhandler_fn	user defined error handling procedure (function)	$13 \\ 14$
OUT	errhandler	MPI error handler (handle)	14
			16
C binding			17
	n_create_errhandler(		18
		unction *win_errhandler_fn,	19
	MPI_Errhandler *errh		20
			21
	008 binding		22
		handler_fn, errhandler, ierror)	23
		<pre>function), INTENT(IN) :: win_errhandler_fn</pre>	24
	PI_Errhandler), INTENT(		25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			26
Fortran binding			27
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)			28
EXTERNAL WIN_ERRHANDLER_FN			29
			30
C I			31
		be attached to a window object. The user routine	32
		Win_errhandler_function which is defined as	33
typedei vo		unction(MPI_Win *win, int *error_code,	34
	);		35
The first	st argument is the window i	n use, the second is the error code to be returned.	36
		ser routine win_errhandler_fn should be of the form:	37
ABSTRACT I	NTERFACE		38
SUBROUTI	SUBROUTINE MPI_Win_errhandler_function(win, error_code) 3		
TYPE(M	TYPE(MPI_Win) :: win		
INTEGE	R :: error_code		41
With the Fe	ertran mni madula and mnif	h the user routing WIN EDDHANDLED EN should	42
be of the for		h, the user routine $WIN\_ERRHANDLER\_FN$ should	43
		N(WIN FRROR CODE)	44
	SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)		
INTEGER WIN, ERROR_CODE			46
			47
			48

```
1
     MPI_WIN_SET_ERRHANDLER(win, errhandler)
\mathbf{2}
       INOUT
                                            window object (handle)
                win
3
       IN
                errhandler
                                            new error handler for window (handle)
4
5
6
     C binding
     int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
7
8
     Fortran 2008 binding
9
     MPI_Win_set_errhandler(win, errhandler, ierror)
10
          TYPE(MPI_Win), INTENT(IN) :: win
11
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
16
          INTEGER WIN, ERRHANDLER, IERROR
17
          Attaches a new error handler to a window. The error handler must be either a pre-
18
     defined error handler, or an error handler created by a call to
19
     MPI_WIN_CREATE_ERRHANDLER.
20
21
22
     MPI_WIN_GET_ERRHANDLER(win, errhandler)
23
       IN
                                            window object (handle)
                 win
24
       OUT
                errhandler
                                            error handler currently associated with window
25
26
                                            (handle)
27
28
     C binding
29
     int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
30
     Fortran 2008 binding
^{31}
     MPI_Win_get_errhandler(win, errhandler, ierror)
32
          TYPE(MPI_Win), INTENT(IN) :: win
33
          TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
34
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     Fortran binding
37
     MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
38
          INTEGER WIN, ERRHANDLER, IERROR
39
         Retrieves the error handler currently associated with a window.
40
41
42
43
44
45
46
47
48
```

9.3.	ERF	ROR HANDLING	457	
9.3.3	3 Er	ror Handlers for File	25	1
				2
				3
MPI.	_FILE	_CREATE_ERRHAN	DLER(file_errhandler_fn, errhandler)	4 5
IN		file_errhandler_fn	user defined error handling procedure (function)	6
οι	JT	errhandler	MPI error handler (handle)	7
				8
Сb	indin	g		9
int	MPI_	File_create_errha	ndler(	10
			chandler_function *file_errhandler_fn,	11
		MPI_Errhand]	ler *errhandler)	12 13
Fort	ran	2008 binding		13
MPI_	File	_create_errhandle	r(file_errhandler_fn, errhandler, ierror)	15
	PROC		rhandler_function), INTENT(IN) ::	16
		file_errhan		17
			INTENT(OUT) :: errhandler	18
	TNIC	GER, UPIIUNAL, IN	TENT(OUT) :: ierror	19
		binding		20 21
			R(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)	21
		RNAL FILE_ERRHAND		23
		GER ERRHANDLER, I		24
			that can be attached to a file object. The user routine should	25
			PI_File_errhandler_function, which is defined as	26
туре	eder	void MPI_File_err );	handler_function(MPI_File *file, int *error_code,	27
				28 29
			file in use, the second is the error code to be returned.	29 30
			dule, the user routine file_errhandler_fn should be of the form:	31
		INTERFACE	handler function(file error code)	32
50	<pre>SUBROUTINE MPI_File_errhandler_function(file, error_code) TYPE(MPI_File) :: file</pre>			
		GER :: error_code		34
<b>W</b> ;+1	a tha	Pontnon mai modulo	and maif h the user routing EILE EDDHANDLED EN should	35
	f the	-	and mpif.h, the user routine FILE_ERRHANDLER_FN should	36 37
			R_FUNCTION(FILE, ERROR_CODE)	38
		GER FILE, ERROR_C		39
				40
				41
MPI.	_FILE	_SET_ERRHANDLE	R(file, errhandler)	42
IN	Ουτ	file	file (handle)	43
IN		errhandler	new error handler for file (handle)	44
				45 46
Сb	indin	lg		40
		•	er(MPI_File file, MPI_Errhandler errhandler)	48

Unofficial Draft for Comment Only

1 Fortran 2008 binding  $\mathbf{2}$ MPI\_File\_set\_errhandler(file, errhandler, ierror) 3 TYPE(MPI\_File), INTENT(IN) :: file 4 TYPE(MPI\_Errhandler), INTENT(IN) :: errhandler 5INTEGER, OPTIONAL, INTENT(OUT) :: ierror 6 Fortran binding 7 MPI\_FILE\_SET\_ERRHANDLER(FILE, ERRHANDLER, IERROR) 8 INTEGER FILE, ERRHANDLER, IERROR 9 10 Attaches a new error handler to a file. The error handler must be either a predefined 11 error handler, or an error handler created by a call to MPI\_FILE\_CREATE\_ERRHANDLER. 1213 MPI\_FILE\_GET\_ERRHANDLER(file, errhandler) 1415IN file file (handle) 16OUT errhandler error handler currently associated with file (handle) 1718 C binding 19 int MPI\_File\_get\_errhandler(MPI\_File file, MPI\_Errhandler \*errhandler) 2021Fortran 2008 binding 22 MPI\_File\_get\_errhandler(file, errhandler, ierror) 23TYPE(MPI\_File), INTENT(IN) :: file  $^{24}$ TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26Fortran binding 27MPI\_FILE\_GET\_ERRHANDLER(FILE, ERRHANDLER, IERROR) 28INTEGER FILE, ERRHANDLER, IERROR 29 30 Retrieves the error handler currently associated with a file.  $^{31}$ 32 9.3.4 Error Handlers for Sessions 33 3435 MPI\_SESSION\_CREATE\_ERRHANDLER(session\_errhandler\_fn, errhandler) 36 37 IN session\_errhandler\_fn user defined error handling procedure (function) 38 OUT errhandler MPI error handler (handle) 39 40 C binding 41 int MPI\_Session\_create\_errhandler( 42MPI\_Session\_errhandler\_function \*session\_errhandler\_fn, 43 MPI\_Errhandler \*errhandler) 4445Fortran 2008 binding 46MPI\_Session\_create\_errhandler(session\_errhandler\_fn, errhandler, ierror) 47PROCEDURE(MPI\_Session\_errhandler\_function), INTENT(IN) :: 48 session\_errhandler\_fn

TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
Fortran binding	4
MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)	5
EXTERNAL SESSION_ERRHANDLER_FN	6
INTEGER ERRHANDLER, IERROR	7
Creates an error handler that can be attached to a session object. In C, the	8
session_errhandler_fn argument should be a function of type MPI_Session_errhandler_function,	9
which is defined as	10
typedef void MPI_Session_errhandler_function(MPI_Session *session,	11
<pre>int *error_code,);</pre>	12
The first comment is the section in one the second is the second state he actumed	13
The first argument is the session in use, the second is the error code to be returned. With the Fortron main 508 module, the cossion errhandler in argument should be of the	14
With the Fortran mpi_f08 module, the session_errhandler_fn argument should be of the form:	15
ABSTRACT INTERFACE	16
SUBROUTINE MPI_Session_errhandler_function(session, error_code)	17
TYPE(MPI_Session) :: session	18
INTEGER :: error_code	19 20
	20 21
With the Fortran mpi module and mpif.h, the SESSION_ERRHANDLER_FN argument	21
should be of the form:	23
SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE) INTEGER SESSION, ERROR_CODE	24
INTEGER SESSION, ERROR_CODE	
	25
	25 26
MPL SESSION SET ERRHANDLER(session errhandler)	
MPI_SESSION_SET_ERRHANDLER(session, errhandler)	26
MPI_SESSION_SET_ERRHANDLER(session, errhandler) INOUT session (handle)	26 27
	26 27 28
INOUT session (handle)	26 27 28 29 30 31
INOUT session (handle)	26 27 28 29 30 31 32
INOUTsession(handle)INerrhandlernew error handler for session (handle)	26 27 28 29 30 31 32 33
INOUT     session     (handle)       IN     errhandler     new error handler for session (handle)       C binding	26 27 28 29 30 31 32 33 34
<pre>INOUT session (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35
<pre>INOUT session (handle) IN errhandler C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36
<pre>INOUT session (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35
<pre>INOUT session (handle) IN errhandler C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37
<pre>INOUT session (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39
<pre>INOUT session (handle) IN errhandler (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler new error handler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler errhandler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
<pre>INOUT session (handle) IN errhandler (mode) IN errhandler errhandler (mode) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
<pre>INOUT session (handle) IN errhandler (handle) IN errhandler errhandler for session (handle) C binding int MPI_Session_set_errhandler(MPI_Session session,</pre>	26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45

1 MPI\_SESSION\_GET\_ERRHANDLER(session, errhandler) 2 IN session (handle) 3 OUT errhandler error handler currently associated with session 4 (handle) 56 C binding 7 8 int MPI\_Session\_get\_errhandler(MPI\_Session session, 9 MPI\_Errhandler \*errhandler) 10 Fortran 2008 binding 11 MPI\_Session\_get\_errhandler(session, errhandler, ierror) 12TYPE(MPI\_Session), INTENT(IN) :: session 13 TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516Fortran binding 17 MPI\_SESSION\_GET\_ERRHANDLER(SESSION, ERRHANDLER, IERROR) 18 INTEGER SESSION, ERRHANDLER, IERROR 19Retrieves the error handler currently associated with a session. 20219.3.5 Freeing Errorhandlers and Retrieving Error Strings 22 2324MPI\_ERRHANDLER\_FREE(errhandler) 2526INOUT errhandler MPI error handler (handle) 2728C binding 29 int MPI\_Errhandler\_free(MPI\_Errhandler \*errhandler) 30  $^{31}$ Fortran 2008 binding MPI\_Errhandler\_free(errhandler, ierror) 32 33 TYPE(MPI\_Errhandler), INTENT(INOUT) :: errhandler 34INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35Fortran binding 36 MPI\_ERRHANDLER\_FREE(ERRHANDLER, IERROR) 37 INTEGER ERRHANDLER, IERROR 38 39 Marks the error handler associated with errhandler for deallocation and sets errhandler 40to MPI\_ERRHANDLER\_NULL. The error handler will be deallocated after all the objects  $^{41}$ associated with it (communicator, window, or file) have been deallocated. 4243 44 4546 47 48

MPI_ERROR_STRING(errorcode, string, resultlen)				
IN	errorcode	Error code returned by an MPI routine	2	
OUT	string	Text that corresponds to the errorcode	3	
	0	-	4	
OUT	resultlen	Length (in printable characters) of the result	5 6	
		returned in string	7	
<b>a</b> 1 1 11			8	
C bindin	•		9	
int MPI_	Error_string(int	errorcode, char *string, int *resultlen)	10	
Fortran	2008 binding		11	
MPI_Erro	r_string(errorcod	e, string, resultlen, ierror)	12	
INTE	GER, INTENT(IN) :	: errorcode	13	
		_ERROR_STRING), INTENT(OUT) :: string	14	
	GER, INTENT(OUT)		15	
INTE	GER, OPTIONAL, IN	TENT(OUT) :: ierror	16	
Fortran	Fortran binding			
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)				
	GER ERRORCODE, RE		19 20	
CHAR	CHARACTER*(*) STRING			
Returns the error string associated with an error code or class. The argument string			21 22	
must represent storage that is at least MPI_MAX_ERROR_STRING characters long.				
The number of characters actually written is returned in the output argument, resultlen.				
	This function must always be thread-safe, as defined in Section 11.6. It is one of the			
	few routines that may be called before MPI is initialized or after MPI is finalized.			
Rate	Rationale. The form of this function was chosen to make the Fortran and C bindings			
similar. A version that returns a pointer to a string has two difficulties. First, the				
return string must be statically allocated and different for each error message (allowing				
the pointers returned by successive calls to MPI_ERROR_STRING to point to the				
correct message). Second, in Fortran, a function declared as returning CHARACTER*(*)				
can not be referenced in, for example, a PRINT statement. (End of rationale.)				
3				

#### 9.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI\_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI\_ERROR\_STRING).

All MPI function calls shall return MPI\_SUCCESS if and only if the specification of that function has been fulfilled at the point of return. For multiple completion functions, if the function returns MPI\_ERR\_IN\_STATUS, the error code in each status object shall be set to MPI\_SUCCESS if and only if the specification of the operation represented by the corresponding MPI\_Request has been fulfilled at the point of return.

When an operation raises an error, it may not satisfy its specification (for example, a synchronizing operation may not have synchronized) and the content of the output buffers, targeted memory, or output parameters is undefined. However, a valid error code shall

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

 $^{41}$ 

```
1
      always be set when an operation raises an error, whether in the return value, error field in
\mathbf{2}
      the status object, or element in an array of error codes.
3
          To make it possible for an application to interpret an error code, the routine
4
      MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes,
\mathbf{5}
      called error classes. Valid error classes are shown in Table 9.1 and Table 9.2.
6
          The error classes are a subset of the error codes: an MPI function may return an error
7
      class number; and the function MPI_ERROR_STRING can be used to compute the error
8
      string associated with an error class. The values defined for MPI error classes are valid MPI
9
      error codes.
10
          The error codes satisfy,
11
                      0 = MPI_SUCCESS < MPI_ERR_... < MPI_ERR_LASTCODE.
12
13
           Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that
14
           MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.
15
16
           Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the sepa-
17
           ration of error classes and error codes allows us to define the error classes this way.
18
           Having a known LASTCODE is often a nice sanity check as well. (End of rationale.)
19
20
21
      MPI_ERROR_CLASS(errorcode, errorclass)
22
23
       IN
                  errorcode
                                               Error code returned by an MPI routine
^{24}
        OUT
                  errorclass
                                               Error class associated with errorcode
25
26
      C binding
27
      int MPI_Error_class(int errorcode, int *errorclass)
28
29
     Fortran 2008 binding
30
      MPI_Error_class(errorcode, errorclass, ierror)
^{31}
          INTEGER, INTENT(IN) :: errorcode
32
          INTEGER, INTENT(OUT) :: errorclass
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
      Fortran binding
35
      MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
36
          INTEGER ERRORCODE, ERRORCLASS, IERROR
37
38
          The function MPI_ERROR_CLASS maps each standard error code (error class) onto
39
      itself.
40
          This function must always be thread-safe, as defined in Section 11.6. It is one of the
41
      few routines that may be called before MPI is initialized or after MPI is finalized.
42
43
      9.5
            Error Classes, Error Codes, and Error Handlers
44
45
```

Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on MPI, see Chapter 14. For this purpose, functions are needed to:

## Unofficial Draft for Comment Only

MPI_SUCCESS	No error	1
	Permission denied	2
MPI_ERR_AMODE	Error related to the <b>amode</b> passed to	3
	MPI_FILE_OPEN	4
MPI_ERR_ARG	Invalid argument of some other kind	5
MPI_ERR_ASSERT	Invalid assert argument	6
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	7
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM	8
MPI_ERR_BUFFER	Invalid buffer pointer	9
MPI_ERR_COMM	Invalid communicator	10
MPI_ERR_CONVERSION	An error occurred in a user supplied data	11
	conversion function.	12
MPI_ERR_COUNT	Invalid count argument	13
MPI_ERR_DIMS	Invalid dimension argument	14
MPI_ERR_DISP	Invalid disp argument	15
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	16
	tered because a data representation identi-	17
	fier that was already defined was passed to	18
	MPI_REGISTER_DATAREP	19
MPI_ERR_FILE	Invalid file handle	20
MPI_ERR_FILE_EXISTS	File exists	21
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	22
	the file is currently open by some process Invalid group	23 24
MPI_ERR_GROUP MPI_ERR_INFO	Invalid info argument	24 25
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY	26
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE	27
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL	28
MPI_ERR_IN_STATUS	Error code is in status	29
MPI_ERR_INTERN	Internal MPI (implementation) error	30
MPI_ERR_IO	Other I/O error	31
MPI_ERR_KEYVAL	Invalid keyval has been passed	32
MPI_ERR_LOCKTYPE	Invalid locktype argument	33
MPI_ERR_NAME	Invalid service name passed to	34
	MPI_LOOKUP_NAME	35
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory	36
	is exhausted	37
MPI_ERR_NO_SPACE	Not enough space	38
MPI_ERR_NO_SUCH_FILE	File does not exist	39
MPI_ERR_NOT_SAME	Collective argument not identical on all	40
	processes, or collective routines called in	41
	a different order by different processes	42
		43
		44
Table 9	.1: Error classes (Part 1)	45
		46

Unofficial Draft for Comment Only

1	MPI_ERR_OP	Invalid operation
2	MPI_ERR_OTHER	Known error not in this list
3	MPI_ERR_PENDING	Pending request
4	MPI_ERR_PORT	Invalid port name passed to
5		MPI_COMM_CONNECT
6	MPI_ERR_PROC_ABORTED	Operation failed because a peer process has
7		aborted
8	MPI_ERR_QUOTA	Quota exceeded
9	MPI_ERR_RANK	Invalid rank
10	MPI_ERR_READ_ONLY	Read-only file or file system
11	MPI_ERR_REQUEST	Invalid request (handle)
12	MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because
13		of resource exhaustion)
		,
14	MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
15	MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the
16		called function
17	MPI_ERR_RMA_RANGE	Target memory is not part of the win-
18		dow (in the case of a window created
19		with MPI_WIN_CREATE_DYNAMIC, tar-
20		get memory is not attached)
21	MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some pro-
22		cess in the group of the specified commu-
23		nicator cannot expose shared memory)
24	MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls
25	MPI_ERR_ROOT	Invalid root
26	MPI_ERR_SERVICE	Invalid service name passed to
27		MPI_UNPUBLISH_NAME
28	MPI_ERR_SESSION	Invalid session argument
29	MPI_ERR_SIZE	Invalid size argument
30	MPI_ERR_SPAWN	Error in spawning processes
31	MPI_ERR_TAG	Invalid tag argument
32	MPI_ERR_TOPOLOGY	Invalid topology
33	MPI_ERR_TRUNCATE	Message truncated on receive
34	MPI_ERR_TYPE	Invalid datatype argument
35	MPI_ERR_UNKNOWN	Unknown error
36	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
37		MPI_FILE_SET_VIEW
38	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
39		a file which supports sequential access only
40	MPI_ERR_VALUE_TOO_LARGE	Value is too large to store
41	MPI_ERR_WIN	Invalid win argument
42	MPI_ERR_LASTCODE	Last error code
43		
44		
45	Table 9.2: Err	cor classes (Part 2)
46		
47		
48		

- 2. associate error codes with this error class, so that MPI\_ERROR\_CLASS works.
- 3. associate strings with these error codes, so that  $\mathsf{MPI\_ERROR\_STRING}$  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.

MPI_ADD_ERROR_CLASS(errorclass)		12
OUT errorclass	value for the new error class (integer)	13 14
		14
C binding		16
<pre>int MPI_Add_error_class(int *er</pre>	rorclass)	17
Fortran 2008 binding		18 19
MPI_Add_error_class(errorclass,		20
INTEGER, INTENT(OUT) :: err INTEGER, OPTIONAL, INTENT(C		21
		22
Fortran binding		23
MPI_ADD_ERROR_CLASS(ERRORCLASS, INTEGER ERRORCLASS, IERROR	IERRUR)	24
INTEGER ERRORCLASS, IERROR		25
Creates a new error class and re	turns the value for it.	26 27
Rationale. To avoid conflicts	with existing error codes and classes, the value is set	28
	by the user. ( <i>End of rationale.</i> )	29
		30
	MPI_ADD_ERROR_CLASS is local, the same errorclass esses that make this call. Thus, it is not safe to assume	31
	a set of processes at the same time will yield the same	32
	Getting the "same" error on multiple processes may	33 34
-	or code to be generated. (End of advice to users.)	35
		36
	E is a constant value and is not affected by new user- ad, a predefined attribute key MPI_LASTUSEDCODE is	37
	The attribute value corresponding to this key is the	38
	g the user-defined ones. This is a local value and may	39
· · · · · · · · · · · · · · · · · · ·	ne value returned by this key is always greater than or	40
equal to $MPI\_ERR\_LASTCODE$ .		41 42
Advice to years. The value ret	urned by the key MPI_LASTUSEDCODE will not change	42
	co explicitly add an error class/code. In a multithreaded	44
	we extra care in assuming this value has not changed.	45
	or classes are not necessarily dense. A user may not	46
assume that each error class be	elow MPI_LASTUSEDCODE is valid. (End of advice to	47

#### Unofficial Draft for Comment Only

users.)

 $^{48}$ 

1 MPI\_ADD\_ERROR\_CODE(errorclass, errorcode) 2 IN errorclass error class (integer) 3 OUT errorcode new error code to be associated with errorclass 4 (integer) 56 C binding 7 8 int MPI\_Add\_error\_code(int errorclass, int \*errorcode) 9 Fortran 2008 binding 10 MPI\_Add\_error\_code(errorclass, errorcode, ierror) 11 INTEGER, INTENT(IN) :: errorclass 12INTEGER, INTENT(OUT) :: errorcode 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415Fortran binding 16MPI\_ADD\_ERROR\_CODE(ERRORCLASS, ERRORCODE, IERROR) 17 INTEGER ERRORCLASS, ERRORCODE, IERROR 18 Creates new error code associated with errorclass and returns its value in errorcode. 19 20Rationale. To avoid conflicts with existing error codes and classes, the value of the 21new error code is set by the implementation and not by the user. (End of rationale.) 22 23 $^{24}$ 25MPI\_ADD\_ERROR\_STRING(errorcode, string) 26IN error code or class (integer) errorcode 27IN string text corresponding to errorcode (string) 2829 C binding 30 int MPI\_Add\_error\_string(int errorcode, const char \*string)  $^{31}$ 32 Fortran 2008 binding 33 MPI\_Add\_error\_string(errorcode, string, ierror) 34 INTEGER, INTENT(IN) :: errorcode 35 CHARACTER(LEN=\*), INTENT(IN) :: string 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 Fortran binding 38 MPI\_ADD\_ERROR\_STRING(ERRORCODE, STRING, IERROR) 39 INTEGER ERRORCODE, IERROR 40 41 CHARACTER\*(\*) STRING 42Associates an error string with an error code or class. The string must be no more 43 than MPI\_MAX\_ERROR\_STRING characters long. The length of the string is as defined in the 44 calling language. The length of the string does not include the null terminator in C. Trailing 45 blanks will be stripped in Fortran. Calling MPI\_ADD\_ERROR\_STRING for an errorcode that 46 already has a string will replace the old string with the new string. It is erroneous to call 47MPI\_ADD\_ERROR\_STRING for an error code or class with a value < MPI\_ERR\_LASTCODE.

If MPI.	_ERROR_STRING is called w	hen no string has been set, it will return a empty	1
string (all s	paces in Fortran, "" in C).		2
Section	9.3 describes the methods	for creating and associating error handlers with	3
communica	tors, files, windows, and sessi	ons.	4
			5
	/_CALL_ERRHANDLER(com	m errorcode)	6
	,	,	7 8
IN	comm	communicator with error handler (handle)	9
IN	errorcode	error code (integer)	10
			11
C binding			12
int MPI_Co	omm_call_errhandler(MPI_0	Comm comm, int errorcode)	13
Fortran 20	008 binding		14
	call_errhandler(comm, er	corcode, ierror)	15
	<pre>4PI_Comm), INTENT(IN) ::</pre>		16
	ER, INTENT(IN) :: errorco		17
INTEGH	ER, OPTIONAL, INTENT(OUT)	) :: ierror	18
Fortran bi	inding		19
	CALL_ERRHANDLER(COMM, ERI		20
	ER COMM, ERRORCODE, IERR		21
			22
		dler assigned to the communicator with the error	23 24
		PI_SUCCESS in C and the same value in IERROR if	24 25
	-	(assuming the process is not aborted and the error	26
handler ret	urns).		27
			28
MPI_WIN_	CALL_ERRHANDLER(win, err	rorcode)	29
IN	win	window with error handler (handle)	30
			31
IN	errorcode	error code (integer)	32
			33
C binding			34
int MPI_WI	in_call_errhandler(MPI_W	in win, int errorcode)	35
Fortran 20	008 binding		36
MPI_Win_ca	all_errhandler(win, error	ccode, ierror)	37 38
	<pre>MPI_Win), INTENT(IN) :: </pre>		39
	ER, INTENT(IN) :: errorco		40
INTEGE	ER, OPTIONAL, INTENT(OUT)	) :: ierror	41
Fortran bi	inding		42
	ALL_ERRHANDLER(WIN, ERRO	RCODE, IERROR)	43
	ER WIN, ERRORCODE, IERROF		44
			45
		adder assigned to the window with the error code	46
supplied. 1	ms function returns MPI_SU	CCESS in C and the same value in IERROR if the	47

1 2 3		handler was succes ler returns).	ssfully called (assuming the process is not aborted and the error
4 5 6 7		Advice to users. MPI_ERRORS_ARE_ advice to users.)	In contrast to communicators, the error handler $FATAL$ is associated with a window when it is created. (End of
8 9	MPI	FILE CALL ERRHA	NDLER(fh, errorcode)
10	IN	fh	file with error handler (handle)
11 12	IN	errorcode	error code (integer)
13		enoicode	error code (integer)
14	C bi	nding	
15		0	<pre>chandler(MPI_File fh, int errorcode)</pre>
16	Fort	ran 2008 binding	
17 18		0	dler(fh, errorcode, ierror)
19		TYPE(MPI_File), ]	
20		INTEGER, INTENT()	
21		INTEGER, OPTIONAL	., INTENT(OUT) :: ierror
22	Fort	ran binding	
23 24 25		FILE_CALL_ERRHANI INTEGER FH, ERROF	DLER(FH, ERRORCODE, IERROR) ACODE, IERROR
26 27 28	This	function returns MP	s the error handler assigned to the file with the error code supplied. I_SUCCESS in C and the same value in IERROR if the error handler ssuming the process is not aborted and the error handler returns).
29 30 31		Advice to users. T advice to users.)	The default error handler for files is $MPI\_ERRORS\_RETURN.$ (End of
32			
33			
34	MPI_	SESSION_CALL_EF	RHANDLER(session, errorcode)
35	IN	session	session with error handler (handle)
36 37	IN	errorcode	error code (integer)
38			
39	C bi	nding	
40	int ]	MPI_Session_call_	errhandler(MPI_Session session, int errorcode)
41	Fort	ran 2008 binding	
42 43			nandler(session, errorcode, ierror)
44		TYPE(MPI_Session)	, INTENT(IN) :: session
45		INTEGER, INTENT()	
46		INTEGER, OPTIONAL	., INTENT(OUT) :: ierror
47	Fort	ran binding	
48	MPI_	SESSION_CALL_ERRH	HANDLER(SESSION, ERRORCODE, IERROR)

INTEGER SESSION, ERRORCODE, IERROR

This function invokes the error handler assigned to the session 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. Users are warned that handlers should not be called recursively with MPI\_COMM\_CALL\_ERRHANDLER, MPI\_FILE\_CALL\_ERRHANDLER, MPI\_WIN\_CALL\_ERRHANDLER, or MPI\_SESSION\_CALL\_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI\_COMM\_CALL\_ERRHANDLER, MPI\_FILE\_CALL\_ERRHANDLER, MPI\_WIN\_CALL\_ERRHANDLER, or MPI\_SESSION\_CALL\_ERRHANDLER is called inside an error handler.

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (End of advice to users.)

#### Timers and Synchronization 9.6

MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high resolution timers. See also Section 2.6.4.

MPI\_WTIME()

```
C binding
double MPI_Wtime(void)
Fortran 2008 binding
DOUBLE PRECISION MPI_Wtime()
Fortran binding
DOUBLE PRECISION MPI WTIME()
    MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-
clock time since some time in the past.
    The "time in the past" is guaranteed not to change during the life of the process.
The user is responsible for converting large numbers of seconds to other units if they are
preferred.
```

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

ł

```
double starttime, endtime;
starttime = MPI_Wtime();
```

2

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18

19 2021

22

23 $^{24}$ 

25

26

2728

29 30

31

32 33

34

35

36

37 38

39

40

41

42

43

44

4546

47

```
1
                stuff to be timed
                                      . . .
           . . .
\mathbf{2}
                      = MPI_Wtime();
          endtime
3
          printf("That took %f seconds\n", endtime-starttime);
4
     }
5
          The times returned are local to the node that called them. There is no requirement
6
      that different nodes return "the same time." (But see also the discussion of
7
     MPI_WTIME_IS_GLOBAL in Section 9.1.2).
8
9
10
     MPI_WTICK()
11
12
      C binding
13
     double MPI_Wtick(void)
14
15
      Fortran 2008 binding
16
     DOUBLE PRECISION MPI_Wtick()
17
     Fortran binding
18
     DOUBLE PRECISION MPI_WTICK()
19
20
          MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns,
21
      as a double precision value, the number of seconds between successive clock ticks. For
22
      example, if the clock is implemented by the hardware as a counter that is incremented
23
      every millisecond, the value returned by MPI_WTICK should be (10^{-3}).
^{24}
25
26
27
28
29
30
^{31}
32
33
34
35
36
37
38
39
40
^{41}
42
43
44
45
46
47
48
```

## Chapter 10

# 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, MPI\_INFO\_GET, and MPI\_INFO\_GET\_STRING must retain all (key,value) pairs so that layered functionality can also use the lnfo 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 IN or INOUT argument to any MPI routine, it is parsed before that routine returns, so that it may be read, modified or freed immediately after return.

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

4243 444546#-by Martin Done in PR443

1 2 3

5 6

9

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

33

35

36

37

3839

40 41 -error

32 #-update4

Done

PR443

in 34

to be removed. because the routines are moved to the deprecated chapter

1 When the descriptions refer to a key or value as being a boolean, an integer, or a list,  $\mathbf{2}$ they mean the string representation of these types. An implementation may define its own 3 rules for how info value strings are converted to other types, but to ensure portability, every 4 implementation must support the following representations. Valid values for a boolean must 5include the strings "true" and "false" (all lowercase). For integers, valid values must include 6 string representations of decimal values of integers that are within the range of a standard 7integer type in the program. (However it is possible that not every integer is a valid value 8 for a given key.) On positive numbers, + signs are optional. No space may appear between 9 a + or - sign and the leading digit of a number. For comma separated lists, the string 10 must contain valid elements separated by commas. Leading and trailing spaces are stripped 11automatically from the types of info values described above and for each element of a comma 12separated list. These rules apply to all info values of these types. Implementations are free 13 to specify a different interpretation for values of other info keys. 1415MPI\_INFO\_CREATE(info) 1617OUT info info object created (handle) 18 19C binding 20int MPI\_Info\_create(MPI\_Info \*info) 21Fortran 2008 binding 22 MPI\_Info\_create(info, ierror) 23TYPE(MPI\_Info), INTENT(OUT) :: info 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526Fortran binding 27MPI\_INFO\_CREATE(INFO, IERROR) 28INTEGER INFO, IERROR 29 MPI\_INFO\_CREATE creates a new info object. The newly created object contains no 30 key/value pairs.  $^{31}$ 32 33 MPI\_INFO\_SET(info, key, value) 34 INOUT info info object (handle) 35 36 IN key (string) key 37 value IN value (string) 38 39 C binding 40 int MPI\_Info\_set(MPI\_Info info, const char \*key, const char \*value) 41 42Fortran 2008 binding 43 MPI\_Info\_set(info, key, value, ierror) 44 TYPE(MPI\_Info), INTENT(IN) :: info 45CHARACTER(LEN=\*), INTENT(IN) :: key, value 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47Fortran binding 48

MPI_INFO_SET(INFO, KEY, VALUE, IERROR)			1		
INTEGER INFO, IERROR			2		
CHARA	CTER*(*) KEY, VALUE			3 4	
	NFO_SET adds the (key,value)			5	
	ey was previously set. key and		<b>e</b> ,	6	
0	l trailing spaces in key and valowed maximums, the errors		•	7	
raised, resp	/			8	
				9 10	
	DELETE(info low)			11	
	_DELETE(info, key)			12	
INOUT	info	info object (handle)		13	
IN	key	key (string)		14	
				15 16	
C binding				10	
int MPI_I	nfo_delete(MPI_Info info,	const char *key)		18	
	008 binding			19	
	delete(info, key, ierror)			20	
	<pre>MPI_Info), INTENT(IN) :: CTER(LEN=*), INTENT(IN)</pre>			21	
	ER, OPTIONAL, INTENT(OUT)	·		22 23	
Fortran b	,			24	
	DELETE(INFO, KEY, IERROR)			25	
	ER INFO, IERROR			26	
CHARA	CTER*(*) KEY			27	
MPI II	NFO_DELETE deletes a (key,v	value) pair from info. If kev i	s not defined in info.	28 29	
	ses an error of class MPI_ERR_			30	
				31	
MPL INFO	_GET(info, key, valuelen, value	flag)	Deprecated function	32	othor
IN	info		descriptions must be		other
		info object (handle)	moved from the		error
ÍN	key	key (string)	original chapter to the Deprecated Interfaces		364 Done
IN	valuelen	length of value arg (integer)	chapter !!!	37 ir	ר
OUT	value	value (string)		38 P	PR443
OUT	flag	true if key defined, $false$ if not	(boolean)	39	
				40 41	
C binding				42	
int MPI_I	nfo_get(MPI_Info info, co	nst char *key, int value	len, char *value,	43	
	int *flag)			44	
	008 binding			45	
	<pre>get(info, key, valuelen, MPI_Info), INTENT(IN) ::</pre>			46 47	
	CTER(LEN=*), INTENT(IN) ::			47 48	
		5			

```
#-other
                   INTEGER, INTENT(IN) :: valuelen
                   CHARACTER(LEN=valuelen), INTENT(OUT) :: value
#-error
                   LOGICAL, INTENT(OUT) :: flag
#-364
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Done
         5
              Fortran binding
in
         6
PR443
              MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
         7
                   INTEGER INFO, VALUELEN, IERROR
         8
                   CHARACTER*(*) KEY, VALUE
         9
                   LOGICAL FLAG
         10
        11
                   This function retrieves the value associated with key in a previous call to
        12
              MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value,
        13
              otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters
        14
              available in value. If it is less than the actual size of the value, the value is truncated. In
        15
              C, valuelen should be one less than the amount of allocated space to allow for the null
        16
              terminator.
        17
                  If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
        18
        19
                                                                                   Deprecated function
              MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
        20
                                                                                   descriptions must be
        21
                IN
                                                       info object (handle)
                          info
                                                                                   moved from the
        22
                                                                                   original chapter to the
                IN
                          key
                                                       key (string)
        23
                                                                                   Deprecated Interfaces
                OUT
                          valuelen
                                                       length of value arg (integer)
        ^{24}
                                                                                   chapter !!!
        25
                OUT
                                                       true if key defined, false if not (boolean)
                          flag
        26
        27
              C binding
        28
              int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
        29
                              int *flag)
        30
        ^{31}
              Fortran 2008 binding
        32
              MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
                   TYPE(MPI_Info), INTENT(IN) :: info
        33
        ^{34}
                   CHARACTER(LEN=*), INTENT(IN) :: key
                   INTEGER, INTENT(OUT) :: valuelen
        35
                   LOGICAL, INTENT(OUT) :: flag
        36
                   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        37
        38
              Fortran binding
        39
              MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
        40
                   INTEGER INFO, VALUELEN, IERROR
        41
                   CHARACTER*(*) KEY
        42
                   LOGICAL FLAG
        43
        44
                   Retrieves the length of the value associated with key. If key is defined, valuelen is set to
        45
              the length of its associated value and flag is set to true. If key is not defined, valuelen is not
        46
              touched and flag is set to false. The length returned in C does not include the end-of-string
        47
              character.
        48
```

If key is larger than MPI\_MAX\_INFO\_KEY, the call is erroneous.

MPI_INF	O_GET_STRING(ii	nfo, key, buflen, value, flag)	1
IN	info	info object (handle)	2
IN	key	key (string)	3 4
INOUT	buflen	length of buffer (integer)	5
OUT	value	value (string)	6
OUT	flag	true if key defined, false if not (boolean)	7
001	liag	true il key denned, faise il not (boolean)	8 9
C bindi	ng		10
int MPI	_Info_get_string	g(MPI_Info info, const char *key, int *buflen,	11
	char *valı	ue, int *flag)	12
Fortran	2008 binding		13
MPI_Inf	o_get_string(inf	Eo, key, buflen, value, flag, ierror)	14 15
	E(MPI_Info), INT		16
	RACTER(LEN=*), 1 EGER, INTENT(INC	INTENT(IN) :: key	17
		INTENT(OUT) :: value	18
	ICAL, INTENT(OUT		19 20
INT	EGER, OPTIONAL,	INTENT(OUT) :: ierror	20 21
Fortran	binding		22
		FO, KEY, BUFLEN, VALUE, FLAG, IERROR)	23
	EGER INFO, BUFLE		24
	RACTER*(*) KEY,	VALUE	25 26
LUG	ICAL FLAG		20 27
		s the value associated with key in a previous call to	28
		key exists, it sets flag to true and returns the value in value,	29
		e and leaves value unchanged. buflen on input is the size of the put of buflen it is the size of the buffer needed to store the value	30
-		into the function is less than the actual size needed to store the	31 32
0		terminator in C), the value is truncated. On return, the value	32
		required buffer size to hold the value string. If buflen is set to	34
	<u> </u>	C, buflen includes the required space for the null terminator. In	35
greater t		ull terminated string in all cases where the <b>buflen</b> input value is	36
0		PI_MAX_INFO_KEY, the call is erroneous.	37
			38 39
		e MPI_INFO_GET_STRING function can be used to obtain the	40
	—	uffer for a value string by setting the buflen to 0. The returned sed to allocate memory before calling MPI_INFO_GET_STRING	41
		alue string. (End of advice to users.)	42
~0'		o (	43
			44 45
			46

```
1
     MPI_INFO_GET_NKEYS(info, nkeys)
2
       IN
                 info
                                            info object (handle)
3
       OUT
                 nkeys
                                            number of defined keys (integer)
4
5
6
     C binding
7
     int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
8
     Fortran 2008 binding
9
     MPI_Info_get_nkeys(info, nkeys, ierror)
10
          TYPE(MPI_Info), INTENT(IN) :: info
11
          INTEGER, INTENT(OUT) :: nkeys
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
16
          INTEGER INFO, NKEYS, IERROR
17
         MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.
18
19
20
     MPI_INFO_GET_NTHKEY(info, n, key)
21
       IN
                 info
                                            info object (handle)
22
       IN
                                            key number (integer)
23
                 n
^{24}
       OUT
                 key
                                            key (string)
25
26
     C binding
27
     int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
28
     Fortran 2008 binding
29
     MPI_Info_get_nthkey(info, n, key, ierror)
30
^{31}
          TYPE(MPI_Info), INTENT(IN) :: info
32
          INTEGER, INTENT(IN) :: n
          CHARACTER(LEN=*), INTENT(OUT) :: key
33
34
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     Fortran binding
36
     MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
37
          INTEGER INFO, N, IERROR
38
          CHARACTER*(*) KEY
39
40
         This function returns the nth defined key in info. Keys are numbered 0 \dots N-1 where
41
     N is the value returned by MPI_INFO_GET_NKEYS. All keys between 0 and N-1 are
42
     guaranteed to be defined. The number of a given key does not change as long as info is not
     modified with MPI_INFO_SET or MPI_INFO_DELETE.
43
44
45
46
47
48
```

MPL INFO	_DUP(info, newinfo)		1
IN	info	info object (handlo)	2
		info object (handle)	3
OUT	newinfo	info object (handle)	4
<b>a</b> 1 1 1			5
C binding			6
int MPI_I	info_dup(MPI_Info info, M	Pl_Info *newinfo)	7 8
Fortran 2	2008 binding		9
	dup(info, newinfo, ierro:		10
	<pre>MPI_Info), INTENT(IN) ::</pre>		11
	(MPI_Info), INTENT(OUT) :		12
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: lerror	13
Fortran b	binding		14
MPI_INFO_	DUP(INFO, NEWINFO, IERRO	R)	15
INTEG	ER INFO, NEWINFO, IERROR		16
MPL I	NFO DUP duplicates an exis	sting info object, creating a new object, with the	17
	value) pairs and the same ord		18 19
	, <b>.</b>		20
			20
MPI_INFO	_FREE(info)		22
INOUT	info	info object (handle)	23
			24
C binding			25
int MPI_I	<pre>info_free(MPI_Info *info)</pre>		26
Fortran 2	2008 binding		27
	free(info, ierror)		28
TYPE(	MPI_Info), INTENT(INOUT)	:: info	29
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	30 31
Fortran k	ainding		32
	FREE(INFO, IERROR)		33
	ER INFO, IERROR		34
			35
	function frees info and sets it t	to MPI_INFO_NOLL. terpreted each time the info is passed to a routine.	36
		coutine do not affect that interpretation.	37
Changes to		outile do not alleet that interpretation.	38
			39
MPI_INFO	_CREATE_ENV(info)		40
OUT	info	info object (handle)	41
			42 43
C binding	g		43 44
int MPI_I	nfo_create_env(int argc,	<pre>char argv[], MPI_Info *info)</pre>	45
Fortran 9	2008 binding		46
	create_env(info, ierror)		47
··· <b>·</b> _ <b>··· ·</b> _			48

1 2	TYPE(MPI_Info), INTENT(OUT) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3	
4	Fortran binding
5	MPI_INFO_CREATE_ENV(INFO, IERROR)
6	INTEGER INFO, IERROR
7	This routine produces an output object info with the same construction as
8	MPI_INFO_ENV as created during MPI_INIT or MPI_INIT_THREAD when the same argu-
9	ments are used. This construction is described in Section 11.2.1; however, this function can
10	be called when not using the World Model, e.g., when using the Sessions Model. This object
11	is not a direct copy or alias of the MPI_INFO_ENV object and could contain different values
12	based on the input arguments and other sources. Multiple calls to this procedure that are
13	given the same input arguments will produce info objects consistent with the definition of
14	MPI_INFO_ENV. The version for ISO C accepts the argc and argv that are provided by the
15	arguments to main or 0 for argc and NULL for argv. The user is responsible for freeing the
16	info object via MPI_INFO_FREE. This procedure is local.
17	This procedure must always be thread-safe, as defined in Section 11.6. It is one of the
18	few routines that may be called before MPI is initialized or after MPI is finalized.
19 20	Advice to score
20 21	Advice to users.
22	In some circumstances (e.g., when passing 0 to argc and NULL to argv in C or in Fortran
23	where such arguments do not exist), the info object may not be populated or may be
24	populated incompletely because this procedure is local and the implementation may not be able to determine the correct values. Note that this could result in different
25	values in the resulting info object at different MPI processes.
26	
27	(End of advice to users.)
28	
29	
30 31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43 44	
44 45	
46	
47	
48	

## Chapter 11

# Process Initialization, Creation, and Management

### 11.1 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 several approaches to MPI initialization and process management while placing minimal restrictions on the execution environment.

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 or initialization procedure to be performed before the complete set of MPI routines may be called.

To this end, MPI presents two models for MPI process initialization. In the World Model, an initial set of processes is created that are related by their membership in a common MPI\_COMM\_WORLD (see Section 11.2) communicator. In the Sessions Model (Section 11.3), an initial set of processes is also created, but the application must explicitly manage the creation of MPI groups, and hence MPI communicators. MPI\_COMM\_WORLD is only valid for use as a communicator in the World Model, i.e., after a successful call to MPI\_INIT\_THREAD and before a call to MPI\_FINALIZE. An application can employ both of these Process Models concurrently. In multi-component MPI applications, for example, a component such as a library can make use of the Sessions Model to instantiate MPI resources without impacting the rest of the application.

Both of these models also support the *Dynamic Process Model* (see Section 11.7), which provides for the creation and management of additional processes after an MPI application has been started. A major impetus for the *Dynamic Process Model* comes from the PVM [25] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

In developing the *Dynamic Process Model*, the MPI Forum decided not to address resource control because it was not able to design a portable interface that would be ap-

 $^{24}$ 

#### 480 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

propriate for the broad spectrum of existing and potential resource and process controllers.

Process management functionality is included in MPI to enable its use in classes of

MPI assumes that resource control is provided externally.

1

 $\mathbf{2}$ 

3

4

message-passing applications requiring process control. These include task farms, serial 5applications with parallel modules, and problems that require a run-time assessment of the 6 number and type of processes that should be started. 7 The following goals are central to the design of MPI process management: 8 9 • The MPI process model must apply to the vast majority of current parallel environments. 10 11 • MPI must not take over operating system responsibilities. It should instead provide a 12clean interface between an application and system software. 13 14• MPI must guarantee communication determinism in the presense of dynamic processes, 15i.e., dynamic process management must not introduce unavoidable race conditions. 16• MPI must not contain features that compromise performance. 17 18 The Dynamic Process Model addresses these issues in two ways. First, MPI remains 19 primarily a communication library. It does not manage the parallel environment in which 20a parallel program executes, though it provides a minimal interface between an application 21and external resource and process managers. 22 Second, MPI maintains a consistent concept of a communicator, regardless of how its 23members came into existence. A communicator is never changed once created, and it is  $^{24}$ always created using deterministic collective operations. 25262711.2 The World Model 282911.2.1 Starting MPI Processes 30 When using the World Model, MPI is initialized by calling either MPI\_INIT or  $^{31}$ MPI\_INIT\_THREAD. 32 33 34MPI\_INIT() 35 36 C binding 37 int MPI\_Init(int \*argc, char \*\*\*argv) 38 Fortran 2008 binding 39 MPI\_Init(ierror) 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42Fortran binding 43 MPI\_INIT(IERROR) 44 INTEGER IERROR 45

In the World Model, an MPI program must contain exactly one call to an MPI ini-46 tialization routine: MPI\_INIT or MPI\_INIT\_THREAD. MPI\_COMM\_WORLD and 47

MPI\_COMM\_SELF are not valid for use as communicators prior to invocation of MPI\_INIT or 48

MPI\_INIT\_THREAD. Subsequent calls to either of these initialization routines are erroneous. A subset of MPI functions may be invoked before MPI initialization routines are called. See Section 11.4. MPI\_INIT accepts the argc and argv that are provided by the arguments to main or NULL:

```
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    /* parse arguments */
    /* main program */
    MPI_Finalize();    /* see below */
    return 0;
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C.

Failures may disrupt the execution of the program before or during MPI initialization. A high-quality implementation shall not deadlock during MPI initialization, even in the presence of failures. Except for functions with the MPI\_T\_ prefix, failures in MPI operations prior to or during MPI initialization are reported by invoking the initial error handler. Users can use the "mpi\_initial\_errhandler" info key during the launch of MPI processes (e.g., MPI\_COMM\_SPAWN / MPI\_COMM\_SPAWN\_MULTIPLE, or mpiexec) to set a non-fatal initial error handler before MPI initialization. When the initial error handler is set to MPI\_ERRORS\_ABORT, raising an error before or during initialization aborts the local MPI process (i.e., it is similar to calling MPI\_ABORT on MPI\_COMM\_SELF). An implementation may not always be capable of determining, before MPI initialization, what constitutes the local MPI process, or the set of connected processes. In this case, errors before initialization, the initial error handler is associated with MPI\_COMM\_WORLD, MPI\_COMM\_SELF, and the communicator returned by MPI\_COMM\_GET\_PARENT (if any).

Advice to implementors. Some failures may leave MPI in an undefined state, or raise an error before the error handling capabilities are fully operational, in which cases the implementation may be incapable of providing the desired error handling behavior. Of note, in some implementations, the notion of an MPI process is not clearly established in the early stages of MPI initialization (for example, when the implementation considers threads that called MPI\_INIT as independent MPI processes); in this case, before MPI is initialized, the MPI\_ERRORS\_ABORT error handler may abort what would have become multiple MPI processes.

When a failure occurs during MPI initialization, the implementation may decide to return MPI\_SUCCESS from the MPI initialization function instead of raising an error. It is recommended that an implementation masks an initialization error only when it expects that later MPI calls will result in well-specified behavior (i.e., barring additional failures, either the outcome of any call will be correct, or the call will raise an appropriate error). For example, it may be difficult for an implementation to avoid 

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 

unspecified behavior when the group of MPI\_COMM\_WORLD does not contain the same set of MPI processes at all members of the communicator, or if the communicator returned from MPI\_COMM\_GET\_PARENT was not initialized correctly. (*End of advice* to implementors.)

After MPI is initialized, the application can access information about the execution environment by querying the predefined info object MPI\_INFO\_ENV. The following keys are predefined for this object, corresponding to the arguments of MPI\_COMM\_SPAWN or of mpiexec:

- <sup>10</sup> "command" Name of program executed.
- <sup>12</sup> "argv" Space separated arguments to command.
- <sup>14</sup> "maxprocs" Maximum number of MPI processes to start.
- <sup>15</sup> "mpi\_initial\_errhandler" Name of the initial errhandler.
- <sup>17</sup> "**soft**" Allowed values for number of processors.
- <sup>18</sup> <sub>19</sub> "host" Hostname.

1

 $\mathbf{2}$ 

3

4

5

13

21

27

28

29

30 31

32

33

34 35

36

37

- <sup>20</sup> "arch" Architecture name.
- <sup>22</sup> "wdir" Working directory of the MPI process.
- $^{23}_{24}$  "file" Value is the name of a file in which additional information is specified.
- "thread\_level" Requested level of thread support, if requested before the program started
   execution.

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''.

Advice to users. If one of the "argv" arguments contains a space, there is no way to tell from the value of the "argv" info key whether a space is part of the argument or is separating different arguments. (*End of advice to users.*)

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.

<sup>38</sup> In cases where the MPI processes were started with MPI\_COMM\_SPAWN\_MULTIPLE <sup>39</sup> or, equivalently, with a startup mechanism that supports multiple process specifications, <sup>40</sup> then the values stored in the info object MPI\_INFO\_ENV at a process are those values that <sup>41</sup> affect the local MPI process.

42 43

44

45

```
Example 11.1 If MPI is started with a call to
```

```
mpiexec -n 5 -arch x86_64 ocean : -n 10 -arch power9 atmos
```

Then the first 5 processes will have in their MPI\_INFO\_ENV object the pairs (command, ocean), (maxprocs, 5), and (arch, x86\_64). The next 10 processes will have in MPI\_INFO\_ENV (command, atmos), (maxprocs, 10), and (arch, power9)

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.)

The following function may be used to initialize MPI, and to initialize the MPI thread environment, instead of MPI\_INIT.

	,		
			13
MPI_INIT_	THREAD(required, provided)		14 15
IN	required	desired level of thread support (integer)	16
OUT	provided	provided level of thread support (integer)	17
001	provided	provided rever or timead support (integer)	18
C binding	r		19
	·	r ***argv, int required, int *provided)	20
	C .	i starge, ind loquilou, ind spiovidou,	21
	008 binding		22
	thread(required, provided		23
	ER, INTENT(IN) :: require		24
	ER, INTENT(OUT) :: provid ER, OPTIONAL, INTENT(OUT)		25
INIEG	ER, OPIIONAL, INTENI(001)	:: 101101	26
Fortran b	<u>g</u>		27 28
	THREAD (REQUIRED, PROVIDED		29
INTEG	ER REQUIRED, PROVIDED, IE	RROR	30
This c	all initializes MPI in the same	e way that a call to MPI_INIT would. In addition,	31
it initialize	s the thread environment. Th	e argument required is used to specify the desired	32
level of thr	ead support. The possible value	ies are listed in increasing order of thread support.	33
	EAD_SINGLE Only one thread	will execute	34
	AD_SINGLE Only one thread	will execute.	35
		may be multithreaded, but the application must	36
		nakes MPI calls (for the definition of main thread,	37
see N	1PI_IS_THREAD_MAIN on pa	ge $485$ ).	38 39
	AD SERIALIZED The process	s may be multithreaded, and multiple threads may	39 40
	1	time: MPI calls are not made concurrently from	41
	listinct threads (all MPI calls		42
	· · · · · · · · · · · · · · · · · · ·		43
MPI_THRE	<b>EAD_MULTIPLE</b> Multiple three	eads may call MPI, with no restrictions.	44
These valu	es are monotonic; i.e., MPI_T	HREAD_SINGLE < MPI_THREAD_FUNNELED <	45
	AD_SERIALIZED < MPI_THREA		46
Differe	ent processes in MPI_COMM_V	VORLD may require different levels of thread sup-	47
port.			48

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

The call returns in **provided** information about the actual level of thread support that  $\mathbf{2}$ will be provided by MPI. It can be one of the four values listed above.

3 The level(s) of thread support that can be provided by MPI\_INIT\_THREAD will depend 4 on the implementation, and may depend on information provided by the user before the  $\mathbf{5}$ program started to execute (e.g., with arguments to mpiexec). If possible, the call will 6 return provided = required. Failing this, the call will return the least supported level such  $\overline{7}$ that provided > required (thus providing a stronger level of support than required by the 8 user). Finally, if the user requirement cannot be satisfied, then the call will return in 9 provided the highest supported level.

10 A thread compliant MPI implementation will be able to return provided

11= MPI\_THREAD\_MULTIPLE. Such an implementation may always return provided

12= MPI\_THREAD\_MULTIPLE, irrespective of the value of required.

13 An MPI library that is not thread compliant must always return provided =14MPI\_THREAD\_SINGLE, even if MPI\_INIT\_THREAD is called on a multithreaded process. 15The library should also return correct values for the MPI calls that can be executed before 16initialization, even if multiple threads have been spawned.

- Such code is erroneous, but if the MPI initialization is performed by a 18 Rationale. library, the error cannot be detected until MPI\_INIT\_THREAD is called. The require-19 ments in the previous paragraph ensure that the error can be properly detected. (End 20of rationale.) 21
- 2223

 $^{24}$ 

35

37

38

39

41

17

1

A call to MPI\_INIT has the same effect as a call to MPI\_INIT\_THREAD with a required = MPI\_THREAD\_SINGLE.

Vendors may provide (implementation dependent) means to specify the level(s) of 25thread support available when the MPI program is started, e.g., with arguments to mpiexec. 26This will affect the outcome of calls to MPI INIT and MPI\_INIT\_THREAD. Suppose, for 27example, that an MPI program has been started so that only MPI\_THREAD\_MULTIPLE is 28available. Then MPI\_INIT\_THREAD will return provided = MPI\_THREAD\_MULTIPLE, irre-29 spective of the value of required; a call to MPI\_INIT will also initialize the MPI thread support 30 level to MPI\_THREAD\_MULTIPLE. Suppose, instead, that an MPI program has been started  $^{31}$ so that all four levels of thread support are available. Then, a call to MPI\_INIT\_THREAD 32 will return provided = required; alternatively, a call to MPI\_INIT will initialize the MPI 33 thread support level to MPI\_THREAD\_SINGLE. 34

Rationale. Various optimizations are possible when MPI code is executed single-36 threaded, or is executed on multiple threads, but not concurrently: mutual exclusion code may be omitted. Furthermore, if only one thread executes, then the MPI library 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 40 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, 42in a cluster of SMPs, then the process computation is multithreaded, but MPI calls 43 will likely execute on a single thread. 44

45The design accommodates a static specification of the thread support level, for en-46vironments that require static binding of libraries, and for compatibility for current 47 multithreaded MPI codes. (End of rationale.)

48

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 3 environment where malloc is not thread safe, then malloc should not be used by the MPI library. 4 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. 10 Note that required need not be the same value on all processes of MPI\_COMM\_WORLD. 11 (End of advice to implementors.) 1213 As with MPI\_INIT, discussed in Section 11.2.1, the version for ISO C accepts the argc 14and argv that are provided by the arguments to main or NULL for both arguments. 15The following function can be used to query the current level of thread support. 1617 18 MPI\_QUERY\_THREAD(provided) 19 OUT provided level of thread support (integer) provided 2021C binding 22 int MPI\_Query\_thread(int \*provided) 23 $^{24}$ Fortran 2008 binding 25MPI\_Query\_thread(provided, ierror) 26INTEGER, INTENT(OUT) :: provided 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28Fortran binding 29MPI\_QUERY\_THREAD(PROVIDED, IERROR) 30 INTEGER PROVIDED, IERROR 3132 The call returns in provided the current level of thread support, which will be the value 33 returned in provided by MPI\_INIT\_THREAD, if MPI was initialized by a call to 34 MPI\_INIT\_THREAD(). This function is only applicable when using the World Model to 35initialize MPI. In the case of applications using both the World Model and the Sessions 36 Model, this function only returns the thread support level returned in provided by 37 MPI\_INIT\_THREAD. 38 39 MPI\_IS\_THREAD\_MAIN(flag) 40 41 OUT flag true if calling thread is main thread, false otherwise 42(logical) 43 44C binding 45int MPI\_Is\_thread\_main(int \*flag) 4647

Fortran 2008 binding MPI\_Is\_thread\_main(flag, ierror) 1

 $\mathbf{2}$ 

5

6

 $\overline{7}$ 

8

9

```
1
          LOGICAL, INTENT(OUT) :: flag
\mathbf{2}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
      Fortran binding
4
     MPI_IS_THREAD_MAIN(FLAG, IERROR)
5
          LOGICAL FLAG
6
          INTEGER IERROR
7
8
          This function can be called by a thread to determine if it is the main thread (the thread
9
     that called MPI_INIT or MPI_INIT_THREAD). This function is only applicable when using
10
      the World Model to initialize MPI. In the case of applications using both the World Model
11
      and the Sessions Model, this function only returns the thread support level returned in
12
      provided by MPI_INIT_THREAD.
13
          All routines listed in this section must be supported by all MPI implementations.
14
15
           Rationale.
                         MPI libraries are required to provide these calls even if they do not
           support threads, so that portable code that contains invocations to these functions
16
17
           can link correctly. MPI_INIT continues to be supported so as to provide compatibility
18
           with current MPI codes. (End of rationale.)
19
                                It is possible to spawn threads before MPI is initialized, but
           Advice to users.
20
           MPI_COMM_WORLD and MPI_COMM_SELF cannot be used until the World Model is
21
           active, i.e. until MPI_INIT_THREAD is invoked by one thread (which, thereby, be-
22
           comes the main thread). In particular, it is possible to enter the MPI execution with
23
           a multithreaded process.
^{24}
25
           In the World Model, the level of thread support provided is a global property of the
26
           MPI process that can be specified only once, when MPI is initialized on that process (or
27
           before). Portable third party libraries have to be written so as to accommodate any
28
           provided level of thread support. Otherwise, their usage will be restricted to specific
29
           level(s) of thread support. If such a library can run only with specific level(s) of thread
30
           support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be
31
           used to check whether the user initialized MPI to the correct level of thread support
32
           and. (End of advice to users.)
33
34
      11.2.2 Finalizing MPI
35
36
37
      MPI_FINALIZE()
38
39
     C binding
40
      int MPI_Finalize(void)
41
42
      Fortran 2008 binding
     MPI_Finalize(ierror)
43
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     Fortran binding
46
     MPI_FINALIZE(IERROR)
47
          INTEGER IERROR
48
```

This routine cleans up all MPI state associated with the World Model. 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 associated with the World Model. It must locally complete all MPI operations that it initiated and must execute matching calls needed to complete MPI communications initiated by other processes. For example, if the process executed a nonblocking send, it must eventually call MPI\_WAIT, MPI\_TEST, MPI\_REQUEST\_FREE, or any derived function; if the process is the target of a send, then it must post the matching receive; if it is part of a group executing a collective operation, then it must have completed its participation in the operation.

The call to MPI\_FINALIZE does not clean up MPI state associated with objects created using MPI\_SESSION\_INIT and other Sessions Model methods, nor objects created using the communicator returned by MPI\_COMM\_GET\_PARENT. See Sections 11.3 and 11.8.

The call to MPI\_FINALIZE does not free objects created by MPI calls; these objects are freed using MPI\_XXX\_FREE calls.

MPI\_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI\_COMM\_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 11.10.4.

The following examples illustrate these rules.

**Example 11.2** The following code is correct

Process O	Process 1
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>
<pre>MPI_Send(dest=1);</pre>	<pre>MPI_Recv(src=0);</pre>
<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>

**Example 11.3** Without a matching receive, the program is erroneous

Process 0	Process 1
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>
<pre>MPI_Send (dest=1);</pre>	
<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>

**Example 11.4** This program is correct: Process 0 calls MPI\_Finalize after it has executed the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call that completes the matching receive operation before it calls MPI\_Finalize.

Process O	Process 1	42
		43
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>	44
<pre>MPI_Isend(dest=1);</pre>	<pre>MPI_Recv(src=0);</pre>	45
<pre>MPI_Request_free();</pre>	<pre>MPI_Finalize();</pre>	46
<pre>MPI_Finalize();</pre>	<pre>exit();</pre>	47
<pre>exit();</pre>		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

39

40

Example 11.5 This program is correct. The attached buffer is a resource allocated by
 the user, not by MPI; it is available to the user after MPI is finalized.

```
Process 0
                                            Process 1
4
         _____
                                            _____
5
                                            MPI_Init();
         MPI_Init();
6
                                            MPI_Recv(src=0);
         buffer = malloc(1000000);
7
         MPI_Buffer_attach();
                                            MPI_Finalize();
8
         MPI_Send(dest=1));
                                            exit();
9
         MPI_Finalize();
10
         free(buffer);
11
         exit();
12
13
                       This program is correct. The cancel operation must succeed, since the
     Example 11.6
14
     send cannot complete normally. The wait operation, after the call to MPI_Cancel, is local-
15
16
     no matching MPI call is required on process 1. Cancelling a send request by calling
     MPI_CANCEL is deprecated.
17
18
                                           Process 1
         Process 0
19
         _____
                                           _____
20
         MPI_Issend(dest=1);
                                           MPI_Finalize();
21
         MPI_Cancel();
22
         MPI_Wait();
23
         MPI_Finalize();
^{24}
25
           Advice to implementors. Even though a process has executed all MPI calls needed to
26
           complete the communications it is involved with, such communication may not yet be
27
           completed from the viewpoint of the underlying MPI system. For example, a blocking
28
           send may have returned, even though the data is still buffered at the sender in an MPI
29
           buffer; an MPI process may receive a cancel request for a message it has completed
30
           receiving. The MPI implementation must ensure that a process has completed any
31
           involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process
32
           exits after the call to MPI_FINALIZE, this will not cause an ongoing communication
33
           to fail. The MPI implementation should also complete freeing all objects marked for
34
           deletion by MPI calls that freed them. (End of advice to implementors.)
35
36
          Failures may disrupt MPI operations during and after MPI finalization. A high quality
37
     implementation shall not deadlock in MPI finalization, even in the presence of failures. The
38
     normal rules for MPI error handling continue to apply. After MPI_COMM_SELF has been
39
     "freed" (see Section 11.2.4), errors that are not associated with a communicator, window,
40
     or file raise the initial error handler (set during the launch operation, see 11.8.4).
41
          Although it is not required that all processes return from MPI_FINALIZE, it is required
42
     that, when it has not failed or aborted, at least the MPI process that was assigned rank 0
43
     in MPI_COMM_WORLD returns, so that users can know that the MPI portion of the com-
44
     putation is over. In addition, in a POSIX environment, users may desire to supply an exit
45
     code for each process that returns from MPI_FINALIZE.
46
          Note that a failure may terminate the MPI process that was assigned rank 0 in
47
     MPI_COMM_WORLD, in which case it is possible that no MPI process returns from
48
     MPI_FINALIZE.
```

Advice to users. Applications that handle errors are encouraged to implement all rank-specific code before the call to MPI\_FINALIZE. In Example 11.7 below, the process with rank 0 in MPI\_COMM\_WORLD may have been terminated before, during, or after the call to MPI\_FINALIZE, possibly leading to the code after MPI\_FINALIZE never being executed. (*End of advice to users.*)

**Example 11.7** 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);
```

#### 11.2.3 Determining Whether MPI Has Been Initialized When Using the World Model

One of the goals of MPI is to allow for layered libraries. For a library using the World Model, it needs to know if MPI has been initialized using MPI\_INIT or MPI\_INIT\_THREAD. In MPI the function MPI\_INITIALIZED is provided to tell if MPI had been initialized using the World Model. In the World Model, once MPI has been finalized it cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this the function MPI\_FINALIZED is needed.

MPI_INITIALIZED(flag)	32
OUT flag Flag is true if MPI_INIT has been called and false	33
otherwise (logical)	34
	35
C binding	36
int MPI_Initialized(int *flag)	37
Int MFI_Initialized(Int *IIag)	38
Fortran 2008 binding	39
MPI_Initialized(flag, ierror)	40
LOGICAL, INTENT(OUT) :: flag	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
Fortron hinding	43
Fortran binding	44
MPI_INITIALIZED(FLAG, IERROR)	45
LOGICAL FLAG	46
INTEGER IERROR	47
	48

 $\mathbf{2}$ 

This routine may be used to determine whether MPI\_INIT or MPI\_INIT\_THREAD has been called. MPI\_INITIALIZED returns true if the calling process has called either of these MPI procedures. Whether MPI\_FINALIZE has been called does not affect the behavior of MPI\_INITIALIZED. This function must always be thread-safe, as defined in Section 11.6. This function returns false for applications using the Sessions Model exclusively.

8 MPI\_FINALIZED(flag)

OUT

1

 $\mathbf{2}$ 

3

4

5

6

9

10 11

17

21

22

26 27 true if MPI was finalized (logical)

```
C binding
int MPI_Finalized(int *flag)
```

flag

```
<sup>14</sup> Fortran 2008 binding
```

```
<sup>15</sup> MPI_Finalized(flag, ierror)
```

```
LOGICAL, INTENT(OUT) :: flag
```

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

- <sup>18</sup> Fortran binding
- <sup>19</sup> MPI\_FINALIZED(FLAG, IERROR)
  - LOGICAL FLAG
  - INTEGER IERROR

This routine returns true if MPI\_FINALIZE has completed. It is valid to call
 MPI\_FINALIZED before MPI\_INIT and after MPI\_FINALIZE. This function must always be
 thread-safe, as defined in Section 11.6.

### 11.2.4 Allowing User Functions at MPI Finalization

28In the context of the World Model, there are times in which it would be convenient to 29have actions happen when an MPI process finalizes MPI. For example, a routine may do 30 initializations that are useful until the MPI job (or that part of the job that is being termi- $^{31}$ nated in the case of dynamically created processes) finalizes MPI. This can be accomplished 32 in MPI by attaching an attribute to MPI\_COMM\_SELF with a callback function. When 33 MPI\_FINALIZE is called, it will first execute the equivalent of an MPI\_COMM\_FREE on 34MPI\_COMM\_SELF. This will cause the delete callback function to be executed on all keys as-35sociated with MPI\_COMM\_SELF, in the reverse order that they were set on MPI\_COMM\_SELF. 36 If no key has been attached to MPI\_COMM\_SELF, then no callback is invoked. The "freeing" 37 of MPI\_COMM\_SELF occurs before any other parts of MPI are affected. Thus, for example, 38 calling MPI\_FINALIZED will return false in any of these callback functions. Once done with 39 MPI\_COMM\_SELF, the order and rest of the actions taken by MPI\_FINALIZE is not specified. 40

- 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
- $\frac{47}{48}$  are made. (End of advice to implementors.)

#### 11.3The Sessions Model

There are a number of limitations with the World Model described in the preceding section. Among these are the following: MPI cannot be initialized from different application components without a priori knowledge or coordination; MPI cannot be initialized more than once; and MPI cannot be reinitialized after MPI\_FINALIZE has been called. This section describes an alternative approach to MPI initialization—the Sessions Model. With this approach, an MPI application, or components of the application, can instantiate MPI resources for the specific communication needs of this component. MPI\_COMM\_WORLD is not valid for use as a communicator. MPI\_INFO\_ENV is not valid for use as an info object when only using the Sessions Model. As described in Section 11.2.1, MPI must be initialized using the World Model to use this info object.

In the Sessions Model, MPI resources can be allocated and freed multiple times in an MPI process.

As shown in Figure 11.1, when using the Sessions Model, an MPI process instantiates an MPI Session handle, which can be used to query the runtime system about characteristics of the job within which the process is running, as well as other system resources. Using this information, the MPI process can then create an MPI Group based on application requirements and available resources, which in turn can be used to create an MPI Communicator, Window, or File. By judicious creation of communicators, an application 20only needs to allocate MPI resources based on its communication requirements. Although 21there are existing MPI interfaces for creating communicators which can, in principle, allow 22for resource optimizations within an MPI implementation, this can only be done following 23initialization of MPI.

For multithreaded applications the Sessions Model provides fine-grain control of the thread support level for MPI objects. It is possible to specify different thread support levels when creating different *MPI Session handles*. Thus different components of an application can use different thread support levels.

The Sessions Model introduces a concept of isolation. MPI objects derived from differ-29 ent MPI Session handles shall not be intermixed with each other in a single MPI procedure 30 call. MPI objects derived from the Sessions Model shall not be intermixed in a single MPI procedure call with MPI objects derived from the World Model. MPI objects derived from the Sessions Model shall not be intermixed in a single MPI procedure call with MPI objects derived from the communicator obtained from a call to MPI\_COMM\_GET\_PARENT or MPI\_COMM\_JOIN.

This restriction does not apply to generalized requests (Section 13.2) as such requests are not associated directly with communicators or other MPI objects. Note however, the Sessions Model does not otherwise change the semantics or behavior of MPI objects.

45

46 47

48

1  $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

24

25

26

27

28

31

32

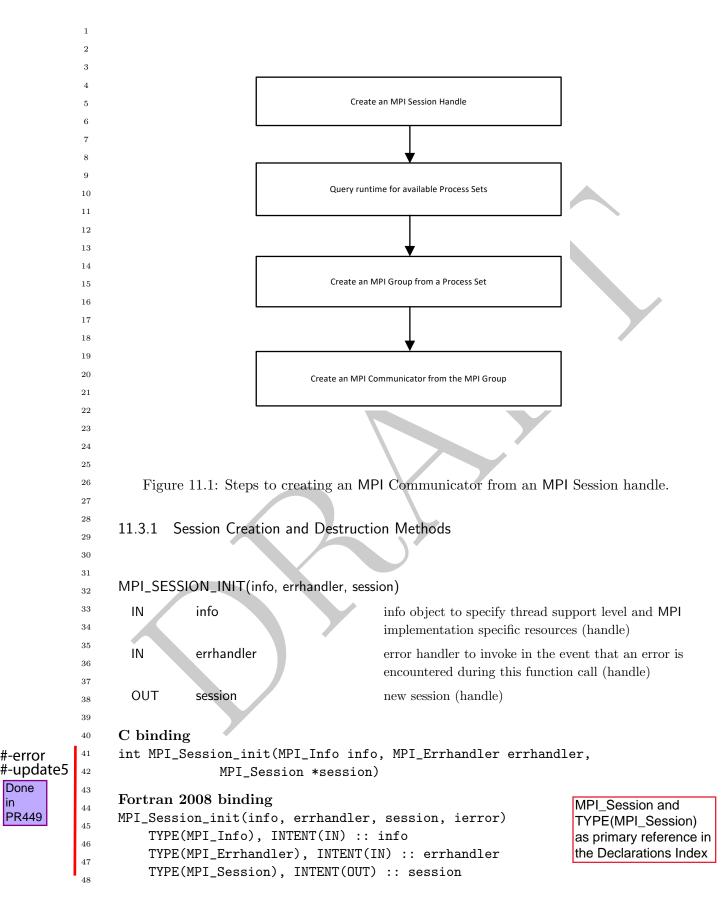
33

34

35

36

37



**Unofficial Draft for Comment Only** 

lin

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

```
MPI_SESSION_INIT(INFO, ERRHANDLER, SESSION, IERROR)
INTEGER INFO, ERRHANDLER, SESSION, IERROR
```

The info argument is used to request MPI functionality requirements and possible MPI implementation specific capabilities. The following info key is predefined:

"mpi\_thread\_support\_level" used to request the thread support level required for MPI objects derived from the Session. Allowed values are "MPI\_THREAD\_SINGLE", "MPI\_THREAD\_FUNNELED", "MPI\_THREAD\_SERIALIZED", and "MPI\_THREAD\_MULTIPLE". Note that the thread support value is specified by a string rather than the integer values supplied to MPI\_INIT\_THREAD. The thread support level actually provided by the MPI implementation can be determined via a subse- quent call to MPI\_SESSION\_GET\_INFO to return the info object associated with the Session. The default thread support level is MPI implementation dependent.

The errhandler argument specifies an error handler to invoke in the event that the Session instantiation call encounters an error. The error handler shall be either a pre-defined error handler (see 9.3) or one created using MPI\_SESSION\_CREATE\_ERRHANDLER. Session instantiation is intended to be a lightweight operation. An MPI process may instantiate multiple Sessions. MPI\_SESSION\_INIT is always thread safe; multiple threads within an application may invoke it concurrently.

Advice to users. Requesting "MPI\_THREAD\_SINGLE" thread support level is generally not recommended, because this will conflict with other components of an application requesting higher levels of thread support. (*End of advice to users.*)

Advice to implementors. Owing to the restrictions of the MPI\_THREAD\_SINGLE thread support level, implementators are discouraged from making this the default thread support level for Sessions. (*End of advice to implementors.*)

	•	33
MPI_SESSION_FINALIZE(session)		34
		35
IN session	session to be finalized (handle)	36
		37
C binding		38
<pre>int MPI_Session_finalize(MPI_Se</pre>	ssion *session)	39
Fortran 2008 binding		40
6	、 、	41
MPI_Session_finalize(session, i	error)	42
TYPE(MPI_Session), INTENT(I	NOUT) :: session	
INTEGER, OPTIONAL, INTENT(O	UT) :: ierror	43
,,,,,		44
Fortran binding		45
MPI_SESSION_FINALIZE(SESSION, I	ERROR)	46
INTEGER SESSION, IERROR		47
		48

<sup>1</sup> This routine cleans up all MPI state associated with the supplied **session**. Every instantiated

<sup>2</sup> Session must be finalized using MPI\_SESSION\_FINALIZE. The handle session is set to
 <sup>3</sup> MPI\_SESSION\_NULL by the call.

<sup>4</sup> Before an MPI process invokes MPI\_SESSION\_FINALIZE, the process must perform <sup>5</sup> all MPI calls needed to complete its involvement in MPI communications: it must locally <sup>6</sup> complete all MPI operations that it initiated and it must execute matching calls needed to <sup>7</sup> complete MPI communications initiated by other processes.

The call to MPI\_SESSION\_FINALIZE does not free objects created by MPI calls; these
 objects are freed using MPI\_XXX\_FREE calls.

<sup>10</sup> MPI\_SESSION\_FINALIZE is collective over all MPI processes that are connected via <sup>11</sup> MPI Communicators, Windows, or Files that were created as part of the Session and still <sup>12</sup> exist. If processes were spawned, accepted, or connected using MPI Communicators created <sup>13</sup> as part of this session, this operation is collective over the union of all processes that have <sup>14</sup> been and continue to be connected via those objects, as explained in Section 11.10.4.

Advice to implementors. An MPI implementation should be able to implement the semantics of MPI\_SESSION\_FINALIZE without synchronization with other MPI processes, provided an application frees all MPI windows, closes all MPI files, and uses MPI\_COMM\_DISCONNECT to free all MPI communicators associated with a session prior to invoking MPI\_SESSION\_FINALIZE on the corresponding session handle. (*End of advice to implementors.*)

#### 11.3.2 Processes Sets

15

16

17

18

19

20

21 22 23

36

Process sets are the mechanism for MPI applications to query the runtime. Process sets are identified by process set names. Process set names have a *Uniform Resource Identifier* (URI) format. Two process set names are mandated: "mpi://WORLD" and

<sup>28</sup> "mpi://SELF". Additional process set names may be defined, for example,

"mpix://UNIVERSE" and "hwloc://L3Cache" may be defined by the MPI implementation. The
 "mpi://" namespace is reserved for exclusive use by the MPI standard. Figure 11.2 depicts
 process sets that the runtime could associate with an instance of an MPI job. In this
 example, the two mandated process sets are defined, in addition to optional, implementation
 specific ones.

Mechanisms for defining process sets and how system resources are assigned to these sets is considered to be implementation dependent.

A process set caches key/value tuples that are accessible to the application via an MPI\_Info object. The "mpi\_size" key is mandatory for all process sets.

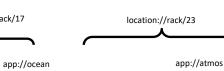
 $14 \\ 15$ 

•	
۱.	

mpi://SELF

 $\sim$ 

MPI process 4



mpi://SELF

 $\sim$ 

MPI process 3

job://12942

mpi://WORLD

mpi://SELF

 $\prec$ 

MPI process 2

location://rack/17

ᄼ

mpi://SELF

 $\sim$ 

MPI process 1

mpi://SELF

 $\sim$ 

MPI process 0

Figure 11.2: Exam	ples of process sets.	Illustrated an	re the two ma	ndated process sets -
"mpi://WORLD" and	d "mpi://SELF" - alon	g with several	l optional ones	that a runtime could
define. In this example	nple, MPI_SESSION_	GET_NUM_PS	SETS would re	turn five at each MPI
process.				

11.3.3 Runtime Qu	uery Functions
-------------------	----------------

MPI_SESS	ION_GET_NUM_PSETS(session	on, info, npset_names)	
IN	session	session (handle)	:
IN	info	info object (handle)	
Ουτ	npset_names	number of available process sets (non-negative integer)	
C binding int MPI_Session_get_num_psets(MPI_Session session, MPI_Info info, int *npset_names)			
Fortran 2	008 binding		4
<pre>MPI_Session_get_num_psets(session, info, npset_names, ierror)    TYPE(MPI_Session), INTENT(IN) :: session    TYPE(MPI_Info), INTENT(IN) :: info    INTEGER, INTENT(OUT) :: npset_names    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			
Fortran b MPI_SESSI	0	, INFO, NPSET_NAMES, IERROR)	4

#### 496 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 INTEGER SESSION, INFO, NPSET\_NAMES, IERROR 2 This function is used to query the runtime for the number of available process sets in 3 which the calling MPI process is a member. An MPI implementation is allowed to increase 4 the number of available process sets during the execution of an MPI application when new 5process sets become available. However, MPI implementations are not allowed to change 6 the index of a particular process set name, or to change the name of the process set at a 7 particular index, or to delete a process set name once it has been added. When a process 8 set becomes invalid, for example, when some processes become unreachable due to failures 9 in the communication system, subsequent usage of the process set name should raise an 10 error. For example, creating an MPI\_Group from such a process set might succeed because it 11 is a local operation, but creating an MPI\_Comm from that group and attempting collective 12communication should raise an error. 13 14Advice to implementation. It is anticipated that an MPI implementation may be re-15lying on an external runtime system to provide process sets. Such runtime systems 16may have the ability to dynamically create process sets during the course of appli-17 cation execution. Requiring the number of process sets returned by 18 MPI\_SESSION\_GET\_NUM\_PSETS to be constant over the course of application exe-19 cution would prevent an application from taking advantage of such capabilities. (End 20of advice to implementors.) 2122 23 $^{24}$ MPI\_SESSION\_GET\_NTH\_PSET(session, info, n, pset\_len, pset\_name) 25IN session (handle) session 26IN info info object (handle) 2728IN index of the desired process set name (integer) n 29 INOUT pset\_len length of the pset\_name argument (integer) 30 OUT name of the nth process set (string) pset\_name  $^{31}$ 32 C binding 33 34int MPI\_Session\_get\_nth\_pset(MPI\_Session session, MPI\_Info info, int n, int \*pset\_len, char \*pset\_name) 35 36 Fortran 2008 binding 37 MPI\_Session\_get\_nth\_pset(session, info, n, pset\_len, pset\_name, ierror) 38 TYPE(MPI\_Session), INTENT(IN) :: session 39 TYPE(MPI\_Info), INTENT(IN) :: info 40 INTEGER, INTENT(IN) :: n 41 INTEGER, INTENT(INOUT) :: pset\_len 42CHARACTER(LEN=\*), INTENT(OUT) :: pset\_name 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 45Fortran binding MPI\_SESSION\_GET\_NTH\_PSET(SESSION, INFO, N, PSET\_LEN, PSET\_NAME, IERROR) 4647INTEGER SESSION, INFO, N, PSET\_LEN, IERROR 48 CHARACTER\*(\*) PSET\_NAME

This function returns the name of the nth process set in the supplied pset\_name buffer. pset\_len is the size of the buffer needed to store the nth process set name. If the pset\_len passed into the function is less than the actual buffer size needed for the process set name, then the string value returned in pset\_name is truncated. If pset\_len is set to 0, pset\_name is not changed. On return, the value of pset\_len will be set to the required buffer size to hold the process set name. In C, pset\_len includes the required space for the null terminator. In C, this function returns a null terminated string in all cases where the pset\_len input value is greater than 0.

If two MPI processes get the same process set name, then the intersection of the two process sets shall either be the empty set or identical to the union of the two process sets.

After a successful call to MPI\_SESSION\_GET\_NTH\_PSET, subsequent calls to routines that query information about the same process set name and same session handle must return the same information. An MPI implementation is not allowed to alter any of the returned process set names.

Process set names have an implementation-defined maximum length of MPI\_MAX\_PSET\_NAME\_LEN.

Advice to users. MPI\_MAX\_PSET\_NAME\_LEN might be very large, so it might not be wise to declare a string of that size. Users are encouraged to use MPI\_SESSION\_GET\_NTH\_PSET both for obtaining the length of a pset\_name and the process set name. (*End of advice to users.*)

MPI_SESSION_GET_INFO(session, info_used)			24
		25	
IN	session	session (handle)	26
OUT	info_used	see explanation below (handle)	27
			28
C bindin	σ		29
	int MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)		30
THE HIT'	Int Mr1_Session_get_Init(Mr1_Session Session, Mr1_Inito *Inito_used)		
Fortran 2008 binding			32
MPI_Sessi	ion_get_info(session, info	o_used, ierror)	33
TYPE	(MPI_Session), INTENT(IN)	:: session	34
TYPE	(MPI_Info), INTENT(OUT) :	info_used	35
INTEC	GER, OPTIONAL, INTENT(OUT)	:: ierror	36
Frankara 1	the dimension		37
Fortran binding			38
_	ION_GET_INFO(SESSION, INFO		39
INTEC	GER SESSION, INFO_USED, II	IKKUK	40

MPI\_SESSION\_GET\_INFO returns a new info object containing the hints of the MPI 41 Session associated with session. The current setting of all hints related to this MPI Session 42is returned in info\_used. An MPI implementation is required to return all hints that are 43 supported by the implementation and have default values specified; any user-supplied hints 44that were not ignored by the implementation; and any additional hints that were set by 45the implementation. If no such hints exist, a handle to a newly created info object is 46returned that contains no key/value pair. The user is responsible for freeing info\_used via 47MPI\_INFO\_FREE. 48

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9

10 11

12

13

14 15

16

17

18

19

20

#### 498 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 MPI\_SESSION\_GET\_PSET\_INFO(session, pset\_name, info) 2 IN session session (handle) 3 IN pset\_name name of process set (string) 4 5OUT info info object containing information about the given 6 process set (handle) 7 8 C binding 9 int MPI\_Session\_get\_pset\_info(MPI\_Session session, const char \*pset\_name, 10 MPI\_Info \*info) 11 Fortran 2008 binding 12MPI\_Session\_get\_pset\_info(session, pset\_name, info, ierror) 13 TYPE(MPI\_Session), INTENT(IN) :: session 14CHARACTER(LEN=\*), INTENT(IN) :: pset\_name 15TYPE(MPI\_Info), INTENT(OUT) :: info 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 18 Fortran binding 19MPI\_SESSION\_GET\_PSET\_INFO(SESSION, PSET\_NAME, INFO, IERROR) 20INTEGER SESSION, INFO, IERROR 21CHARACTER\*(\*) PSET\_NAME 22 This function is used to query properties of a specific process set. The returned *info* 23object can be queried with existing MPI info object query functions. One key/value pair  $^{24}$ must be defined, "mpi\_size". The value of the "mpi\_size" key specifies the number of MPI 25processes in the process set. The user is responsible for freeing the returned MPI\_Info object. 262711.3.4 Sessions Model Examples 2829This section presents several examples of how to use MPI Sessions to create MPI Groups 30 and MPI Communicators.  $^{31}$ 32 Example 11.8 Simple example illustrating creation of an MPI communicator using the 33 Sessions Model. 34 35 #include <stdio.h> 36 #include <stdlib.h> 37 #include <string.h> 38#include "mpi.h" 39 40static MPI\_Session lib\_shandle = MPI\_SESSION\_NULL; 41static MPI\_Comm lib\_comm = MPI\_COMM\_NULL; 4243int library\_foo\_init(void) 44{ 45int rc, flag; 46int ret = 0;47const char pset\_name[] = "mpi://WORLD"; 48 const char mt\_key[] = "mpi\_thread\_support\_level";

```
const char mt_value[] = "MPI_THREAD_MULTIPLE";
char out_value[100];
                       /* large enough */
MPI_Group wgroup = MPI_GROUP_NULL;
MPI_Info sinfo = MPI_INFO_NULL;
MPI_Info tinfo = MPI_INFO_NULL;
MPI_Info_create(&sinfo);
MPI_Info_set(sinfo, mt_key, mt_value);
rc = MPI_Session_init(sinfo, MPI_ERRORS_RETURN,
                       &lib_shandle);
if (rc != MPI_SUCCESS) {
   ret = -1;
   goto fn_exit;
}
/*
 * check we got thread support level foo library needs
 */
rc = MPI_Session_get_info(lib_shandle, &tinfo);
if (rc != MPI_SUCCESS) {
   ret = -1;
   goto fn_exit;
}
MPI_Info_get(tinfo, mt_key, sizeof(out_value),
             out_value, &flag);
if (flag != 1) {
   printf("Could not find key %s\n", mt_key);
   ret = -1;
   goto fn_exit;
}
if (strcmp(out_value, mt_value)) {
   printf("Did not get thread multiple support, got %s\n",
          out_value);
   ret = -1;
   goto fn_exit;
}
/*
 * create a group from the WORLD process set
 */
rc = MPI_Group_from_session_pset(lib_shandle,
                                 pset_name,
                                  &wgroup);
if (rc != MPI_SUCCESS) {
   ret = -1;
   goto fn_exit;
```

1

 $\mathbf{2}$ 

3

4

5 6

7

9 10

11

12 13

14 15 16

17

18

19

20 21

22

23 24 25

26

27

28 29

30

31 32 33

34

35

36 37

38 39

40 41

42

43 44

45

46

47

```
1
         }
\mathbf{2}
3
         /*
4
          * get a communicator
5
          */
6
         rc = MPI_Comm_create_from_group(wgroup,
7
                                              "org.mpi-forum.mpi-v4_0.example-ex10_8",
8
                                              MPI_INFO_NULL,
9
                                              MPI_ERRORS_RETURN,
10
                                              &lib_comm);
11
         if (rc != MPI_SUCCESS) {
12
            ret = -1;
13
            goto fn_exit;
14
         }
15
16
         /*
17
          * free group, library doesn't need it.
18
          */
19
20
     fn_exit:
21
         MPI_Group_free(&wgroup);
22
23
         if (sinfo != MPI_INFO_NULL) {
^{24}
            MPI_Info_free(&sinfo);
25
         }
26
27
         if (tinfo != MPI_INFO_NULL) {
28
            MPI_Info_free(&tinfo);
29
         }
30
^{31}
         if (ret != 0) {
32
            MPI_Session_finalize(&lib_shandle);
33
         }
34
35
         return ret;
36
     }
37
         Example 11.8 shows how the pre-defined "mpi://WORLD" process set can be used to
38
```

Example 11.8 shows how the pre-defined "mpi://WORLD" process set can be used to first create a local MPI group and then subsequently to create an MPI communicator from this group.

Example 11.9 This example illustrates the use of Process Set query functions to select a Process Set to use for MPI Group creation.

```
<sup>44</sup> #include <stdio.h>
<sup>45</sup> #include <stdlib.h>
<sup>46</sup> #include <string.h>
<sup>47</sup> #include "mpi.h"
```

```
1
int main(int argc, char *argv[])
                                                                                      \mathbf{2}
{
                                                                                      3
   int i, n_psets, psetlen, rc, ret;
   int valuelen;
                                                                                      4
                                                                                      5
   int flag = 0;
                                                                                      6
   char *pset_name = NULL;
   char *info_val = NULL;
                                                                                      7
   MPI_Session shandle = MPI_SESSION_NULL;
                                                                                      9
   MPI_Info sinfo = MPI_INFO_NULL;
                                                                                      10
   MPI_Group pgroup = MPI_GROUP_NULL;
                                                                                      11
   if (argc < 2) {
                                                                                      12
      fprintf(stderr, "A process set name fragment is required\n");
                                                                                      13
                                                                                      14
      return -1;
                                                                                      15
   }
                                                                                      16
                                                                                      17
   rc = MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &shandle);
                                                                                      18
   if (rc != MPI_SUCCESS) {
      fprintf(stderr, "Could not initialize session, bailing out\n");
                                                                                      19
      return -1;
                                                                                      20
   }
                                                                                      21
                                                                                      22
   MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, &n_psets);
                                                                                      23
                                                                                      ^{24}
   for (i=0, pset_name=NULL; i<n_psets; i++) {</pre>
                                                                                      25
                                                                                      26
       psetlen = 0;
       MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
                                                                                      27
                                                                                      28
                                   &psetlen, NULL);
                                                                                      29
       pset_name = (char *)malloc(sizeof(char) * psetlen);
                                                                                      30
       MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
                                                                                      31
                                   &psetlen, pset_name);
       if (strstr(pset_name, argv[1]) != NULL) break;
                                                                                      32
                                                                                      33
                                                                                      34
       free(pset_name);
       pset_name = NULL;
                                                                                      35
   }
                                                                                      36
                                                                                      37
                                                                                      38
   /*
                                                                                      39
    * get instance of an info object for this Session
                                                                                      40
    */
                                                                                      41
                                                                                      42
   MPI_Session_get_pset_info(shandle, pset_name, &sinfo);
   MPI_Info_get_valuelen(sinfo, "mpi_size", &valuelen, &flag);
                                                                                      43
                                                                                      44
   info_val = (char *)malloc(valuelen+1);
   MPI_Info_get(sinfo, "mpi_size", valuelen, info_val, &flag);
                                                                                      45
                                                                                      46
   free(info_val);
                                                                                      47
                                                                                      48
```

```
/*
```

```
1
          * create a group from the process set
\mathbf{2}
          */
3
4
        rc = MPI_Group_from_session_pset(shandle, pset_name,
5
                                              &pgroup);
6
        ret = (rc == MPI_SUCCESS) ? 0 : -1;
7
8
        free(pset_name);
9
        MPI_Group_free(&pgroup);
10
        MPI_Info_free(&sinfo);
11
        MPI_Session_finalize(&shandle);
12
13
        fprintf(stderr, "Test completed ret = %d\n", ret);
14
        return ret;
15
16
     }
17
         Example 11.9 illustrates several aspects of the Sessions Model. First, the default error
18
     handler can be specified when instantiating a Session instance. Second, there must be at
19
     least two process sets associated with a Session. Third, the example illustrates use of the
20
     Sessions info object and the one required key: "mpi_size".
21
22
     Example 11.10 A Fortran 2008 example illustrating how to obtain information about
23
     available process sets, create an MPI Group from a process set, and subsequently create an
24
     MPI Communicator.
25
26
     PROGRAM MAIN
27
         USE mpi_f08
28
         IMPLICIT NONE
29
         INTEGER :: pset_len, ierror, n_psets
30
         CHARACTER(LEN=:), ALLOCATABLE :: pset_name
31
         TYPE(MPI_Session) :: shandle
32
         TYPE(MPI_Group) :: pgroup
33
         TYPE(MPI_Comm) :: pcomm
34
35
         CALL MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &
36
                                 shandle, ierror)
37
         IF (ierror .NE. MPI_SUCCESS) THEN
38
             WRITE(*,*) "MPI_Session_init failed"
39
             ERROR STOP
40
         END IF
41
42
         CALL MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, n_psets)
43
         IF (n_psets .LT. 2) THEN
44
             WRITE(*,*) "MPI_Session_get_num_psets didn't return at least 2 psets"
45
             ERROR STOP
46
         END IF
47
48
     ļ
```

than MPI\_SESSION\_FINALIZE.

```
1
!
    Just get the second pset's length and name
                                                                                       \mathbf{2}
!
    Note that index values are zero-based, even in Fortran
!
    pset_len = 0
                                                                                       5
    CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
                                                                                       6
                                                                       &
                                     pset_len, pset_name)
    ALLOCATE(CHARACTER(LEN=pset_len)::pset_name)
                                                                                       9
    CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
                                                                       &
                                                                                       10
                                     pset_len, pset_name)
                                                                                       11
ļ
                                                                                       12
                                                                                       13
!
    create a group from the pset
                                                                                       14
!
                                                                                       15
    CALL MPI_Group_from_session_pset(shandle, pset_name, pgroup)
                                                                                       16
L
                                                                                       17
i
    free the buffer used for the pset name
                                                                                       18
!
                                                                                       19
    DEALLOCATE(pset_name)
                                                                                       20
                                                                                      21
ļ
                                                                                       22
!
    create a MPI communicator from the group
                                                                                      23
ļ
                                                                                       ^{24}
                                                "session_example",
    CALL MPI_Comm_create_from_group(pgroup,
                                                                       &
                                                                                       25
                                                MPI_INFO_NULL,
                                                                       &
                                                                                       26
                                                MPI_ERRORS_RETURN,
                                                                       &
                                                pcomm)
                                                                                       27
                                                                                       28
                                                                                       29
    CALL MPI_Barrier(pcomm, ierror)
                                                                                       30
    IF (ierror .NE. MPI_SUCCESS) THEN
                                                                                       31
        WRITE(*,*) "Barrier call on communicator failed"
        ERROR STOP
                                                                                       32
    END IF
                                                                                       33
                                                                                      34
    CALL MPI_Comm_free(pcomm)
                                                                                      35
    CALL MPI_Group_free(pgroup)
                                                                                       36
                                                                                      37
    CALL MPI_Session_finalize(shandle, ierror)
                                                                                       38
                                                                                       39
END PROGRAM MAIN
                                                                                       40
                                                                                       41
    Note in this example that the call to MPI_SESSION_FINALIZE may block in order
                                                                                       42
to ensure that the calling MPI process has completed its involvement in the preceding
                                                                                       43
MPI_BARRIER operation. If MPI_COMM_DISCONNECT had been used instead of
                                                                                       44
MPI_COMM_FREE, the example would have blocked in MPI_COMM_DISCONNECT rather
                                                                                       45
```

```
Unofficial Draft for Comment Only
```

## 11.4 Common Elements of Both Process Models

## 11.4.1 MPI Functionality that is Always Available

1 2

3 4

5

6

 $\overline{7}$ 

8

33

36

37

38

Some MPI functions may be invoked at any time, including prior to calling MPI\_INIT or MPI\_SESSION\_INIT, and following MPI finalization, independent of whether the World Model, Sessions Model, or both are used. These functions can be called concurrently by multiple threads within an MPI Process. Table 11.1 lists the applicable MPI functions.

	9	MPI_INITIALIZED	
	.0	MPI_FINALIZED	
	1	MPI_GET_VERSION	
	2	MPI_GET_LIBRARY_VERSION	
	.3	MPI_INFO_CREATE	
	4	MPI_INFO_CREATE_ENV	
	5	MPI_INFO_SET	
	.6	MPI_INFO_DELETE	
	7	MPI_INFO_GET	
	.8	MPI_INFO_GET_VALUELEN	
	9	MPI_INFO_GET_NKEYS	
:	20	MPI_INFO_GET_NTHKEY	
:	21	MPI_INFO_DUP	
:	22	MPI_INFO_FREE	
:	23	MPI_INFO_F2C	
:	24	MPI_INFO_C2F	
:	25	MPI_SESSION_CREATE_ERRHANDLER	
:	26	MPI_SESSION_CALL_ERRHANDLER	
:	27	MPI_ERRHANDLER_FREE	
:	28	MPI_ERRHANDLER_F2C	
:	29	MPI_ERRHANDLER_C2F	
:	30	MPI_ERROR_STRING	
:	31		
;	32	MPI_ERROR_CLASS	

Table 11.1: List of MPI Functions that can be called at any time within an MPI program,
 including prior to MPI initialization and following MPI finalization

In addition to the functions listed in Table 11.1, any function with the prefix MPI\_T\_ (within the constraints for functions with this prefix listed in Section 15.3.4) may also be called prior to MPI initialization and after MPI finalization.

11.4.2	Aborting MPI Processes		
MPI_AB	ORT(comm, errorcode)		
IN	comm	communicator of tasks to abort (handle)	
IN	errorcode	error code to return to invoking environment (integer)	
C binding int MPI_Abort(MPI_Comm comm, int errorcode)			
	2008 binding		

MPI\_Abort(comm, errorcode, ierror)
 TYPE(MPI\_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(IN) :: errorcode
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_ABORT(COMM, ERRORCODE, IERROR) INTEGER COMM, ERRORCODE, IERROR

This routine makes a "best attempt" to abort all MPI processes in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a return errorcode from the main program.

It may not be possible for an MPI implementation to abort only the processes represented by comm if this is a subset of the processes. In this case, the MPI implementation should attempt to abort all the connected processes but should not abort any unconnected processes. When using the World Model, and if no processes were spawned, accepted, or connected then this has the effect of aborting all the processes associated with MPI\_COMM\_WORLD. In the case of the Sessions Model, if an MPI process has instantiated multiple sessions, the union of the process sets in these sessions are considered connected processes. Thus invoking MPI\_ABORT on a communicator derived from one of these sessions will result in all MPI processes in this union being aborted.

Advice to implementors. After aborting a subset of processes, a high quality implementation should be able to provide error handling for communicators, windows, and files involving both aborted and non-aborted processes. As an example, if the user changes the error handler for MPI\_COMM\_WORLD to MPI\_ERRORS\_RETURN or a 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 an appropriate error code when attempting communication with an aborted process (e.g., an error of class MPI\_ERR\_PROC\_ABORTED). A high quality implementation should support equivalent behavior for communicators derived from sessions. (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.*)

#### Unofficial Draft for Comment Only

 $45 \\ 46$ 

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.)

## 11.5 Portable MPI Process Startup

A number of implementations of  $\mathsf{MPI}$  provide a startup command for  $\mathsf{MPI}$  programs that is of the form

9 10 11

27 28

29

34

35

36

37

38

1

2

3

4 5

> 6 7

> 8

#### 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 startup mechanism. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead of mpirun MPI specifies mpiexec.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an **mpiexec** startup command and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called **mpiexec**, it must be of the form described below.

- It is suggested that
- mpiexec -n <numprocs> <program>

be at least one way to start <program> with an initial set of <numprocs> processes, which
 will be accessible as the process set named "mpi://world" in the Sessions Model and/or
 used to the form the group associated with the built-in communicator, MPI\_COMM\_WORLD
 in the World Model. 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 order that mpiexec be able to be viewed as a command-line version of MPI\_COMM\_SPAWN (See Section 11.8.4).

Analogous to MPI\_COMM\_SPAWN, we have

mpiexec -n	<maxp< th=""><th>orocs&gt;</th></maxp<>	orocs>
-soft	<	>
-host	<	>
-arch	<	>
-wdir	<	>
-path	<	>
-file	<	>
-initial-errhandler	<	>

. . .

<command line>

for the case where a single command line for the application program and its arguments will suffice. See Section 11.8.4 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  $-n \ge argument$  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 11.11** Start 16 instances of myprog on the current or default machine:

mpiexec -n 16 myprog

**Example 11.12** Start 10 instances of myprog on the machine called ferrari:

mpiexec -n 10 -host ferrari myprog

**Example 11.13** Start 3 instances of the same program myprog with different command-line arguments:

mpiexec myprog infile1 : myprog infile2 : myprog infile3

**Example 11.14** Start 5 instances of the ocean program on x86\_64 hosts and 10 instances of the atmos program on Power9 hosts (Form B):

mpiexec -n 5 -arch x86\_64 ocean : -n 10 -arch power9 atmos

Unofficial Draft for Comment Only

 $\mathbf{2}$ 

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

**Example 11.15** Start the ocean program on five Suns and the atmos program on 10 RS/6000's (Form B):

mpiexec -configfile myfile

where myfile contains

1

2

3 4

5

6 7

8 9

10 11

12

13 14

1516

17

32

33

37

38 39

40

41

42

43 44 -n 5 -arch sun ocean

-n 10 -arch rs6000 atmos

(End of advice to implementors.)

#### MPI and Threads 11.6

18 This section specifies the interaction between MPI calls and threads. Although thread com-19 pliance is not required, the standard specifies how threads are to work if they are provided. 20The section lists minimal requirements for thread compliant MPI implementations and 21defines functions that can be used for initializing the thread environment. MPI may be im-22 plemented in environments where threads are not supported or perform poorly. Therefore, 23MPI implementations are not required to be thread compliant as defined in this section. 24Regardless of whether or not the MPI implementation is thread compliant, a subset of MPI 25functions must always be thread safe. A complete list of such MPI functions is given in Ta-26ble 11.1. When a thread is executing one of these routines, if another concurrently running 27thread also makes an MPI call, the outcome will be as if the calls executed in some order. 28This section generally assumes a thread package similar to POSIX threads [44], but

29 the syntax and semantics of thread calls are not specified here—these are beyond the scope 30 of this document.  $^{31}$ 

11.6.1 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 can be received by any thread in this process.

This model corresponds to the POSIX model of interprocess commu-Rationale. nication: the fact that a process is multithreaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations in which MPI 'processes' are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their "processes" are single-threaded). (End of rationale.)

Advice to users. It is the user's responsibility to prevent races when threads within 45the same application post conflicting communication calls. The user can make sure 4647 that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. (End of advice to users.) 48

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 11.16** 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.*)

## 11.6.2 Clarifications

Initialization and Completion When using the World Model, 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.)

Threads and the Sessions Model The Sessions Model provides a finer-grain approach to <sup>45</sup> controlling the interaction between MPI calls and threads. When using this model, the <sup>46</sup> desired level of thread support is specified at Session initialization time. See Section 11.3. <sup>47</sup> Thus it is possible for communicators and other MPI objects derived from one Session <sup>48</sup>

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

to provide a different level of thread support than those created from another Session
 for which a different level of thread support was requested. Depending on the level of
 thread support requested at Session initialization time, different threads in a MPI process
 can make concurrent calls to MPI when using MPI objects derived from different session
 handles. Note that the requested and provided level of thread support when creating a
 Session may influence the granted level of thread support in a subsequent invocation of
 MPI\_SESSION\_INIT. Likewise, if the application at some point calls

<sup>8</sup> MPI\_INIT\_THREAD, the requested and granted level of thread support may influence the <sup>9</sup> granted level of thread support for subsequent calls to MPI\_SESSION\_INIT. Similarly, if the <sup>10</sup> application calls MPI\_INIT\_THREAD after a call to MPI\_SESSION\_INIT, the level of thread <sup>11</sup> support returned from MPI\_INIT\_THREAD may be similarly influenced by the requested <sup>12</sup> level of thread support in the prior call to MPI\_SESSION\_INIT.

<sup>13</sup> In addition, if an MPI application is only using the Sessions Model, the provided thread <sup>14</sup> support level returned by MPI\_QUERY\_THREAD is the same as that returned prior to <sup>15</sup> invocation of MPI\_INIT\_THREAD or MPI\_INIT. If the application also used the World <sup>16</sup> Model in some component of the application, MPI\_QUERY\_THREAD will return the level <sup>17</sup> of thread support returned by the original call to MPI\_INIT\_THREAD.

<sup>19</sup> Multiple threads completing the same request. A program in which two threads block, wait <sup>20</sup> ing on the same request, is erroneous. Similarly, the same request cannot appear in the
 <sup>21</sup> array of requests of two concurrent MPI\_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a
 <sup>22</sup> request can only be completed once. Any combination of wait or test that violates this rule
 <sup>23</sup> is erroneous.

Rationale. This restriction is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is possible to post a second wait on the same handle. With threads, an MPI\_WAIT{ANY|SOME|ALL} may be blocked without having nullified its request(s)

- so it becomes the user's responsibility to avoid using the same request in an MPI\_WAIT on another thread. This constraint also simplifies implementation, as only one thread will be blocked on any communication or I/O event. (*End of rationale.*)
- <sup>34</sup> Probe A receive call that uses source and tag values returned by a preceding call to <sup>35</sup> MPI\_PROBE or MPI\_IPROBE will receive the message matched by the probe call only <sup>36</sup> if there was no other matching receive after the probe and before that receive. In a multi-<sup>37</sup> threaded environment, it is up to the user to enforce this condition using suitable mutual <sup>38</sup> exclusion logic. This can be enforced by making sure that each communicator is used by <sup>39</sup> only one thread on each process. Alternatively, MPI\_MPROBE or MPI\_IMPROBE can be <sup>40</sup> used.
- 41

18

25

26

27

28

29

30

 $^{31}$ 

32

33

<sup>42</sup> Collective calls Matching of collective calls on a communicator, window, or file handle is
 <sup>43</sup> done according to the order in which the calls are issued at each process. If concurrent
 <sup>44</sup> threads issue such calls on the same communicator, window or file handle, it is up to the
 <sup>45</sup> user to make sure the calls are correctly ordered, using interthread synchronization.

- 46
- Advice to users. With three concurrent threads in each MPI process of a communica tor comm, it is allowed that thread A in each MPI process calls a collective operation

on comm, thread B calls a file operation on an existing filehandle that was formerly opened on comm, and thread C invokes one-sided operations on an existing window handle that was also formerly created on comm. (*End of advice to users.*)

*Rationale.* As specified in MPI\_FILE\_OPEN and MPI\_WIN\_CREATE, a file handle and a window handle inherit only the group of processes of the underlying communicator, 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.*)

**Error handlers** An error handler does not necessarily execute in the context of the thread that made the error-raising MPI call; the error handler may be executed by a thread that is distinct from the thread that will return the error code.

*Rationale.* The MPI implementation may be multithreaded, so that part of the communication protocol may execute on a thread that is distinct from the thread that made the MPI call. The design allows the error handler to be executed on the thread where the error is raised. (*End of rationale.*)

Interaction with signals and cancellations The outcome is undefined if a thread that executes an MPI call is cancelled (by another thread), or if a thread catches a signal while executing an MPI call. However, a thread of an MPI process may terminate, and may catch signals or be cancelled by another thread when not executing MPI calls.

*Rationale.* Few C library functions are signal safe, and many have cancellation points—points at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancel-safe" or "async-signal-safe"). (*End of rationale.*)

Advice to users. Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to sigwait for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

Advice to implementors. The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (*End of advice to implementors.*)

## 11.7 The Dynamic Process Model

The dynamic process model allows for the creation and cooperative termination of processes <sup>44</sup> after an MPI application has started. It provides a mechanism to establish communication <sup>45</sup> between the newly created processes and the existing MPI application. It also provides a <sup>46</sup> mechanism to establish communication between two existing MPI applications, even when <sup>47</sup> one did not "start" the other. <sup>48</sup>

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

11.7.1 Starting Processes

MPI applications may start new processes through an interface to an external process manager.

MPI\_COMM\_SPAWN starts MPI processes and establishes communication with them, returning an inter-communicator. MPI\_COMM\_SPAWN\_MULTIPLE starts several different 6 binaries (or the same binary with different arguments), placing them in the same MPI\_COMM\_WORLD and returning an inter-communicator.

MPI uses the group abstraction to represent processes. A process is identified by a (group, rank) pair.

10 11 12

39 40

41

42

43

44

4546

47

48

1

 $\mathbf{2}$ 

3

4

5

7

8

9

11.7.2 The Runtime Environment

13 The MPI\_COMM\_SPAWN and MPI\_COMM\_SPAWN\_MULTIPLE routines provide an inter-14face between MPI and the *runtime environment* of an MPI application. The difficulty is 15that there is an enormous range of runtime environments and application requirements, and 16MPI must not be tailored to any particular one.

17MPI assumes, implicitly, the existence of an environment in which an application runs. 18 It does not provide "operating system" services, such as a general ability to query what 19processes are running, to kill arbitrary processes, to find out properties of the runtime 20environment (how many processors, how much memory, etc.). Complex interaction of an 21MPI application with its runtime environment should be done through an environment-22specific API.

23At some low level, obviously, MPI must be able to interact with the runtime system,  $^{24}$ but the interaction is not visible at the application level and the details of the interaction 25are not specified by the MPI standard.

26In many cases, it is impossible to keep environment-specific information out of the MPI 27interface without seriously compromising MPI functionality. To permit applications to take 28advantage of environment-specific functionality, many MPI routines take an info argument 29that allows an application to specify environment-specific information. There is a tradeoff 30 between functionality and portability: applications that make use of environment-specific  $^{31}$ info are not portable.

32 MPI does not require the existence of an underlying "virtual machine" model, in which 33 there is a consistent global view of an MPI application and an implicit "operating system" 34managing resources and processes. For instance, processes spawned by one task may not 35 be visible to another; additional hosts added to the runtime environment by one process 36 may not be visible in another process; tasks spawned by different processes may not be 37 automatically distributed over available resources. 38

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.
- An attribute MPI\_UNIVERSE\_SIZE (See Section 11.10.1) 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.

## 11.8 Process Manager Interface

#### 11.8.1 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.

## 11.8.2 Starting Processes and Establishing Communication

The following routine starts a number of MPI processes and establishes communication with them, returning an inter-communicator.

Advice to users. It is possible in MPI to start an SPMD or MPMD application with a fixed number of processes after initialization 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.*)

# MPI\_COMM\_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array\_of\_errcodes)

IN	command	name of program to be spawned (string, significant only at root)	27 28 29
IN	argv	arguments to <b>command</b> (array of strings, significant only at root)	30 31
IN	maxprocs	maximum number of processes to start (integer, significant only at root)	32 33
ĮN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root)	34 35 36
IN	root	rank of process in which previous arguments are examined (integer)	37 38 39
IN	comm	intra-communicator containing group of spawning processes (handle)	40 41
OUT	intercomm	inter-communicator between original group and the newly spawned group (handle)	42 43
OUT	array_of_errcodes	one code per process (array of integer)	44 45 46
C binding			

```
1
     int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
\mathbf{2}
                    MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
3
                    int array_of_errcodes[])
4
     Fortran 2008 binding
5
     MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
6
                    array_of_errcodes, ierror)
7
          CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
8
          INTEGER, INTENT(IN) :: maxprocs, root
9
          TYPE(MPI_Info), INTENT(IN) :: info
10
          TYPE(MPI_Comm), INTENT(IN) :: comm
11
          TYPE(MPI_Comm), INTENT(OUT) :: intercomm
12
          INTEGER :: array_of_errcodes(*)
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     Fortran binding
16
     MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
17
                    ARRAY_OF_ERRCODES, IERROR)
18
          CHARACTER*(*) COMMAND, ARGV(*)
19
          INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
20
                     IERROR
21
          MPI_COMM_SPAWN tries to start maxprocs identical copies of the MPI program spec-
22
     ified by command, establishing communication with them and returning an inter-commu-
23
     nicator. The spawned processes are referred to as children. The children have their own
^{24}
     MPI_COMM_WORLD, which is separate from that of the parents. MPI_COMM_SPAWN is
25
     collective over comm, and also may not return until MPL INIT has been called in the chil-
26
     dren. Similarly, MPI_INIT in the children may not return until all parents have called
27
     MPI_COMM_SPAWN. In this sense, MPI_COMM_SPAWN in the parents and MPI_INIT in
28
     the children form a collective operation over the union of parent and child processes. The
29
     inter-communicator returned by MPI_COMM_SPAWN contains the parent processes in the
30
     local group and the child processes in the remote group. The ordering of processes in the
^{31}
     local and remote groups is the same as the ordering of the group of the comm in the parents
32
     and of MPI_COMM_WORLD of the children, respectively. This inter-communicator can be
33
     obtained in the children through the function MPI_COMM_GET_PARENT.
34
35
           Advice to users.
                             An implementation may automatically establish communication
36
          before MPI_INIT is called by the children. Thus, completion of MPI_COMM_SPAWN
37
          in the parent does not necessarily mean that MPI_INIT has been called in the children
38
           (although the returned inter-communicator can be used immediately). (End of advice
39
           to users.)
40
41
     The command argument The command argument is a string containing the name of a pro-
42
     gram to be spawned. The string is null-terminated in C. In Fortran, leading and trailing
43
     spaces are stripped. MPI does not specify how to find the executable or how the working
44
     directory is determined. These rules are implementation-dependent and should be appro-
45
     priate for the runtime environment.
46
47
           Advice to implementors. The implementation should use a natural rule for finding
```

```
Unofficial Draft for Comment Only
```

executables and determining working directories. For instance, a homogeneous system

with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation 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.*)

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 11.17 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};
```

or, if not everything is known at compile time:

MPI\_Comm\_spawn(command, argv, ...);

```
char *command;
char **argv;
command = "ocean";
argv=(char **)malloc(3 * sizeof(char *));
argv[0] = "-gridfile";
argv[1] = "ocean1.grd";
argv[2] = NULL;
MPI_Comm_spawn(command, argv, ...);
```

In Fortran:

```
      CHARACTER*25 command, argv(3)
      43

      command = 'ocean'
      44

      argv(1) = '-gridfile'
      45

      argv(2) = 'ocean1.grd'
      46

      argv(3) = ','
      47

      call MPI_COMM_SPAWN(command, argv, ...)
      48
```

**Unofficial Draft for Comment Only** 

 $\mathbf{2}$ 

## 516 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 Arguments are supplied to the program if this is allowed by the operating system. In  $\mathbf{2}$ C, the MPI\_COMM\_SPAWN argument argv differs from the argv argument of main in two 3 respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the 4 implementation and conventionally contains the name of the program (given by **command**).  $\mathbf{5}$ argv[1] of main corresponds to argv[0] in MPI\_COMM\_SPAWN, argv[2] of main to argv[1] 6 of MPI\_COMM\_SPAWN, etc. Passing an argv of MPI\_ARGV\_NULL to MPI\_COMM\_SPAWN  $\overline{7}$ results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the 8 name of the program. Second, argv of MPI\_COMM\_SPAWN must be null-terminated, so 9 that its length can be determined.

<sup>10</sup> If a Fortran implementation supplies routines that allow a program to obtain its ar-<sup>11</sup> guments, the arguments may be available through that mechanism. In C, if the operating <sup>12</sup> system does not support arguments appearing in argv of main(), the MPI implementation <sup>13</sup> may add the arguments to the argv that is passed to MPI\_INIT.

14

<sup>15</sup> The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn <sup>16</sup> maxprocs processes, it raises an error of class MPI\_ERR\_SPAWN.

<sup>17</sup> An implementation may allow the info argument to change the default behavior, such <sup>18</sup> that if the implementation is unable to spawn all maxprocs processes, it may spawn a <sup>19</sup> smaller number of processes instead of raising an error. In principle, the info argument <sup>20</sup> may specify an arbitrary set  $\{m_i : 0 \le m_i \le \text{maxprocs}\}$  of allowed values for the number <sup>21</sup> of processes spawned. The set  $\{m_i\}$  does not necessarily include the value maxprocs. If <sup>22</sup> an implementation is able to spawn one of these allowed numbers of processes,

<sup>23</sup> MPI\_COMM\_SPAWN returns successfully and the number of spawned processes, *m*, is given <sup>24</sup> by the size of the remote group of intercomm. If *m* is less than maxproc, reasons why the <sup>25</sup> other processes were not spawned are given in array\_of\_errcodes as described below. If it is <sup>26</sup> not possible to spawn one of the allowed numbers of processes, MPI\_COMM\_SPAWN raises <sup>27</sup> 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 11.8.4 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  $\{0, \ldots, N\}$ . 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 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 10.

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.

48

32

33

34

35

36

37

MPI does not specify the content of the info argument, except to reserve a number of special key values (see Section 11.8.4). The info argument is quite flexible and could even be used, for example, to specify the executable and its command-line arguments. In this case the command argument to MPI\_COMM\_SPAWN could be empty. The ability to do this follows from the fact that MPI does not specify how an executable is found, and the info argument can tell the runtime system where to "find" the executable "" (empty string). Of course a program that does this will not be portable across MPI implementations.

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.

The array\_of\_errcodes argument The array\_of\_errcodes is an array of length maxprocs in which MPI reports the status of each process that MPI was requested to start. If all maxprocs processes were spawned, array\_of\_errcodes is filled in with the value MPI\_SUCCESS. If only m ( $0 \le m < \text{maxprocs}$ ) processes are spawned, m of the entries will contain MPI\_SUCCESS and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class MPI\_ERR\_SPAWN if there was no error in the argument list. In C or Fortran, an application may pass MPI\_ERRCODES\_IGNORE if it is not interested in the error codes.

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.*)

MPI_COMM_GET_PARENT(parent)	29
	30
OUT   parent   the parent communicator (handle)	31
	32
C binding	33
<pre>int MPI_Comm_get_parent(MPI_Comm *parent)</pre>	34
Fortran 2008 binding	35
	36
MPI_Comm_get_parent(parent, ierror)	37
TYPE(MPI_Comm), INTENT(OUT) :: parent	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
	39
Fortran binding	40
MPI_COMM_GET_PARENT(PARENT, IERROR)	41
INTEGER PARENT, IERROR	42
If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE,	43
	44
MPI_COMM_GET_PARENT returns the "parent" inter-communicator of the current pro-	45
cess. This parent inter-communicator is created implicitly inside of MPI_INIT and is the	

same inter-communicator returned by SPAWN in the parents.

If the process was not spawned, MPI\_COMM\_GET\_PARENT returns MPI\_COMM\_NULL.

 $\mathbf{2}$ 

 $^{24}$ 

12		the parent communicator is PI_COMM_NULL.	s freed or disconnected, $MPI\_COMM\_GET\_PARENT$		
3 4 5 6 7 8 9	com to ti or M inva	municator. Calling MPI_CC he same inter-communicator. MPI_COMM_FREE will cause	<b>1_GET_PARENT</b> returns a handle to a single inter- DMM_GET_PARENT a second time returns a handle Freeing the handle with MPI_COMM_DISCONNECT other references to the inter-communicator to become ling MPI_COMM_FREE on the parent communicator users.)		
10 11 12 13 14	<i>Rationale.</i> 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. ( <i>End of rationale.</i> )				
15 16	11.8.3 5	Starting Multiple Executable	s and Establishing Communication		
17 18 19 20 21	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 follow- ing 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.				
22 23 24 25	MPI_CON	•	nt, array_of_commands, array_of_argv, ray_of_info, root, comm, intercomm,		
26 27	IN	count	number of commands (positive integer, significant only at root)		
28 29 30	IN	array_of_commands	programs to be executed (array of strings, significant only at root)		
31 32	IN	array_of_argv	arguments for <b>commands</b> (array of array of strings, significant only at root)		
33 34 25	IN	array_of_maxprocs	maximum number of processes to start for each command (array of integers, significant only at root)		
35 36 37 38	IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles, significant only at root)		
39 40	IN	root	rank of process in which previous arguments are examined (integer)		
41 42	IN	comm	intra-communicator containing group of spawning processes (handle)		
43 44 45	OUT	intercomm	inter-communicator between original group and the newly spawned group (handle)		
46 47	OUT	array_of_errcodes	one error code per process (array of integers)		
48	C bindir	ng			

```
1
int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
                                                                                   \mathbf{2}
              char **array_of_argv[], const int array_of_maxprocs[],
                                                                                   3
              const MPI_Info array_of_info[], int root, MPI_Comm comm,
              MPI_Comm *intercomm, int array_of_errcodes[])
                                                                                   4
Fortran 2008 binding
                                                                                   6
MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
              array_of_maxprocs, array_of_info, root, comm, intercomm,
              array_of_errcodes, ierror)
                                                                                   9
    INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
                                                                                   10
    CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
                                                                                   11
              array_of_argv(count, *)
                                                                                   12
    TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
                                                                                   13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   14
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
                                                                                   15
    INTEGER :: array_of_errcodes(*)
                                                                                   16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   17
                                                                                   18
Fortran binding
                                                                                   19
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
                                                                                   20
              ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
                                                                                   21
              ARRAY_OF_ERRCODES, IERROR)
    INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,
                                                                                   22
              INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
                                                                                   23
                                                                                   24
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
                                                                                   25
```

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.

#### Unofficial Draft for Comment Only

26

27

28

29

30

31

32

33

34

35

36

37

38

39 40

41

42

43

44

45

 $46 \\ 47$ 

1 The effect of setting individual elements of array\_of\_argv to MPI\_ARGV\_NULL is not defined.  $\mathbf{2}$ To specify arguments for some commands but not others, the commands without arguments 3 should have a corresponding argv whose first element is null ((char \*)0 in C and empty 4 string in Fortran). In Fortran at non-root processes, the count argument must be set to  $\mathbf{5}$ a value that is consistent with the provided array\_of\_argv although the content of these 6 arguments has no meaning for this operation.

7All of the spawned processes have the same MPI\_COMM\_WORLD. Their ranks in 8 MPI\_COMM\_WORLD correspond directly to the order in which the commands are specified 9 in MPI\_COMM\_SPAWN\_MULTIPLE. Assume that  $m_1$  processes are generated by the first 10 command,  $m_2$  by the second, etc. The processes corresponding to the first command have 11ranks  $0, 1, \ldots, m_1 - 1$ . The processes in the second command have ranks  $m_1, m_1 + 1, \ldots, m_1 + 1$ 12 $m_2 - 1$ . The processes in the third have ranks  $m_1 + m_2, m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 - 1$ , 13etc.

1415

16

18

21

25

30

Calling MPI\_COMM\_SPAWN multiple times would create many Advice to users. sets of children with different MPI\_COMM\_WORLDs whereas

MPI\_COMM\_SPAWN\_MULTIPLE creates children with a single MPI\_COMM\_WORLD, 17 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 19 use MPI\_COMM\_SPAWN\_MULTIPLE instead of calling MPI\_COMM\_SPAWN several 20times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes 22 spawned at the same time may be faster than communication between processes 23spawned separately. (End of advice to users.)  $^{24}$ 

The array\_of\_errcodes argument is a 1-dimensional array of size  $\sum_{i=1}^{count} n_i$ , where  $n_i$  is 26the *i*-th element of array\_of\_maxprocs. Command number *i* corresponds to the  $n_i$  contiguous 27slots 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 28MPI\_COMM\_SPAWN. 29

Example 11.18 Examples of array\_of\_argv in C and Fortran  $^{31}$ To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" and the program 32 "atmos" with argument "atmos.grd" in C: 33

```
34
             char *array_of_commands[2] = {"ocean", "atmos"};
35
             char **array_of_argv[2];
36
             char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
37
             char *argv1[] = {"atmos.grd", (char *)0};
38
             array_of_argv[0] = argv0;
39
             array_of_argv[1] = argv1;
40
             MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
41
42
     Here is how you do it in Fortran:
43
44
45
46
47
48
```

```
CHARACTER*25 commands(2), array_of_argv(2, 3)

commands(1) = 'ocean'

array_of_argv(1, 1) = '-gridfile'

array_of_argv(1, 2) = 'ocean1.grd'

array_of_argv(1, 3) = ' '

commands(2) = 'atmos'

array_of_argv(2, 1) = 'atmos.grd'

array_of_argv(2, 2) = ' '

call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)

11.8.4 Reserved Keys

The following keys are reserved. An implementation is not required to interpret these keys,

but if it does interpret the key, it must provide the functionality described.
```

- "host" Value is a hostname. The format of the hostname is determined by the implementation.
- "arch" Value is an architecture name. Valid architecture names and what they mean are determined by the implementation.
- "wdir" Value is the name of a directory on a machine on which the spawned process(es) execute(s). This directory is made the working directory of the executing process(es). The format of the directory name is determined by the implementation.
- "**path**" Value is a directory or set of directories where the implementation should look for the executable. The format of "**path**" is determined by the implementation.
- "file" Value is the name of a file in which additional information is specified. The format of the filename and internal format of the file are determined by the implementation.
- "mpi\_initial\_errhandler" Value is the name of an errhandler that will be set as the initial error handler. The "mpi\_initial\_errhandler" key can take the case insensitive values "mpi\_errors\_are\_fatal", "mpi\_errors\_abort", and "mpi\_errors\_return" representing the predefined MPI error handlers (MPI\_ERRORS\_ARE\_FATAL—the default, MPI\_ERRORS\_ABORT, and MPI\_ERRORS\_RETURN, respectively). Other, non-standard values may be supported by the implementation, which should document the resultant behavior.
- "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 commaseparated list of Fortran-90 triplets each of which specifies a set of integers and which together specify the set formed by the union of these sets. Negative values in this set and values greater than maxprocs are ignored. MPI will spawn the largest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant.

By Fortran-90 triplets, we mean:

1. a means a

1

2

3

5 6

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

46

#### 522 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
            2. a:b means a, a + 1, a + 2, ..., b
\mathbf{2}
            3. a:b:c means a, a + c, a + 2c, \ldots, a + ck, where for c > 0, k is the largest integer
3
               for which a + ck < b and for c < 0, k is the largest integer for which a + ck > b.
4
               If b > a then c must be positive. If b < a then c must be negative.
5
6
          Examples:
7
            1. a:b gives a range between a and b
8
            2. O:N gives full "soft" functionality
9
10
            3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows a power-of-two num-
11
               ber of processes.
12
            4. 2:10000:2 allows an even number of processes.
13
            5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.
14
15
     11.8.5 Spawn Example
16
17
18
     Example 11.19 Manager-worker Example Using MPI_COMM_SPAWN
19
20
     /* manager */
21
     #include "mpi.h"
22
     int main(int argc, char *argv[])
23
     {
24
        int world_size, universe_size, *universe_sizep, flag;
25
        MPI_Comm everyone;
                                         /* inter-communicator */
26
        char worker_program[100];
27
28
        MPI_Init(&argc, &argv);
29
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
30
        if (world_size != 1)
31
                                   error("Top heavy with management");
32
33
        MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
34
                             &universe_sizep, &flag);
35
        if (!flag) {
36
              printf("This MPI does not support UNIVERSE_SIZE. How many\n\
37
     processes total?");
38
              scanf("%d", &universe_size);
39
        } else universe_size = *universe_sizep;
40
        if (universe_size == 1) error("No room to start workers");
41
42
        /*
43
         * Now spawn the workers. Note that there is a run-time determination
44
         * of what type of worker to spawn, and presumably this calculation must
45
         * be done at run time and cannot be calculated before starting
46
         * the program. If everything is known when the application is
47
         * first started, it is generally better to start them all at once
48
         * in a single MPI_COMM_WORLD.
```

```
1
    */
                                                                                       2
   choose_worker_program(worker_program);
   MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
              MPI_INFO_NULL, 0, MPI_COMM_SELF, & everyone,
                                                                                       5
              MPI_ERRCODES_IGNORE);
                                                                                       6
   /*
    * Parallel code here. The communicator "everyone" can be used
    * to communicate with the spawned processes, which have ranks 0,...
                                                                                       9
                                                                                      10
    * MPI_UNIVERSE_SIZE-1 in the remote group of the inter-communicator
                                                                                      11
    * "everyone".
    */
                                                                                      12
                                                                                      13
                                                                                      14
   MPI_Finalize();
                                                                                      15
   return 0;
                                                                                      16
}
                                                                                      17
/* worker */
                                                                                      18
                                                                                      19
#include "mpi.h"
                                                                                      20
int main(int argc, char *argv[])
                                                                                      21
{
                                                                                      22
   int size;
                                                                                      23
   MPI_Comm parent;
                                                                                      ^{24}
   MPI_Init(&argc, &argv);
                                                                                      25
   MPI_Comm_get_parent(&parent);
                                                                                      26
   if (parent == MPI_COMM_NULL) error("No parent!");
                                                                                      27
   MPI_Comm_remote_size(parent, &size);
                                                                                      28
   if (size != 1) error("Something's wrong with the parent");
                                                                                      29
                                                                                      30
   /*
                                                                                      31
    * Parallel code here.
                                                                                      32
    \ast The manager is represented as the process with rank 0 in (the remote
                                                                                      33
    * group of) the parent communicator. If the workers need to communicate
                                                                                      34
    * among themselves, they can use MPI_COMM_WORLD.
                                                                                      35
    */
                                                                                      36
                                                                                      37
   MPI_Finalize();
                                                                                      38
   return 0:
                                                                                      39
}
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
11.9
       Establishing Communication
                                                                                      45
                                                                                      46
This section provides functions that establish communication between two sets of MPI
                                                                                      47
processes that do not share a communicator.
                                                                                      48
```

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.

<sup>7</sup> In each of these situations, MPI must establish communication channels where none ex-<sup>8</sup> isted before, and there is no parent/child relationship. The routines described in this <sup>9</sup> section establish communication between the two sets of processes by creating an MPI <sup>10</sup> inter-communicator, where the two groups of the inter-communicator are the original sets <sup>11</sup> 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.*)

23 24

26

35

36 37

38 39

40

41

42

43 44

45

46 47 48

18

19

20

21

22

1

2

3 4

5

6

<sup>25</sup> 11.9.1 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.
- 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.
  - Unofficial Draft for Comment Only

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.

- 1. Applications that do not rely on the ability to publish names are the most portable. Typically the port\_name must be transferred "by hand" from server to client.
- 2. Applications that use the MPI\_PUBLISH\_NAME mechanism are completely portable among implementations that provide this service. To be portable among all implementations, these applications should have a fall-back mechanism that can be used 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.

#### 11.9.2 Server Routines

A server makes itself available with two routines. First it must call MPI\_OPEN\_PORT to establish a port at which it may be contacted. Secondly it must call MPI\_COMM\_ACCEPT to accept connections from clients.

#### MPI\_OPEN\_PORT(info, port\_name)

IN	info	implementation-specific information on how to
		establish an address (handle)
OUT	port_name	newly established port (string)

C binding

int MPI\_Open\_port(MPI\_Info info, char \*port\_name)

```
Fortran 2008 binding
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
```

 $^{24}$ 

1	Tester his dis s
2	Fortran binding MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
3	INTEGER INFO, IERROR
4	CHARACTER*(*) PORT_NAME
5	CHARACIER*(*) FURI_NAME
6	This function establishes a network address, encoded in the port_name string, at which
7	the server will be able to accept connections from clients. port_name is supplied by the
8	system, possibly using information in the info argument.
9	MPI copies a system-supplied port name into port_name. port_name identifies the newly
10	opened port and can be used by a client to contact the server. The maximum size string
11	that may be supplied by the system is MPI_MAX_PORT_NAME.
12	Advice to users. The system copies the port name into port_name. The application
13 14	must pass a buffer of sufficient size to hold this value. ( <i>End of advice to users.</i> )
15	port_name is essentially a network address. It is unique within the communication
16	universe to which it belongs (determined by the implementation), and may be used by any
17	client within that communication universe. For instance, if it is an internet (host:port)
18	address, it will be unique on the internet. If it is a low level switch address on an IBM SP,
19	it will be unique to that SP.
20	
21	Advice to implementors. These examples are not meant to constrain implementa-
22	tions. A port_name could, for instance, contain a user name or the name of a batch
23	job, as long as it is unique within some well-defined communication domain. The
24	larger the communication domain, the more useful MPI's client/server functionality
25	will be. (End of advice to implementors.)
26	The precise form of the address is implementation-defined. For instance, an internet address
27	may be a host name or IP address, or anything that the implementation can decode into
28	an IP address. A port name may be reused after it is freed with MPI_CLOSE_PORT and
29	released by the system.
30	
31	Advice to implementors. Since the user may type in port_name by hand, it is useful
32	to choose a form that is easily readable and does not have embedded spaces. (End of
33	advice to implementors.)
34	
35	info may be used to tell the implementation how to establish the address. It may, and
36	usually will, be MPI_INFO_NULL in order to get the implementation defaults.
37	
38	MPI_CLOSE_PORT(port_name)
39	
40 41	IN port_name a port (string)
42	C binding
43	int MPI_Close_port(const char *port_name)
44	
45	Fortran 2008 binding
46	MPI_Close_port(port_name, ierror)
47	CHARACTER(LEN=*), INTENT(IN) :: port_name
48	INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_CLO CHA	binding SE_PORT(PORT_NAME, IF RACTER*(*) PORT_NAME EGER IERROR	ERROR)	1 2 3 4
This	function releases the ne	etwork address represented by port_name.	5 6
MPI_CO	MM_ACCEPT(port_name	e, info, root, comm, newcomm)	7 8
IN	port_name	port name (string, significant only at root)	9 10
IN	info	implementation-dependent information (handle, significant only at root)	10 11 12
IN	root	rank in comm of root node (integer)	13
IN	comm	intra-communicator over which call is collective (handle)	14 15 16
OUT	newcomm	inter-communicator with client as remote group (handle)	17 18 19
C bindi	ng		20
int MPI	-	nar *port_name, MPI_Info info, int root,	21
	MPI_Comm comm,	MPI_Comm *newcomm)	22 23
Fortran	2008 binding		23
<pre>MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)</pre>			25
CHARACTER(LEN=*), INTENT(IN) :: port_name			26
TYPE(MPI_Info), INTENT(IN) :: info INTEGER, INTENT(IN) :: root			27
TYPE(MPI_Comm), INTENT(IN) :: comm			28
TYPE(MPI_Comm), INTENT(OUT) :: newcomm			29 30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			31
Fortran binding			32
	J.	INFO, ROOT, COMM, NEWCOMM, IERROR)	33
	RACTER*(*) PORT_NAME		34
INT	EGER INFO, ROOT, COMM	1, NEWCOMM, IERROR	35
MPI	COMM ACCEPT estab	lishes communication with a client. It is collective over the	36
		an inter-communicator that allows communication with	37 38
the clien			39
The	port_name must have be	een established through a call to MPI_OPEN_PORT.	40
	can be used to provide o	lirectives that may influence the behavior of the ACCEPT	41
call.			42
11 0 2	Client Routines		43
			44 45
There is	only one routine on the	client side.	40
			48

#### 528 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
     MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)
2
       IN
                                             network address (string, significant only at root)
                 port_name
3
       IN
                 info
                                             implementation-dependent information (handle,
4
                                             significant only at root)
5
6
       IN
                 root
                                             rank in comm of root node (integer)
7
       IN
                                             intra-communicator over which call is collective
                 comm
8
                                             (handle)
9
       OUT
                 newcomm
                                             inter-communicator with server as remote group
10
                                             (handle)
11
12
13
     C binding
14
     int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
                    MPI_Comm comm, MPI_Comm *newcomm)
15
16
     Fortran 2008 binding
17
     MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
18
          CHARACTER(LEN=*), INTENT(IN) :: port_name
19
          TYPE(MPI_Info), INTENT(IN) :: info
20
          INTEGER, INTENT(IN) :: root
21
          TYPE(MPI_Comm), INTENT(IN) :: comm
22
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
     Fortran binding
26
     MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
27
          CHARACTER*(*) PORT_NAME
          INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
28
29
         This routine establishes communication with a server specified by port_name. It is
30
     collective over the calling communicator and returns an inter-communicator in which the
^{31}
     remote group participated in an MPI_COMM_ACCEPT.
32
          If the named port does not exist (or has been closed), MPI_COMM_CONNECT raises
33
     an error of class MPI_ERR_PORT.
34
       If the port exists, but does not have a pending MPI_COMM_ACCEPT, the connection
35
     attempt will eventually time out after an implementation-defined time, or succeed when
36
     the server calls MPI_COMM_ACCEPT. In the case of a time out, MPI_COMM_CONNECT
37
     raises an error of class MPI_ERR_PORT.
38
39
           Advice to implementors.
                                      The time out period may be arbitrarily short or long.
40
           However, a high-quality implementation will try to queue connection attempts so
41
           that a server can handle simultaneous requests from several clients. A high-quality
42
           implementation may also provide a mechanism, through the info arguments to
43
           MPI_OPEN_PORT, MPI_COMM_ACCEPT, and/or MPI_COMM_CONNECT, for the
44
           user to control timeout and queuing behavior. (End of advice to implementors.)
45
46
          MPI provides no guarantee of fairness in servicing connection attempts. That is, connec-
47
     tion attempts are not necessarily satisfied in the order they were initiated and competition
48
```

from other connection attempts may prevent a particular connection attempt from being satisfied.

port\_name is the address of the server. It must be the same as the name returned by MPI\_OPEN\_PORT on the server. Some freedom is allowed here. If there are equivalent forms of port\_name, an implementation may accept them as well. For instance, if port\_name is (hostname:port), an implementation may accept (ip\_address:port) as well.

#### 11.9.4 Name Publishing

The routines in this section provide a mechanism for publishing names. A (service\_name, port\_name) pair is published by the server, and may be retrieved by a client using the service\_name only. An MPI implementation defines the *scope* of the service\_name, that is, the domain over which the service\_name can be retrieved. If the domain is the empty set, that is, if no client can retrieve the information, then we say that name publishing is not supported. Implementations should document how the scope is determined. High-quality implementations will give some control to users through the info arguments to name publishing functions. Examples are given in the descriptions of individual functions.

			19	
MP	MPI_PUBLISH_NAME(service_name, info, port_name)			
	· · · · · · ·			
11	service_name	a service name to associate with the port (string)	21	
II	N info	implementation-specific information (handle)	22	
11	J port_name	a port name (string)	23	
	. –		24	
C	in dia m		25	
	pinding		26	
int		*service_name, MPI_Info info,	27	
	const char *port_name)			
Fortran 2008 binding				
	MPI_Publish_name(service_name, info, port_name, ierror)			
rir 1			30 31	
		) :: service_name, port_name	31	
	TYPE(MPI_Info), INTENT(IN) :: info			
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
Б				
	tran binding		35	
MPI	_PUBLISH_NAME(SERVICE_NAME, ]	INFO, PORT_NAME, IERROR)	36	
	CHARACTER*(*) SERVICE_NAME,	PORT_NAME	37	
	INTEGER INFO, IERROR			
			38	
	This routine publishes the pair (	port_name, service_name) so that an application may	39	

This routine publishes the pair (port\_name, service\_name) so that an application may retrieve a system-supplied port\_name using a well-known service\_name.

The implementation must define the *scope* of a published service name, that is, the domain over which the service name is unique, and conversely, the domain over which the (port name, service name) pair may be retrieved. For instance, a service name may be unique to a job (where job is defined by a distributed operating system or batch scheduler), unique to a machine, or unique to a Kerberos realm. The scope may depend on the info argument to MPI\_PUBLISH\_NAME.

MPI permits publishing more than one service\_name for a single port\_name. On the <sup>47</sup> other hand, if service\_name has already been published within the scope determined by info, <sup>48</sup>

#### Unofficial Draft for Comment Only

 $45 \\ 46$ 

the behavior of MPI\_PUBLISH\_NAME is undefined. An MPI implementation may, through
 a mechanism in the info argument to MPI\_PUBLISH\_NAME, provide a way to allow multiple
 servers with the same service in the same scope. In this case, an implementation-defined
 policy will determine which of several port names is returned by MPI\_LOOKUP\_NAME.

<sup>5</sup> Note that while service\_name has a limited scope, determined by the implementation,
 <sup>6</sup> port\_name always has global scope within the communication universe used by the imple <sup>7</sup> 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.*)

a port name (string)

```
25
26
27
```

28

29

30

31 32

33 34

35

36

37

38

42

10

11

12

13

14

15

16

 MPI\_UNPUBLISH\_NAME(service\_name, info, port\_name)

 IN
 service\_name

 IN
 info

 info
 implementation-specific information (handle)

IN port\_name

```
C binding
int MPI_Unpublish_name(const char *service_name, MPI_Info info,
const char *port_name)
Fortran 2008 binding
```

```
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
```

```
43
44 Fortran binding
```

```
    MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR
```

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 how to publish names, the implementation may require that info passed to MPI\_UNPUBLISH\_NAME contain information to tell the implementation how to unpublish a name.

MPI_LOOKUP_NAME(service_name, info, port_name)			13
IN	service_name	a service name (string)	14
IN	info	implementation-specific information (handle)	15
OUT	port_name	a port name (string)	16
001	port_name	a port name (string)	17
Chindin	-		18 19
C binding		ervice_name, MPI_Info info,	20
IIIC MFI_L	char *port_name)	sivice_name, MF1_1110 1110,	20
			22
	2008 binding		23
	p_name(service_name, info		24
CHARACTER(LEN=*), INTENT(IN) :: service_name			25
TYPE(MPI_Info), INTENT(IN) :: info			26
CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror			27
			28
Fortran binding			29
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)			30
CHARACTER*(*) SERVICE_NAME, PORT_NAME			31 32
INTEG	INTEGER INFO, IERROR		
This f	unction retrieves a port_name	published by MPI_PUBLISH_NAME with	33 34
		een published, it raises an error in the error class	34 35
MPI_ERR_NAME. The application must supply a port_name buffer large enough to hold the			36
largest pos	sible port name (see discussio	n above under MPI_OPEN_PORT).	37
If an	If an implementation allows multiple entries with the same service_name within the		
-		osen in a way determined by the implementation.	39
	If the info argument was used with MPI_PUBLISH_NAME to tell the implementation $_{40}$		
how to pul	how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME. $_{41}$		

#### 11.9.5 Reserved Key Values

The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.

"ip\_port" Value contains IP port number at which to establish a port. (Reserved for MPI\_OPEN\_PORT only).

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8 9

10

11 12

41 42

 $\frac{43}{44}$ 

45

 $46 \\ 47$ 

```
1
     "ip_address" Value contains IP address at which to establish a port. If the address is not a
\mathbf{2}
          valid IP address of the host on which the MPI_OPEN_PORT call is made, the results
3
          are undefined. (Reserved for MPI_OPEN_PORT only).
4
5
     11.9.6 Client/Server Examples
6
7
     Example 11.20 Simplest Example—Completely Portable.
8
     The following example shows the simplest way to use the client/server interface. It does
9
     not use service names at all.
10
     On the server side:
11
12
13
         char myport[MPI_MAX_PORT_NAME];
14
         MPI_Comm intercomm;
15
         /* ... */
16
         MPI_Open_port(MPI_INFO_NULL, myport);
17
         printf("port name is: %s\n", myport);
18
19
         MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
20
         /* do something with intercomm */
21
     The server prints out the port name to the terminal and the user must type it in when
22
     starting up the client (assuming the MPI implementation supports stdin such that this
23
     works). On the client side:
24
25
         MPI_Comm intercomm;
26
          char name[MPI_MAX_PORT_NAME];
27
         printf("enter port name: ");
28
         gets(name);
29
         MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
30
31
     Example 11.21 Ocean/Atmosphere—Relies on Name Publishing
32
         In this example, the "ocean" application is the "server" side of a coupled ocean-
33
     atmosphere climate model. It assumes that the MPI implementation publishes names.
34
35
36
         MPI_Open_port(MPI_INFO_NULL, port_name);
37
         MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);
38
39
         MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
40
          /* do something with intercomm */
41
         MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
42
43
44
     On the client side:
45
46
         MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
47
         MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
48
                              &intercomm);
```

#### Example 11.22 Simple Client-Server Example

This is a simple example; the server accepts only a single connection at a time and serves that connection until the client requests to be disconnected. The server is a single process.

Here is the server. It accepts a single connection and then processes data until it receives a message with tag 1. A message with tag 0 tells the server to exit.

```
#include "mpi.h"
                                                                                        7
int main(int argc, char *argv[])
                                                                                        8
{
                                                                                        9
    MPI_Comm client;
                                                                                        10
    MPI_Status status;
                                                                                        11
    char port_name[MPI_MAX_PORT_NAME];
                                                                                        12
    double buf[MAX_DATA];
                                                                                        13
    int
            size, again;
                                                                                        14
                                                                                        15
    MPI_Init(&argc, &argv);
                                                                                        16
    MPI_Comm_size(MPI_COMM_WORLD, &size);
                                                                                        17
    if (size != 1) error(FATAL, "Server too big");
                                                                                        18
    MPI_Open_port(MPI_INFO_NULL, port_name);
                                                                                        19
    printf("server available at %s\n", port_name);
                                                                                        20
    while (1) {
                                                                                        21
        MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                                                                                        22
                          &client);
                                                                                        23
        again = 1;
                                                                                        ^{24}
        while (again) {
                                                                                        25
             MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
                                                                                        26
                       MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
                                                                                        27
             switch (status.MPI_TAG) {
                                                                                        28
                case 0: MPI_Comm_free(&client);
                                                                                        29
                          MPI_Close_port(port_name);
                                                                                        30
                          MPI_Finalize();
                                                                                        31
                          return 0;
                                                                                        32
                  case 1: MPI_Comm_disconnect(&client);
                                                                                        33
                          again = 0;
                                                                                        34
                          break;
                                                                                        35
                 case 2: /* do something */
                                                                                        36
                  . . .
                                                                                        37
                 default:
                                                                                        38
                          /* Unexpected message type */
                                                                                        39
                          MPI_Abort(MPI_COMM_WORLD, 1);
                                                                                        40
                 }
                                                                                        41
             }
                                                                                        42
        }
                                                                                        43
}
                                                                                        44
                                                                                        45
Here is the client.
                                                                                        46
```

1

 $\mathbf{2}$ 

3

4

5 6

## 534 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
     #include "mpi.h"
\mathbf{2}
     int main(int argc, char **argv)
3
     ſ
4
         MPI_Comm server;
5
         double buf [MAX_DATA];
6
         char port_name[MPI_MAX_PORT_NAME];
7
8
         MPI_Init(&argc, &argv);
9
         strcpy(port_name, argv[1]);/* assume server's name is cmd-line arg */
10
11
         MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
12
                            &server);
13
14
         while (!done) {
15
              tag = 2; /* Action to perform */
16
              MPI_Send(buf, n, MPI_DOUBLE, 0, tag, server);
17
              /* etc */
18
              }
19
         MPI_Send(buf, 0, MPI_DOUBLE, 0, 1, server);
20
         MPI_Comm_disconnect(&server);
21
         MPI_Finalize();
22
         return 0;
23
     }
^{24}
25
```

## 11.10 Other Functionality

11.10.1 Universe Size

<sup>29</sup> Many "dynamic" MPI applications are expected to exist in a static runtime environment, <sup>30</sup> in which resources have been allocated before the application is run. When a user (or <sup>31</sup> possibly a batch system) runs one of these quasi-static applications, she will usually specify <sup>32</sup> a number of processes to start and a total number of processes that are expected. An <sup>33</sup> application simply needs to know how many slots there are, i.e., how many processes it <sup>34</sup> should spawn.

MPI provides an attribute on MPI\_COMM\_WORLD, MPI\_UNIVERSE\_SIZE, that allows the 35 application to obtain this information in a portable manner. This attribute indicates the 36 total number of processes that are expected. In Fortran, the attribute is the integer value. 37 In C, the attribute is a pointer to the integer value. An application typically subtracts 38 the size of MPI\_COMM\_WORLD from MPI\_UNIVERSE\_SIZE to find out how many processes it 39 should spawn. MPI\_UNIVERSE\_SIZE is initialized in MPI\_INIT and is not changed by MPI. If 40defined, it has the same value on all processes of MPI\_COMM\_WORLD. MPI\_UNIVERSE\_SIZE 41 is determined by the application startup mechanism in a way not specified by MPI. (The 42size of MPI\_COMM\_WORLD is another example of such a parameter.) 43

44

26 27

28

45

47

48

46

Possibilities for how  $\mathsf{MPI\_UNIVERSE\_SIZE}$  might be set include

- $\bullet$  A <code>-universe\_size</code> 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.

 $\mathbf{2}$ 

 $\overline{7}$ 

- An environment variable set by the user.
- Extra information passed to MPI\_COMM\_SPAWN through the info argument.

An implementation must document how MPI\_UNIVERSE\_SIZE is set. An implementation may not support the ability to set MPI\_UNIVERSE\_SIZE, in which case the attribute MPI\_UNIVERSE\_SIZE is not set.

MPI\_UNIVERSE\_SIZE is a recommendation, not necessarily a hard limit. For instance, some implementations may allow an application to spawn 50 processes per processor, if they are requested. However, it is likely that the user only wants to spawn one process per processor.

MPI\_UNIVERSE\_SIZE is assumed to have been specified when an application was started, and is in essence a portable mechanism to allow the user to pass to the application (through 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.

#### 11.10.2 Singleton MPI Initialization

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, MPI\_INIT\_THREAD, or MPI\_SESSION\_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. Special coordination is required to start MPI processes belonging to the same MPI\_COMM\_WORLD in the case of the World Model, or the same "mpi://world" process set in the Sessions Model. 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.

Considering the World Model, 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.

1 A high-quality implementation will try to create a singleton MPI process and not raise  $\mathbf{2}$ an error. 3 (End of advice to implementors.) 4 511.10.3 MPI\_APPNUM 6  $\overline{7}$ There is a predefined attribute MPI\_APPNUM of MPI\_COMM\_WORLD. In Fortran, the at-8 tribute is an integer value. In C, the attribute is a pointer to an integer value. If a process 9 was spawned with MPI\_COMM\_SPAWN\_MULTIPLE, MPI\_APPNUM is the command number 10 that generated the current process. Numbering starts from zero. If a process was spawned 11with MPI\_COMM\_SPAWN, it will have MPI\_APPNUM equal to zero. 12Additionally, if the process was not started by a spawn call, but by an implementation-13 specific startup mechanism that can handle multiple process specifications, MPI\_APPNUM 14should be set to the number of the corresponding process specification. In particular, if it 15is started with 16mpiexec spec0 [: spec1 : spec2 : ...] 1718 MPI\_APPNUM should be set to the number of the corresponding specification. 19If an application was not spawned with MPI\_COMM\_SPAWN or 20MPI\_COMM\_SPAWN\_MULTIPLE, and MPI\_APPNUM does not make sense in the context of 21the implementation-specific startup mechanism, MPI\_APPNUM is not set. 22 MPI implementations may optionally provide a mechanism to override the value of 23MPI\_APPNUM through the info argument. MPI reserves the following key for all SPAWN 24calls. 2526"appnum" Value contains an integer that overrides the default value for MPI\_APPNUM in 27the child. 2829 *Rationale.* When a single application is started, it is able to figure out how many pro-30 cesses there are by looking at the size of MPI\_COMM\_WORLD. An application consisting 31of multiple SPMD sub-applications has no way to find out how many sub-applications 32 there are and to which sub-application the process belongs. While there are ways to 33 figure it out in special cases, there is no general mechanism. MPI\_APPNUM provides 34 such a general mechanism. (End of rationale.) 35 36 **Releasing Connections** 11.10.4 37 Before a client and server connect, they are independent MPI applications. An error in one 38 does not affect the other. After establishing a connection with MPI\_COMM\_CONNECT and 39 MPI\_COMM\_ACCEPT, an error in one may affect the other. It is desirable for a client and 40 server to be able to disconnect, so that an error in one will not affect the other. Similarly, 41 it might be desirable for a parent and child to disconnect, so that errors in the child do not 42affect the parent, or vice-versa. 43 44 • Two processes are **connected** if there is a communication path (direct or indirect) 45 between them. More precisely: 46

1. Two processes are connected if

47

(a) they both belong to the same communicator (inter- or intra-, including MPI_COMM_WORLD) or	1 2
(b) they have previously belonged to a communicator that was freed with MPI_COMM_FREE instead of MPI_COMM_DISCONNECT or	3 4
(c) they both belong to the group of the same window or filehandle.	5
2. If A is connected to B and B to C, then A is connected to C.	6
2. If A is connected to b and b to c, then A is connected to c.	7
• Two processes are <b>disconnected</b> (also <b>independent</b> ) if they are not connected.	8 9
• By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.	10 11 12
• 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.	13 14 15 16
The following additional rules apply to MPI routines in other chapters:	17 18
• MPI_FINALIZE is collective over a set of connected processes.	19
• MPI_ABORT does not abort independent processes. It may abort all processes in	20
the caller's MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may	21
abort connected processes as well, though it makes a "best attempt" to abort only	22
the processes in comm.	23 24
• If a process terminates without calling MPI_FINALIZE, independent processes are not	25
• If a process terminates without caning MPI_FINALIZE, independent processes are not affected but the effect on connected processes is not defined.	26
anceded but the chect on connected processes is not defined.	27
Advice to implementors. In practice, it may be difficult to distinguish between an	28
MPI process failure and an erroneous program that terminates without calling an	29
MPI finalization function: an implementation that defines semantics for process fail-	30
ure management may have to exhibit the behavior defined for MPI process failures	31
with such erroneous programs. A high quality implementation should exhibit a dif- forent behavior for erroneous programs and MPI process failures. (End of advise to	32 33
ferent behavior for erroneous programs and MPI process failures. ( <i>End of advice to implementors.</i> )	34
	35
	36
MDL COMM DISCONNECT (comm)	37
MPI_COMM_DISCONNECT(comm)	38
INOUT comm communicator (handle)	39
	40
C binding	41
<pre>int MPI_Comm_disconnect(MPI_Comm *comm)</pre>	42 43
Fortran 2008 binding	43 44
MPI_Comm_disconnect(comm, ierror)	45
TYPE(MPI_Comm), INTENT(INOUT) :: comm	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
Fortran hinding	48

Fortran binding

Unofficial Draft for Comment Only

#### 1 MPI\_COMM\_DISCONNECT(COMM, IERROR) $\mathbf{2}$

INTEGER COMM, IERROR

3

4

5

6

7

8

9

10

11

12

13 14

15

16

17

18

19 20

21

22

23 $^{24}$ 

2526

42

43

This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI\_COMM\_NULL. It is a collective operation.

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.

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.

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. (End of advice to users.)

Rationale. 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. (End of rationale.)

Another Way to Establish MPI Communication 11.10.5

27MPI\_COMM\_JOIN(fd, intercomm) 2829 IN fd socket file descriptor 30 OUT new inter-communicator (handle) intercomm  $^{31}$ 32 C binding 33 int MPI\_Comm\_join(int fd, MPI\_Comm \*intercomm) 34 35 Fortran 2008 binding 36 MPI\_Comm\_join(fd, intercomm, ierror) 37 INTEGER, INTENT(IN) :: fd 38 TYPE(MPI\_Comm), INTENT(OUT) :: intercomm 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40

Fortran binding 41

MPI\_COMM\_JOIN(FD, INTERCOMM, IERROR) INTEGER FD, INTERCOMM, IERROR

44MPI\_COMM\_JOIN is intended for MPI implementations that exist in an environment 45supporting the Berkeley Socket interface [50, 56]. Implementations that exist in an environ-46ment not supporting Berkeley Sockets should provide the entry point for MPI\_COMM\_JOIN 47and should return MPI\_COMM\_NULL. 48

This call creates an inter-communicator 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 inter-communicator, and for nothing else. Upon return from MPI\_COMM\_JOIN, the file descriptor will be open and quiescent (see below).

If MPI is unable to create an inter-communicator, but is able to leave the socket in its 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 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.

Advice to implementors. MPI is free to use any available communication path(s) for MPI messages in the new communicator; the socket is only used for the initial handshaking. (*End of advice to implementors.*)

MPI\_COMM\_JOIN uses non-MPI communication to do its work. The interaction of non-MPI communication with pending MPI communication is not defined. Therefore, the result of calling MPI\_COMM\_JOIN on two connected processes (see Section 11.10.4 for the definition of connected) is undefined.

The returned communicator may be used to establish MPI communication with additional processes, through the usual MPI communicator creation mechanisms.

#### Unofficial Draft for Comment Only

 $\mathbf{5}$ 

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

 $45 \\ 46$ 



# Chapter 12

# **One-Sided** Communications

# 12.1 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.

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 12.4. Both models support several synchronization calls to support 8 different synchronization styles.

<sup>9</sup> The design of the RMA functions allows implementors to take advantage of fast or <sup>10</sup> asynchronous communication mechanisms provided by various platforms, such as coherent <sup>11</sup> or noncoherent shared memory, DMA engines, hardware-supported put/get operations, and <sup>12</sup> communication coprocessors. The most frequently used RMA communication mechanisms <sup>13</sup> can be layered on top of message-passing. However, certain RMA functions might need <sup>14</sup> support for asynchronous communication agents in software (handlers, threads, etc.) in a <sup>15</sup> distributed memory environment.

<sup>16</sup> We shall denote by **origin** the process that performs the call, and by **target** the <sup>17</sup> process in which the memory is accessed. Thus, in a put operation, source = origin and <sup>18</sup> destination = target; in a get operation, source = target and destination = origin.

The use of terms such as nonblocking and local in this chapter follow the usage in
 MPI-3.1, and this chapter has not been updated to follow the definitions in Section 2.4.
 The MPI Forum intends to update this chapter in a subsequent version of the MPI standard to follow the definitions in Section 2.4.

23 24

25

# 12.2 Initialization

<sup>26</sup> MPI provides the following window initialization functions: MPI\_WIN\_CREATE,

<sup>27</sup> MPI\_WIN\_ALLOCATE, MPI\_WIN\_ALLOCATE\_SHARED, and

<sup>26</sup> MPI\_WIN\_CREATE\_DYNAMIC, which are collective on an intra-communicator.

<sup>29</sup> MPI\_WIN\_CREATE allows each process to specify a "window" in its memory that is made <sup>30</sup> accessible to accesses by remote processes. The call returns an opaque object that represents <sup>31</sup> the group of processes that own and access the set of windows, and the attributes of each <sup>32</sup> window, as specified by the initialization call. MPI\_WIN\_ALLOCATE differs from

<sup>34</sup> MPI\_WIN\_CREATE in that the user does not pass allocated memory;

<sup>35</sup> MPI\_WIN\_ALLOCATE returns a pointer to memory allocated by the MPI implementation. <sup>35</sup> MPI\_WIN\_ALLOCATE\_SHARED differs from MPI\_WIN\_ALLOCATE in that the allocated <sup>36</sup> memory can be accessed from all processes in the window's group with direct load/store <sup>37</sup> instructions. Some restrictions may apply to the specified communicator.

<sup>39</sup> MPI\_WIN\_CREATE\_DYNAMIC creates a window that allows the user to dynamically control which memory is exposed by the window.

- 41
- 42

- 44
- 45 46
- 47
- 48

12.2.1	Window Creation		1
			2 3
MPI V	/IN CREATE(base, size,	disp_unit, info, comm, win)	4
IN	base	. ,	5
		initial address of window (choice)	6
IN	size	size of window in bytes (non-negative integer)	7
IN	disp_unit	local unit size for displacements, in bytes (positive integer)	8 9
IN	info	info argument (handle)	10 11
IN	comm	intra-communicator (handle)	12
OUT	win	window object (handle)	13
001	VVIII	window object (nandic)	14
C bin	ding		15
	0	base, MPI_Aint size, int disp_unit, MPI_Info info,	16
		m, MPI_Win *win)	17
int MI	T Win crosto c(void	*base, MPI_Aint size, MPI_Aint disp_unit,	18 19
IIIC M		o, MPI_Comm comm, MPI_Win *win)	20
_			
Fortran 2008 binding MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)			22
		, disp_unit, info, comm, win, ierror) ), ASYNCHRONOUS :: base	23
		ESS_KIND), INTENT(IN) :: size	24
	INTEGER, INTENT(IN) :: disp_unit		
	PE(MPI_Info), INTEN	-	26
	PE(MPI_Comm), INTEN		27 28
TY	PE(MPI_Win), INTENT	(OUT) :: win	20
II	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
MPI_Wi	.n_create(base, size	, disp_unit, info, comm, win, ierror)	31
	TYPE(*), DIMENSION(), ASYNCHRONOUS :: base		
II	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit		
	TYPE(MPI_Info), INTENT(IN) :: info		
	TYPE(MPI_Comm), INTENT(IN) :: comm		
	PE(MPI_Win), INTENT		36 37
ΤŢ	ITEGER, OPTIONAL, IN	IENI(UUI) :: lerror	38
Fortra	n binding		39
	· ·	, DISP_UNIT, INFO, COMM, WIN, IERROR)	40
	<pre>Sype&gt; BASE(*)</pre>		41
	ITEGER(KIND=MPI_ADDR		42
ΤΓ	ILGER DISP_UNII, IN	FO, COMM, WIN, IERROR	43

This is a collective call executed by all processes in the group of comm. It returns 44a window object that can be used by these processes to perform RMA operations. Each 4546process 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 4748 base. In C, base is the starting address of a memory region. In Fortran, one can pass the

# **Unofficial Draft for Comment Only**

1	first element of a memory region or a whole array, which must be 'simply contiguous' (for
2	'simply contiguous,' see also Section 19.1.12). A process may elect to expose no memory
3	by specifying $size = 0$ .
4	The displacement unit argument is provided to facilitate address arithmetic in RMA
5	operations: the target displacement argument of an RMA operation is scaled by the factor
6	disp_unit specified by the target process, at window creation.
7	
8	Rationale. The window size is specified using an address-sized integer, rather than a
9	basic integer type, to allow windows that span more memory than can be described
10	with a basic integer type. (End of rationale.)
11	
12	Advice to users. Common choices for disp_unit are 1 (no scaling), and (in C syntax)
13	sizeof(type), for a window that consists of an array of elements of type type. The
14	latter choice will allow one to use array indices in RMA calls, and have those scaled
15	correctly to byte displacements, even in a heterogeneous environment. (End of advice
16	to users.)
17	
18	The info argument provides optimization hints to the runtime about the expected usage
19	pattern of the window. The following info keys are predefined:
	patient of the window The following the help are pretented.
20	"no_locks"—if set to true, then the implementation may assume that passive target synchro-
21	nization (i.e., MPI_WIN_LOCK, MPI_WIN_LOCK_ALL) will not be used on the given
22	window. This implies that this window is not used for 3-party communication, and
23 24	RMA can be implemented with no (less) asynchronous agent activity at this process.
25	"accumulate_ordering"—controls the ordering of accumulate operations at the target. See
26	Section 12.7.2 for details.
27	
28	"accumulate_ops"—if set to "same_op", the implementation will assume that all concurrent
29	accumulate calls to the same target address will use the same operation. If set to
30	"same_op_no_op", then the implementation will assume that all concurrent accumulate
31	calls to the same target address will use the same operation or MPI_NO_OP. This can
32	eliminate the need to protect access for certain operation types where the hardware
33	can guarantee atomicity. The default is "same_op_no_op".
34	
35	"same_size"—if set to true, then the implementation may assume that the argument size is
36	identical on all processes, and that all processes have provided this info key with the
37	same value.
38	
39	"same_disp_unit"—if set to true, then the implementation may assume that the argument
40	disp_unit is identical on all processes, and that all processes have provided this info
40	key with the same value.
42	Advice to users. The info query mechanism described in Section 12.2.7 can be used
43	to query the specified info arguments for windows that have been passed to a library.
44	It is recommended that libraries check attached info keys for each passed window.
45	(End of advice to users.)
46	
47	The various processes in the group of comm may specify completely different target
48	windows, in location, size, displacement units, and info arguments. As long as all the get,

# **Unofficial Draft for Comment Only**

put and accumulate accesses to a particular process fit their specific target window this should pose no problem. The same area in memory may appear in multiple windows, each associated with a different window object. However, concurrent communications to distinct, overlapping windows may lead to undefined results.

*Rationale.* The reason for specifying the memory that may be accessed from another process in an RMA operation is to permit the programmer to specify what memory can be a target of RMA operations and for the implementation to enforce that specification. For example, with this definition, a server process can safely allow a client process to use RMA operations, knowing that (under the assumption that the MPI implementation does enforce the specified limits on the exposed memory) an error in the client cannot affect any memory other than what was explicitly exposed. (*End of rationale.*)

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 9.2) 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.*)

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13 14

15

16

17

18

19 20

21

22

23

24

25

26

27

28

29

30 31

32 33

34

35

```
1
     12.2.2 Window That Allocates Memory
\mathbf{2}
3
4
     MPI_WIN_ALLOCATE(size, disp_unit, info, comm, baseptr, win)
5
       IN
                size
                                            size of window in bytes (non-negative integer)
6
       IN
                disp_unit
                                            local unit size for displacements, in bytes (positive
7
8
                                            integer)
9
       IN
                 info
                                            info argument (handle)
10
       IN
                comm
                                            intra-communicator (handle)
11
       OUT
                                            initial address of window (choice)
                 baseptr
12
13
       OUT
                                            window object returned by call (handle)
                win
14
15
     C binding
16
     int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
17
                    MPI_Comm comm, void *baseptr, MPI_Win *win)
18
     int MPI_Win_allocate_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,
19
                    MPI_Comm comm, void *baseptr, MPI_Win *win)
20
21
     Fortran 2008 binding
22
     MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
23
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
25
         INTEGER, INTENT(IN) :: disp_unit
26
         TYPE(MPI_Info), INTENT(IN) :: info
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(C_PTR), INTENT(OUT) :: baseptr
29
         TYPE(MPI_Win), INTENT(OUT) :: win
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
33
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
34
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
         TYPE(MPI_Info), INTENT(IN) :: info
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         TYPE(C_PTR), INTENT(OUT) :: baseptr
37
         TYPE(MPI_Win), INTENT(OUT) :: win
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     Fortran binding
41
     MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
42
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
43
         INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
44
45
         This is a collective call executed by all processes in the group of comm. On each
46
     process, it allocates memory of at least size bytes, returns a pointer to it, and returns a
47
     window object that can be used by all processes in comm to perform RMA operations. The
48
     returned memory consists of size bytes local to each process, starting at address baseptr
```

and is associated with the window as if the user called MPI\_WIN\_CREATE on existing memory. The size argument may be different at each process and size = 0 is valid; however, a library might allocate and expose more memory in order to create a fast, globally symmetric allocation. The discussion of and rationales for MPI\_ALLOC\_MEM and MPI\_FREE\_MEM in Section 9.2 also apply to MPI\_WIN\_ALLOCATE; in particular, see the rationale in Section 9.2 for an explanation of the type used for baseptr.

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:

```
INTERFACE MPI_WIN_ALLOCATE
   SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
                                WIN, IERROR)
       IMPORT :: MPI_ADDRESS_KIND
       INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
       INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
   END SUBROUTINE
   SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
                                     WIN, IERROR)
       USE, INTRINSIC ::
                          ISO_C_BINDING, ONLY : C_PTR
       IMPORT :: MPI_ADDRESS_KIND
       INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
       INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
       TYPE(C_PTR) :: BASEPTR
   END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI\_WIN\_ALLOCATE\_CPTR. The implied specific procedure names are described in Section 19.1.5.

*Rationale.* By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (*End of rationale.*)

The info argument can be used to specify hints similar to the info argument for MPI\_WIN\_CREATE and MPI\_ALLOC\_MEM.

The default memory alignment requirements and the "mpi\_minimum\_memory\_alignment" info key described for MPI\_ALLOC\_MEM in Section 9.2 apply to all processes with non-zero size argument.

 $\overline{7}$ 

 $^{24}$ 

	548	$C_{1}$	HAPTER 12. ONE-SIDED COMMUNICATIONS
1 2 3	12.2.3 W	indow That Allocates Shared	d Memory
4	MPI_WIN_	ALLOCATE_SHARED(size, di	sp_unit, info, comm, baseptr, win)
5 6	IN	size	size of local window in bytes (non-negative integer)
7 8	IN	disp_unit	local unit size for displacements, in bytes (positive integer)
9	IN	info	info argument (handle)
10 11	IN	comm	intra-communicator (handle)
12	OUT	baseptr	address of local allocated window segment (choice)
13 14	OUT	win	window object returned by the call (handle)
15 16 17 18 19		in_allocate_shared(MPI_A MPI_Comm comm, void	int size, int disp_unit, MPI_Info info, *baseptr, MPI_Win *win) _Aint size, MPI_Aint disp_unit,
20 21		MPI_Info info, MPI_C	Comm comm, void *baseptr, MPI_Win *win)
22 23 24 25 26 27 28 29 30 31	MPI_Win_a USE, INTEG INTEG TYPE( TYPE( TYPE(	008 binding llocate_shared(size, dis INTRINSIC :: ISO_C_BINDI ER(KIND=MPI_ADDRESS_KIND ER, INTENT(IN) :: disp_u MPI_Info), INTENT(IN) :: MPI_Comm), INTENT(IN) :: C_PTR), INTENT(OUT) :: b MPI_Win), INTENT(OUT) :: ER, OPTIONAL, INTENT(OUT)	), INTENT(IN) :: size nit info comm aseptr win
32 33 34 35 36 37 38 39	USE, INTEG TYPE( TYPE( TYPE( TYPE(	INTRINSIC :: ISO_C_BINDI	), INTENT(IN) :: size, disp_unit info comm aseptr win
40 41 42 43 44 45	INTEG INTEG	LLOCATE_SHARED(SIZE, DIS ER(KIND=MPI_ADDRESS_KIND ER DISP_UNIT, INFO, COMM	, WIN, IERROR
45 46 47 48	process, it comm, and	allocates memory of at leas l returns a pointer to the loc	by all processes in the group of <b>comm</b> . On each t <b>size</b> bytes that is shared among all processes in ally allocated segment in <b>baseptr</b> that can be used process. The locally allocated memory can be the

1 target of load/store accesses by remote processes; the base pointers for other processes  $\mathbf{2}$ can be queried using the function MPI\_WIN\_SHARED\_QUERY. The call also returns a 3 window object that can be used by all processes in comm to perform RMA operations. 4 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 5can create a shared memory segment that can be accessed by all processes in the group. 6  $\overline{7}$ The discussions of rationales for MPI\_ALLOC\_MEM and MPI\_FREE\_MEM in Section 9.2 also apply to MPI\_WIN\_ALLOCATE\_SHARED; in particular, see the rationale in Section 9.2 8 9 for 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 10 11process 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 1213 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:

INTERFACE MPI_WIN_ALLOCATE_SHARED
SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
BASEPTR, WIN, IERROR)
IMPORT :: MPI_ADDRESS_KIND
INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
END SUBROUTINE
SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
BASEPTR, WIN, IERROR)
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
IMPORT :: MPI_ADDRESS_KIND
INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
TYPE(C_PTR) :: BASEPTR
END SUBROUTINE
END INTERFACE

The base procedure name of this overloaded function is MPI\_WIN\_ALLOCATE\_SHARED\_CPTR. The implied specific procedure names are described in Section 19.1.5.

The info argument can be used to specify hints similar to the info argument for MPI\_WIN\_CREATE, MPI\_WIN\_ALLOCATE, and MPI\_ALLOC\_MEM. The additional info key "alloc\_shared\_noncontig" allows the library to optimize the layout of the shared memory segments in memory.

Advice to users. If the info key "alloc\_shared\_noncontig" is not set to true, the allocation strategy is to allocate contiguous memory across process ranks. This may limit the performance on some architectures because it does not allow the implementation to modify the data layout (e.g., padding to reduce access latency). (End of advice to users.)

# Unofficial Draft for Comment Only

14 15

16

17

18

34 35

36

37

38

39

40

41

Advice to implementors. If the user sets the info key "alloc\_shared\_noncontig" to true, the implementation can allocate the memory requested by each process in a location that is close to this process. This can be achieved by padding or allocating memory in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. (End of advice to implementors.)

For contiguous shared memory allocations, the default alignment requirements outlined for MPI\_ALLOC\_MEM in Section 9.2 and the "mpi\_minimum\_memory\_alignment" info key apply to the start of the contiguous memory that is returned in baseptr to the first process with non-zero size argument. For noncontiguous memory allocations, the default alignment requirements and the "mpi\_minimum\_memory\_alignment" info key apply to all processes with non-zero size argument.

Advice to users. If the info key "alloc\_shared\_noncontig" is not set to true (or ignored by the MPI implementation), the alignment of the memory returned in baseptr to all but the first process with non-zero size argument depends on the value of the size argument provided by other processes. It is thus the user's responsibility to control the alignment of contiguous memory allocated for these processes by ensuring that each process provides a size argument that is an integral multiple of the alignment required for the application. (End of advice to users.)

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the *unified memory model* (see Section 12.4) by utilizing the window synchronization functions (see Section 12.5) or explicitly completing outstanding store accesses (e.g., by calling MPI\_WIN\_FLUSH). MPI does not define semantics for accessing shared memory windows in the *separate memory model*.

MPI\_WIN\_SHARED\_QUERY(win, rank, size, disp\_unit, baseptr)

31	IN	win	shared memory window object (handle)	
32	IN	rank	rank in the group of window win or	
33			MPI_PROC_NULL (non-negative integer)	
34	OUT	size	size of the window segment (non-negative integer)	
35 36 37	OUT	disp_unit	local unit size for displacements, in bytes (positive integer)	
38 39	OUT	baseptr	address for load/store access to window segment (choice)	
40				
41	C binding			
42	int MPI_W	in_shared_query(MPI_Win w	vin, int rank, MPI_Aint *size,	
43		int *disp_unit, void	*baseptr)	
44	int MPI_W	in_shared_query_c(MPI_Wir	n win, int rank, MPI_Aint *size,	
45 46	-	MPI_Aint *disp_unit,		
40				
48		0	ize, disp_unit, baseptr, ierror)	

1

2

3

4

5

6

7

8

9

10

11

12 13

14

15

16

17

18

19

20

28 29

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 INTEGER, INTENT(OUT) :: disp\_unit TYPE(C\_PTR), INTENT(OUT) :: baseptr INTEGER, OPTIONAL, INTENT(OUT) :: ierror 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, disp\_unit TYPE(C\_PTR), INTENT(OUT) :: baseptr INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI\_WIN\_SHARED\_QUERY(WIN, RANK, SIZE, DISP\_UNIT, BASEPTR, IERROR) INTEGER WIN, RANK, DISP\_UNIT, IERROR

```
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

This function queries the process-local address for remote memory segments created with MPI\_WIN\_ALLOCATE\_SHARED. This function can return different process-local addresses for the same physical memory on different processes. The returned memory can be used for load/store accesses subject to the constraints defined in Section 12.7. This function can only be called with windows of flavor MPI\_WIN\_FLAVOR\_SHARED. If the passed window is not of flavor MPI\_WIN\_FLAVOR\_SHARED, the error MPI\_ERR\_RMA\_FLAVOR is raised. When rank is MPI\_PROC\_NULL, the pointer, disp\_unit, and size returned are the pointer, disp\_unit, and size of the memory segment belonging the lowest rank that specified size > 0. If all processes in the group attached to the window specified size = 0, then the call returns size = 0 and a baseptr as if MPI\_ALLOC\_MEM was called with size = 0.

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:

INTERFACE MPI_WIN_SHARED_QUERY	
SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &	36
BASEPTR, IERROR)	37
	38
IMPORT :: MPI_ADDRESS_KIND	39
INTEGER WIN, RANK, DISP_UNIT, IERROR	40
INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	41
END SUBROUTINE	42
SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT,	
BASEPTR, IERROR)	
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	44
,,	45
IMPORT :: MPI_ADDRESS_KIND	46
INTEGER :: WIN, RANK, DISP_UNIT, IERROR	47
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	48

1

2

5 6

10

11

12

13

14

15 16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

12	TYPE(C_PTR) :: BASEPTR END SUBROUTINE
3	END SUBJUCTIONS
4 5 6 7 8	The base procedure name of this overloaded function is MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in Section 19.1.5.
9	12.2.4 Window of Dynamically Attached Memory
10 11 12 13 14 15 16 17 18 19 20 21 20 21 22 23 24 25 26	The MPI-2 RMA model requires the user to identify the local memory that may be a target of RMA calls at the time the window is created. This has advantages for both the programmer (only this memory can be updated by one-sided operations and provides greater safety) and the MPI implementation (special steps may be taken to make one-sided access to such memory more efficient). However, consider implementing a modifiable linked list using RMA operations; as new items are added to the list, memory must be allocated. In a C or C++ program, this memory is typically allocated using malloc or new respectively. In MPI-2 RMA, the programmer must create a window with a predefined amount of memory and then implement routines for allocating memory from within the window's memory. In addition, there is no easy way to handle the situation where the predefined amount of memory turns out to be inadequate. To support this model, the routine MPI_WIN_CREATE_DYNAMIC creates a window that makes it possible to expose memory without remote synchronization. It must be used in combination with the local routines MPI_WIN_ATTACH and MPI_WIN_DETACH.
27 28	IN info info info argument (handle)
29	IN comm intra-communicator (handle)
30 31	OUT win window object returned by the call (handle)
32 33 34	C binding int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
35 36 37 38 39 40 41	<pre>Fortran 2008 binding MPI_Win_create_dynamic(info, comm, win, ierror)     TYPE(MPI_Info), INTENT(IN) :: info     TYPE(MPI_Comm), INTENT(IN) :: comm     TYPE(MPI_Win), INTENT(OUT) :: win     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
41 42 43 44	Fortran binding MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR) INTEGER INFO, COMM, WIN, IERROR
45 46 47 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)

WIFT_WINV_ATTACH(WIII, base, size)		32
IN win	window object (handle)	33
IN base	initial address of memory to be attached (choice)	34
IN size	size of memory to be attached in bytes (non-negative	35
	integer)	36
	<i>c</i> ,	37
		38
C binding		39
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)		40
Fortran 2008 binding		41
MPI_Win_attach(win, base, size, ierror)		42
TYPE(MPI_Win), INTENT(IN) :: w		43
TYPE(*), DIMENSION(), ASYNCHRONOUS :: base		44
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size		45
		46
		47
Fortran binding		48

1		TTACH(WIN, BASE, SIZE, IE	ERROR)
2		ER WIN, IERROR	
3	• -	> BASE(*)	
4	INTEGE	ER(KIND=MPI_ADDRESS_KIND)	SIZE
5	Attach	es a local memory region beg	ginning at <b>base</b> for remote access within the given
6			nust not contain any part that is already attached
7			verlapping memory concurrently within the same
8			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
12		_	y contiguous,' see Section $19.1.12$ ). Multiple (but
13			e attached to the same window.
14	non-overiap	ping) memory regions may b	e attached to the same window.
15	Ratio	nale Requiring that mem	ory be explicitly attached before it is exposed to
16			can simplify implementations and improve perfor-
17			y available for RMA operations without requiring a
18		ē	s needed for some one-sided programming models.
19		of rationale.)	a needed for bome one breed programming medder.
20	(		
21	Advic	e to users. Attaching mer	nory to a window may require the use of scarce
22	resour	cces; thus, attaching large re	gions of memory is not recommended in portable
23	progra	ams. Attaching memory to	a window may fail if sufficient resources are not
24	availa	ble; this is similar to the beh	avior of MPI_ALLOC_MEM.
25 26	The u	ser is also responsible for en	suring that MPI_WIN_ATTACH at the target has
20			to target that memory with an MPI RMA call.
28			memory that has not been attached to a window
29			DYNAMIC is erroneous. ( <i>End of advice to users.</i> )
30	create		TRAINIC IS errolleous. (End of davice to users.)
31	Advic	e to implementors. A high	-quality implementation will attempt to make as
32			ing as possible. Any limitations should be docu-
33		ed by the implementor. $(End$	
34			<b>3 1</b> )
35	Attach	ing memory is a local operat	ion as defined by MPI, which means that the call
36	is not collec	tive and completes without r	equiring any MPI routine to be called in any other
37	process. Me	mory may be detached with	the routine MPI_WIN_DETACH. After memory has
38	been detach	ed, it may not be the target	of an MPI RMA operation on that window (unless
39	the memory	$v$ is re-attached with MPI_WI	N_ATTACH).
40			
41			
42		DETACH(win, base)	
43	IN	win	window object (handle)
44	IN	base	initial address of memory to be detached (choice)
45			
46	C binding		
47	int MPI_Win_detach(MPI_Win win, const void *base)		
48	_	-	

# Fortran 2008 binding

MPI_Win_detach(win, base, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
<pre>TYPE(*), DIMENSION(), ASYNCHRONOUS :: base</pre>
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

# Fortran binding

MPI\_WIN\_DETACH(WIN, BASE, IERROR) INTEGER WIN, IERROR <type> BASE(\*)

Detaches a previously attached memory region beginning at base. The arguments base and win must match the arguments passed to a previous call to MPI\_WIN\_ATTACH.

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.)

Memory becomes detached when the associated dynamic memory window is freed, see Section 12.2.5.

12.2.5 Window Destruction
MPI_WIN_FREE(win)
INOUT win window object (handle)
C binding int MPI_Win_free(MPI_Win *win)
Fortran 2008 binding
MPI_Win_free(win, ierror)
TYPE(MPI_Win), INTENT(INOUT) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
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

1	memory that was alloca	ated in MPI_WIN_ALLOCATE	E. If the window was created with	
2	MPI_WIN_ALLOCATE_SHARED, MPI_WIN_FREE will free the window memory that was			was
3	allocated in MPI_WIN_ALLOCATE_SHARED.			
4	9		to MPI_WIN_CREATE_DYNAMIC	
5			effect as if all attached memory	was
6	detached by calls to MP	PI_WIN_DETACH.		
7	Advise to impleme	ntona MDI W/INI EDEE room	ires a barrier synchronization: no	pro
8 9	-	-	he group of win call free. This ens	-
9 10		-	window (e.g., with lock/unlock) a	
11	-	_	when the user sets the "no_locks"	
12			case, an MPI implementation may	
13	°.	8	on. (End of advice to implemented	
14		interiout partier synemonizati		0.)
15	12.2.6 Window Attrib	utes		
16	The following attributes	are cached with a window w	when the window is created	
17	The following attributes			
18	MPI_WIN_BASE	window bas	se address.	
19	MPI_WIN_SIZE	window size	e, in bytes.	
20	MPI_WIN_DISP_UNIT	-	nt unit associated with the window	Ν.
21	MPI_WIN_CREATE_FL		ndow was created.	
22	MPI_WIN_MODEL	memory me	odel for window.	
23 24	In C, calls to MPI_	Win_get_attr(win, MPI_WIN_	BASE, &base, &flag),	
24 25		MPI_WIN_SIZE, &size, &flag	= / 1	
26		MPI_WIN_DISP_UNIT, &disp		
27		MPI_WIN_CREATE_FLAVOR		
28	MPI_Win_get_attr(win,	MPI_WIN_MODEL, & memor	y_model, &flag) will return in ba	se a
29	pointer to the start of t	he window win, and will retu	arn in size, disp_unit, create_kind,	and
30	memory_model pointers	to the size, displacement un	it of the window, the kind of rou	$\operatorname{tine}$
31	used to create the wind	ow, and the memory model,	respectively. A detailed listing of	the
32			to $MPI_WIN_GET_ATTR$ and	
33	MPI_WIN_SET_ATTR is	s shown in Table 12.1.		
34		Attribute	C Type	
35		MPI_WIN_BASE	void *	
36		MPI_WIN_BASE	MPI_Aint *	
37		MPI_WIN_SIZE MPI_WIN_DISP_UNIT		
38		MPI_WIN_CREATE_FLAVOR	int * int *	
39		MPI_WIN_MODEL	int *	
40		MFI_WIN_MODEL		
41				
42	Table 12.1: C types of	attribute value argument to	MPI_WIN_GET_ATTR and	
43	MPI_WIN_SET_ATTR.			
44				
45	· · · · · · · · · · · · · · · · · · ·	· ·	MPI_WIN_BASE, base, flag, ierror)	,
46 47		win, MPI_WIN_SIZE, size, flag		
47 48		win, MPI_WIN_DISP_UNIT, c		
10	IVIPI_VVIN_GET_ATTR()	WIN, WIFI_WIN_CKEATE_FLA	VOR, create_kind, flag, ierror), and	

MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_MODEL, memory\_model, flag, ierror) will return in base, size, disp\_unit, create\_kind, and memory\_model the (integer representation of) the base address, the size, the displacement unit of the window win, the kind of routine used to create the window, and the memory model, respectively.

The values of create\_kind are

MPI_WIN_FLAVOR_CREATE	Window was created with MPI_WIN_CREATE.	7
MPI_WIN_FLAVOR_ALLOCATE	Window was created with MPI_WIN_ALLOCATE.	8
MPI_WIN_FLAVOR_DYNAMIC	Window was created with	9
	MPI_WIN_CREATE_DYNAMIC.	10
MPI_WIN_FLAVOR_SHARED	Window was created with	11
	MPI_WIN_ALLOCATE_SHARED.	12
		13

The values of memory\_model are MPI\_WIN\_SEPARATE and MPI\_WIN\_UNIFIED. The meaning of these is described in Section 12.4.

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 7.7.3.) 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.

		23		
MPI_WIN_GET_GROUP(win, group)				
IN win	window object (handle)	25		
		26		
OUT group	group of processes which share access to the window	27		
	(handle)	28		
		29		
C binding		30		
<pre>int MPI_Win_get_group(MPI_Win win)</pre>	, MPI_Group *group)	31		
Fortran 2008 binding		32		
MPI_Win_get_group(win, group, ierror)				
TYPE(MPI_Win), INTENT(IN) :: win TYPE(MPI_Group), INTENT(OUT) :: group				
-	0	36		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
Fortran binding				
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)				
INTEGER WIN, GROUP, IERROR				
		41		

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.

# 12.2.7 Window Info

46Hints specified via info (see Section 10) allow a user to provide information to direct opti-47mization. Providing hints may enable an implementation to deliver increased performance 48 or use system resources more efficiently. An implementation is free to ignore all hints;

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

1  $\mathbf{2}$ 

3

4

14

15

16

17

18

19

20

21

2223

42

43 44

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.

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

window object (handle)

info argument (handle)

```
<sup>16</sup> MPI_WIN_SET_INFO(win, info)
```

INOUT win
 IN info

```
C binding
```

```
int MPI_Win_set_info(MPI_Win win, MPI_Info info)
```

```
<sup>24</sup> Fortran 2008 binding
```

```
MPI_Win_set_info(win, info, ierror)
TYPE(MPI Win) INTENT(IN) · · windowing
```

```
TYPE(MPI_Win), INTENT(IN) :: win
```

```
TYPE(MPI_Info), INTENT(IN) :: info
```

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

<sup>29</sup><sub>30</sub> Fortran binding

```
MPI_WIN_SET_INFO(WIN, INFO, IERROR)
```

INTEGER WIN, INFO, IERROR

<sup>33</sup> MPI\_WIN\_SET\_INFO updates the hints of the window associated with win using the <sup>34</sup> hints provided in info. This operation has no effect on previously set or defaulted hints <sup>35</sup> that are not specified by info. It also has no effect on previously set or defaulted hints that <sup>36</sup> are specified by info, but are ignored by the MPI implementation in this call to <sup>37</sup> MPI\_WIN\_SET\_INFO. The call is collective on the group of win. The info object may be <sup>38</sup> different on each process, but any info entries that an implementation requires to be the <sup>39</sup> same on all processes must appear with the same value in each process's info object.

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

46 47 48

41

42

43

44

45

8 9

10

11

12

13 14 15

20 21

27

28

#### MPI\_WIN\_GET\_INFO(win, info\_used)

IN	win	window object (handle)
OUT	info_used	new info object (handle)

#### C binding

int MPI\_Win\_get\_info(MPI\_Win win, MPI\_Info \*info\_used)

#### Fortran 2008 binding

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

#### Fortran binding

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.

# 12.3 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 12.5. Transfers can also be completed with calls to flush routines; see Section 12.5.4 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 46 memory locations is undefined; if a location is updated by a put or accumulate operation, 47 then the outcome of loads or other RMA operations is undefined until the updating operation 48

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

2 3

.5

6

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

has completed at the target. There is one exception to this rule; namely, the same location  $\mathbf{2}$ 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 12.7.

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 messagepassing. It simplifies some coding, and is very useful with accumulate operations, to allow atomic updates of local variables. (*End of rationale.*)

MPI\_PROC\_NULL is a valid target rank in all MPI RMA communication calls. The effect is the same as for MPI\_PROC\_NULL in MPI point-to-point communication. After any RMA operation with rank MPI\_PROC\_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch. 

# 12.3.1 Put

The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target process. The obvious difference is that all arguments are provided by one call—the call executed by the origin process. 

# MPI\_PUT(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, win)

30			
31	IN	origin_addr	initial address of origin buffer (choice)
32	IN	origin_count	number of entries in origin buffer (non-negative
33			integer)
34	IN	origin_datatype	datatype of each entry in origin buffer (handle)
35 36	IN	target_rank	rank of target (non-negative integer)
37	IN	target_disp	displacement from start of window to target buffer
38			(non-negative integer)
39	IN	target_count	number of entries in target buffer (non-negative
40			integer)
41 42	IN	target_datatype	datatype of each entry in target buffer (handle)
43	IN	win	window object used for communication (handle)
44			
45	C bind	ing	

```
int MPI_Put(const void *origin_addr, int origin_count,
46
```

```
47
                   MPI_Datatype origin_datatype, int target_rank,
48
```

```
1
              MPI_Aint target_disp, int target_count,
                                                                                     \mathbf{2}
              MPI_Datatype target_datatype, MPI_Win win)
                                                                                     3
int MPI_Put_c(const void *origin_addr, MPI_Count origin_count,
                                                                                     4
              MPI_Datatype origin_datatype, int target_rank,
                                                                                     5
              MPI_Aint target_disp, MPI_Count target_count,
                                                                                     6
              MPI_Datatype target_datatype, MPI_Win win)
                                                                                     7
Fortran 2008 binding
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                     9
                                                                                     10
              target_disp, target_count, target_datatype, win, ierror)
                                                                                     11
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                     12
                                                                                     13
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                    14
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                     15
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                     16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     17
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                     18
              target_disp, target_count, target_datatype, win, ierror)
                                                                                     19
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                     20
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
                                                                                    21
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                    22
    INTEGER, INTENT(IN) :: target_rank
                                                                                    23
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                     ^{24}
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                     25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     26
                                                                                    27
Fortran binding
                                                                                    28
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
                                                                                    29
              TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
                                                                                    30
    <type> ORIGIN_ADDR(*)
                                                                                     31
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
                                                                                     32
               TARGET_DATATYPE, WIN, IERROR
                                                                                     33
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
                                                                                    34
    Transfers origin_count successive entries of the type specified by the origin_datatype,
                                                                                    35
starting at address origin_addr on the origin node, to the target node specified by the win,
                                                                                    36
target_rank pair. The data are written in the target buffer at address target_addr =
                                                                                    37
```

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 <sup>41</sup> a send operation with arguments origin\_addr, origin\_count, origin\_datatype, target\_rank, tag, <sup>42</sup> comm, and the target process executed a receive operation with arguments target\_addr, <sup>43</sup> target\_count, target\_datatype, source, tag, comm, where target\_addr is the target buffer <sup>44</sup> address computed as explained above, the values of tag are arbitrary valid matching tag <sup>45</sup> values, and comm is a communicator for the group of win. <sup>46</sup>

The communication must satisfy the same constraints as for a similar message-passing 47 communication. The target\_datatype may not specify overlapping entries in the target 48

### Unofficial Draft for Comment Only

38

39

1 buffer. The message sent must fit, without truncation, in the target buffer. Furthermore,  $\mathbf{2}$ the target buffer must fit in the target window or in attached memory in a dynamic window. 3 The target\_datatype argument is a handle to a datatype object defined at the origin 4process. However, this object is interpreted at the target process: the outcome is as if 5the target datatype object was defined at the target process by the same sequence of calls 6 used to define it at the origin process. The target datatype must contain only relative  $\overline{7}$ displacements, not absolute addresses. The same holds for get and accumulate operations.

8 9

10

11

12

13

18

21

22

23

 $^{24}$ 

25

26

27

28

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).

14The performance of a put transfer can be significantly affected, on some systems, by 15the choice of window location and the shape and location of the origin and target 16buffer: transfers to a target window in memory allocated by MPI\_ALLOC\_MEM or 17 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 19 communication buffers may also impact performance. (End of advice to users.) 20

A high-quality implementation will attempt to prevent Advice to implementors. 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 error at the origin call if an out-ofbound 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.)

29 30 31

12.3.2	Get		1 2
			3
MPI_GE	ET(origin_addr, origin_count, target_datatype, wii	, origin_datatype, target_rank, target_disp, target_count, n)	4 5
OUT	origin_addr	initial address of origin buffer (choice)	6 7
IN	origin_count	number of entries in origin buffer (non-negative integer)	8 9
IN	origin_datatype	datatype of each entry in origin buffer (handle)	10
IN	target_rank	rank of target (non-negative integer)	11 12
IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)	13 14
IN	target_count	number of entries in target buffer (non-negative integer)	15 16
IN	target_datatype	datatype of each entry in target buffer (handle)	17 18
IN	win	window object used for communication (handle)	19
			20
C bind			21
int MP	I_Get(void *origin_addr	-	22 23
		gin_datatype, int target_rank, disp, int target_count,	24
	<b>•</b>	get_datatype, MPI_Win win)	25
			26
int MP	Ŭ	dr, MPI_Count origin_count,	27
		gin_datatype, int target_rank,	28
		disp, MPI_Count target_count,	29
	MPI_Datatype tar	get_datatype, MPI_Win win)	30
Fortra	n 2008 binding		31
MPI_Get		ount, origin_datatype, target_rank,	32
		get_count, target_datatype, win, ierror)	33 34
		SYNCHRONOUS :: origin_addr	35
		igin_count, target_rank, target_count	36
		T(IN) :: origin_datatype, target_datatype KIND), INTENT(IN) :: target_disp	37
	PE(MPI_Win), INTENT(IN)	<b>a</b> .	38
	TEGER, OPTIONAL, INTENT		39
			40
MPI_Get	<b>e e</b>	ount, origin_datatype, target_rank,	41
	<b>o i</b>	get_count, target_datatype, win, ierror)	42
		SYNCHRONOUS :: origin_addr	43
		ND), INTENT(IN) :: origin_count, target_count	44
	PE(MPI_Datatype), INTEN TEGER, INTENT(IN) :: ta:	T(IN) :: origin_datatype, target_datatype	45
		rget_rank KIND), INTENT(IN) :: target_disp	46
	PE(MPI_Win), INTENT(IN)		47 48
			10

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     Fortran binding
3
     MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
4
                    TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
5
          <type> ORIGIN_ADDR(*)
6
          INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
7
                     TARGET_DATATYPE, WIN, IERROR
8
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
9
10
         Similar to MPI_PUT, except that the direction of data transfer is reversed. Data
11
     are copied from the target memory to the origin. The origin_datatype may not specify
12
     overlapping entries in the origin buffer. The target buffer must be contained within the
13
     target window or within attached memory in a dynamic window, and the copied data must
14
     fit, without truncation, in the origin buffer.
15
16
     12.3.3 Examples for Communication Calls
17
     These examples show the use of the MPI_GET function. As all MPI RMA communication
18
     functions are nonblocking, they must be completed. In the following, this is accomplished
19
     with the routine MPI_WIN_FENCE, introduced in Section 12.5.
20
21
     Example 12.1 We show how to implement the generic indirect assignment A = B(map),
22
     where A, B, and map have the same distribution, and map is a permutation. To simplify, we
23
     assume a block distribution with equal size blocks.
24
25
     SUBROUTINE MAPVALS(A, B, map, m, comm, p)
26
     USE MPI
27
     INTEGER m, map(m), comm, p
28
     REAL A(m), B(m)
29
30
     INTEGER otype(p), oindex(m),
                                       & ! used to construct origin datatypes
^{31}
           ttype(p), tindex(m),
                                       & ! used to construct target datatypes
32
           count(p), total(p),
                                       &
33
           disp_int, win, ierr
34
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
35
36
     ! This part does the work that depends on the locations of B.
37
     ! Can be reused while this does not change
38
39
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
40
     disp_int = realextent
41
     size = m * realextent
42
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                                   &
43
                           comm, win, ierr)
44
45
     ! This part does the work that depends on the value of map and
46
     ! the locations of the arrays.
47
     ! Can be reused while these do not change
48
```

Unofficial Draft for Comment Only

```
\mathbf{2}
! Compute number of entries to be received from each process
                                                                                       3
DO i=1,p
                                                                                       4
   count(i) = 0
                                                                                       5
                                                                                       6
END DO
DO i=1,m
                                                                                       7
                                                                                       8
   j = map(i)/m+1
   count(j) = count(j)+1
                                                                                       9
                                                                                       10
END DO
                                                                                       11
total(1) = 0
                                                                                       12
                                                                                       13
DO i=2,p
   total(i) = total(i-1) + count(i-1)
                                                                                       14
                                                                                       15
END DO
                                                                                       16
                                                                                       17
DO i=1,p
                                                                                       18
   count(i) = 0
                                                                                       19
END DO
                                                                                       20
                                                                                       21
! compute origin and target indices of entries.
! entry i at current process is received from location
                                                                                       22
                                                                                       23
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
                                                                                       ^{24}
! j = 1...p and k = 1...m
                                                                                       25
                                                                                       26
DO i=1,m
   j = map(i)/m+1
                                                                                       27
   k = MOD(map(i), m) + 1
                                                                                       28
                                                                                       29
   count(j) = count(j)+1
                                                                                       30
   oindex(total(j) + count(j)) = i
   tindex(total(j) + count(j)) = k
                                                                                       31
                                                                                       32
END DO
                                                                                       33
                                                                                       34
! create origin and target datatypes for each get operation
DO i=1,p
                                                                                       35
   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                                                                       36
                                                                                       37
                                       oindex(total(i)+1:total(i)+count(i)), &
                                                                                       38
                                       MPI_REAL, otype(i), ierr)
   CALL MPI_TYPE_COMMIT(otype(i), ierr)
                                                                                       39
   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                                                                       40
                                                                                       41
                                       tindex(total(i)+1:total(i)+count(i)), &
                                                                                       42
                                       MPI_REAL, ttype(i), ierr)
   CALL MPI_TYPE_COMMIT(ttype(i), ierr)
                                                                                       43
                                                                                       44
END DO
                                                                                       45
                                                                                       46
! this part does the assignment itself
                                                                                       47
CALL MPI_WIN_FENCE(0, win, ierr)
                                                                                       48
disp_aint = 0
```

```
1
     DO i=1,p
\mathbf{2}
        CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
3
     END DO
4
     CALL MPI_WIN_FENCE(0, win, ierr)
5
6
     CALL MPI_WIN_FREE(win, ierr)
\overline{7}
     DO i=1,p
8
        CALL MPI_TYPE_FREE(otype(i), ierr)
9
        CALL MPI_TYPE_FREE(ttype(i), ierr)
10
     END DO
11
     RETURN
12
     END
13
14
     Example 12.2 A simpler version can be written that does not require that a datatype
15
     be built for the target buffer. But, one then needs a separate get call for each entry, as
16
     illustrated below. This code is much simpler, but usually much less efficient, for large arrays.
17
18
     SUBROUTINE MAPVALS(A, B, map, m, comm, p)
19
     USE MPI
20
     INTEGER m, map(m), comm, p
21
     REAL A(m), B(m)
22
     INTEGER disp_int, win, ierr
23
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
24
25
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
26
     disp_int = realextent
27
     size = m * realextent
28
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                                   &
29
                            comm, win, ierr)
30
^{31}
     CALL MPI_WIN_FENCE(0, win, ierr)
32
     DO i=1,m
33
        j = map(i)/m
34
       disp_aint = MOD(map(i),m)
35
        CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL, win, ierr)
36
     END DO
37
     CALL MPI_WIN_FENCE(0, win, ierr)
38
     CALL MPI_WIN_FREE(win, ierr)
39
     RETURN
40
     END
41
42
     12.3.4 Accumulate Functions
43
44
     It is often useful in a put operation to combine the data moved to the target process with the
45
```

<sup>45</sup> It is often useful in a put operation to combine the data moved to the target process with the <sup>46</sup> data that resides at that process, rather than replacing it. This will allow, for example, the <sup>46</sup> accumulation of a sum by having all involved processes add their contributions to the sum <sup>47</sup> variable in the memory of one process. The accumulate functions have slightly different <sup>48</sup>

# Unofficial Draft for Comment Only

semantics with respect to overlapping data accesses than the put and get functions; see Section 12.7 for details.

### Accumulate Function

# MPI\_ACCUMULATE(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, op, win)

	10.800_000.00, 10.800_0000		9
IN	origin_addr	initial address of buffer (choice)	10
IN	origin_count	number of entries in buffer (non-negative integer)	11
IN	origin_datatype	datatype of each entry (handle)	12
IN	target_rank	rank of target (non-negative integer)	13 14
IN	target_disp	displacement from start of window to beginning of	15
		target buffer (non-negative integer)	16
IN	target_count	number of entries in target buffer (non-negative	17
	-	integer)	18
IN	target_datatype	datatype of each entry in target buffer (handle)	19 20
IN	ор	reduce operation (handle)	21
IN	win	window object (handle)	22
	WIII	window object (nandre)	23
1 <b>L</b>			24
binding ?			25

# C binding

e binding	25
<pre>int MPI_Accumulate(const void *origin_addr, int origin_count,</pre>	26
MPI_Datatype origin_datatype, int target_rank,	27
MPI_Aint target_disp, int target_count,	28
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	29
<pre>int MPI_Accumulate_c(const void *origin_addr, MPI_Count origin_count,</pre>	30
MPI_Datatype origin_datatype, int target_rank,	31
MPI_Aint target_disp, MPI_Count target_count,	32
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	33
	34
Fortran 2008 binding	35
MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,	36
<pre>target_disp, target_count, target_datatype, op, win, ierror)</pre>	37
<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>	38
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count	39
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype	40
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	41
TYPE(MPI_Op), INTENT(IN) :: op	42
TYPE(MPI_Win), INTENT(IN) :: win	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
<pre>MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,</pre>	45
<pre>target_disp, target_count, target_datatype, op, win, ierror)</pre>	46
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr	47
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count	48

## **Unofficial Draft for Comment Only**

 $\mathbf{2}$ 

1	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
2	INTEGER, INTENT(IN) :: target_rank
3	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
4	TYPE(MPI_Op), INTENT(IN) :: op
5	TYPE(MPI_Win), INTENT(IN) :: win
6	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7	Fortran binding
8	MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
9	TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
10	<pre><type> ORIGIN_ADDR(*)</type></pre>
11	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
12	
13	TARGET_DATATYPE, OP, WIN, IERROR
14	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
15	Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count, and
16	origin_datatype) to the buffer specified by arguments target_count and target_datatype, at
17	offset target_disp, in the target window specified by target_rank and win, using the operation
18	op. This is like MPI_PUT except that data is combined into the target area instead of
19	overwriting it.
20	Any of the predefined operations for MPI_REDUCE can be used. User-defined functions
21	cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added
22	to the corresponding element in the target, replacing the former value in the target.
23	Each datatype argument must be a predefined datatype or a derived datatype, where
24	all basic components are of the same predefined datatype. Both datatype arguments must
25	be constructed from the same predefined datatype. The operation <b>op</b> applies to elements of
26	that predefined type. The parameter target_datatype must not specify overlapping entries,
27	and the target buffer must fit in the target window.
28	A new predefined operation, $MPI_REPLACE$ , is defined. It corresponds to the associative
29	function $f(a, b) = b$ ; i.e., the current value in the target memory is replaced by the value
30	supplied by the origin.
31	MPI_REPLACE can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE,
32	MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but not
33	in collective reduction operations such as MPI_REDUCE.
34	
35	Advice to users. MPI_PUT is a special case of MPI_ACCUMULATE, with the op-
36	eration MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have
37	different constraints on concurrent updates. (End of advice to users.)
38	
39 40	<b>Example 12.3</b> We want to compute $B(j) = \sum_{map(i)=j} A(i)$ . The arrays A, B, and map
40	are distributed in the same manner. We write the simple version.
41	are distributed in the same manner. We write the simple version.
42	SUBROUTINE SUM(A, B, map, m, comm, p)
43 44	USE MPI
44 45	INTEGER m, map(m), comm, p, win, ierr, disp_int
46	REAL A(m), B(m)
47	INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
48	•

```
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 12.2, except that a call to get has been 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 example.) In a similar manner, we can replace in Example 12.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

### Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 12.7 for details). The predefined operation MPI\_REPLACE provides fetch-and-set behavior.

 $\mathbf{2}$ 

 $^{24}$ 

 $^{31}$ 

```
1
     MPI_GET_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)
4
       IN
                 origin_addr
                                              initial address of buffer (choice)
5
                 origin_count
       IN
                                             number of entries in origin buffer (non-negative
6
                                             integer)
7
8
       IN
                 origin_datatype
                                             datatype of each entry in origin buffer (handle)
9
       OUT
                 result_addr
                                             initial address of result buffer (choice)
10
                 result_count
                                              number of entries in result buffer (non-negative
       IN
11
                                             integer)
12
       IN
                 result_datatype
                                              datatype of each entry in result buffer (handle)
13
14
       IN
                 target_rank
                                             rank of target (non-negative integer)
15
       IN
                 target_disp
                                              displacement from start of window to beginning of
16
                                              target buffer (non-negative integer)
17
       IN
                 target_count
                                              number of entries in target buffer (non-negative
18
                                              integer)
19
20
       IN
                 target_datatype
                                              datatype of each entry in target buffer (handle)
21
       IN
                                              reduce operation (handle)
                 op
22
       IN
                 win
                                              window object (handle)
23
24
25
     C binding
26
     int MPI_Get_accumulate(const void *origin_addr, int origin_count,
27
                    MPI_Datatype origin_datatype, void *result_addr,
28
                     int result_count, MPI_Datatype result_datatype,
                     int target_rank, MPI_Aint target_disp, int target_count,
29
30
                    MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
^{31}
     int MPI_Get_accumulate_c(const void *origin_addr, MPI_Count origin_count,
32
                    MPI_Datatype origin_datatype, void *result_addr,
33
                    MPI_Count result_count, MPI_Datatype result_datatype,
34
                     int target_rank, MPI_Aint target_disp, MPI_Count target_count,
35
                    MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
36
37
     Fortran 2008 binding
38
     MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
39
                    result_count, result_datatype, target_rank, target_disp,
40
                    target_count, target_datatype, op, win, ierror)
41
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
42
          INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
43
                     target_count
44
          TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
45
                     target_datatype
46
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
47
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
48
          TYPE(MPI_Op), INTENT(IN) :: op
```

```
1
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   \mathbf{2}
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   3
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
                                                                                   4
              result_count, result_datatype, target_rank, target_disp,
                                                                                   5
              target_count, target_datatype, op, win, ierror)
                                                                                   6
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                   7
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
                                                                                   8
              target_count
                                                                                   9
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                   10
              target_datatype
                                                                                   11
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                   12
    INTEGER, INTENT(IN) :: target_rank
                                                                                   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
Fortran binding
                                                                                   19
MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
              RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
                                                                                   20
                                                                                   21
              TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
                                                                                   22
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
                                                                                   23
                                                                                   ^{24}
              TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
                                                                                   25
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
                                                                                   26
```

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 12.7 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, 41 42 42 42 43 44 44 45 4647

27

28

29

30

31

32

33

34

35

36

37

38

39

40

	572	СН	IAPTER 12. ONE-SIDED COMMUNICATIONS	
1 2 3	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.			
4 5 6 7	tion M	MPI_NO_OP. Note, however, the	ilar to MPI_GET_ACCUMULATE, with the opera- hat MPI_GET and MPI_GET_ACCUMULATE have updates. ( <i>End of advice to users.</i> )	
8	Fetch and (	Dp Function		
9 10 11 12 13 14	and-increm ations. MP	The generic functionality of MPI_GET_ACCUMULATE might limit the performance of fetch- and-increment or fetch-and-add calls that might be supported by special hardware oper- ations. MPI_FETCH_AND_OP thus allows for a fast implementation of a commonly used subset of the functionality of MPI_GET_ACCUMULATE.		
15 16	MPI_FETC	H_AND_OP(origin_addr, result	t_addr, datatype, target_rank, target_disp, op, win)	
17 18	IN	origin_addr	initial address of buffer (choice)	
19	OUT	result_addr	initial address of result buffer (choice)	
20 21	IN	datatype	datatype of the entry in origin, result, and target buffers (handle)	
22 23	IN	target_rank	rank of target (non-negative integer)	
24 25	IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)	
26 27	IN	ор	reduce operation (handle)	
27 28	IN	win	window object (handle)	
29 30 31 32 33 34	C binding int MPI_Fetch_and_op(const void *origin_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Op op, MPI_Win win)			
34 35 36 37 38 39 40 41 42 43 44	<pre>Fortran 2008 binding MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,</pre>			
45	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
46 47 48	Fortran b	•	LT_ADDR, DATATYPE, TARGET_RANK, , IERROR)	

INT	De> ORIGIN_ADDR(*), 1 EGER DATATYPE, TARGE EGER(KIND=MPI_ADDRES:	I_RANK, OP, WIN, IERROR	1 2 3
Accumulate one element of type datatype from the origin buffer (origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation. The origin and result buffers (origin_addr and result_addr) must be disjoint. Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE, can be specified as op; user-defined functions cannot be used. The datatype argument must be a predefined datatype. The operation is executed atomically.			
Compare	and Swap Function		$13 \\ 14$
compared the value	d to the value at the tar is at origin and target an		15 16 17 18 19
MPI_CO	viPARE_AND_SWAP(ori target_rank, targe	gin_addr, compare_addr, result_addr, datatype, et_disp, win)	20 21
IN	origin_addr	initial address of buffer (choice)	22
IN	compare_addr	initial address of compare buffer (choice)	23 24
OUT	result_addr	initial address of result buffer (choice)	25
IN	datatype	datatype of the element in all buffers (handle)	26
IN	target_rank	rank of target (non-negative integer)	27
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)	28 29 30
IN	win	window object (handle)	31
			32 33
C bindi	-		34
int MPI	<pre>int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,</pre>		
		ddr, MPI_Datatype datatype, int target_rank, t_disp, MPI_Win win)	36
<b>D</b> (			37
	2008 binding	_addr, compare_addr, result_addr, datatype,	38 39
MP1_COM		arget_disp, win, ierror)	40
TYPI	-	INTENT(IN), ASYNCHRONOUS :: origin_addr,	41
	compare_addr	C C	42
		ASYNCHRONOUS :: result_addr	43
	E(MPI_Datatype), INT		44 45
	EGER, INTENT(IN) :: 1 EGER(KIND=MPI ADDRES)	target_rank S_KIND), INTENT(IN) :: target_disp	45 46
	EGEN(KIND=MI_RDDNES. E(MPI_Win), INTENT(II	<b>o</b> .	47
	EGER, OPTIONAL, INTE		48

1	Fortran binding
2	MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
3	TARGET_RANK, TARGET_DISP, WIN, IERROR)
4	<type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)</type>

INTEGER DATATYPE, TARGET\_RANK, WIN, IERROR

INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP

This function compares one element of type datatype in the compare buffer 8 compare\_addr with the buffer at offset target\_disp in the target window specified by 9 target\_rank and win and replaces the value at the target with the value in the origin buffer 10 origin\_addr if the compare buffer and the target buffer are identical. The original value at 11 the target is returned in the buffer result\_addr. The parameter datatype must belong to 12one of the following categories of predefined datatypes: C integer, Fortran integer, Logical, 13 Multi-language types, or Byte as specified in Section 6.9.2. The origin and result buffers 14(origin\_addr and result\_addr) must be disjoint. 15

16

5

6

7

## 17 12.3.5 Request-based RMA Communication Operations

<sup>18</sup> Request-based RMA communication operations allow the user to associate a request handle <sup>19</sup> with the RMA operations and test or wait for the completion of these requests using the <sup>20</sup> functions described in Section 3.7.3. Request-based RMA operations are only valid within <sup>21</sup> a passive target epoch (see Section 12.5).

<sup>22</sup> Upon returning from a completion call in which an RMA operation completes, all fields <sup>23</sup> of the status object, if any, and the results of status query functions (e.g.,

<sup>24</sup> MPI\_GET\_COUNT) are undefined with the exception of MPI\_ERROR if appropriate (see <sup>25</sup> Section 3.2.5). It is valid to mix different request types (e.g., any combination of RMA <sup>26</sup> requests, collective requests, I/O requests, generalized requests, or point-to-point requests) <sup>27</sup> in functions that enable multiple completions (e.g., MPI\_WAITALL). It is erroneous to call <sup>28</sup> MPI\_REQUEST\_FREE or MPI\_CANCEL for a request associated with an RMA operation. <sup>29</sup> RMA requests are not persistent.

<sup>0</sup> The end of the epoch, or explicit bulk synchronization using

<sup>31</sup> MPI\_WIN\_FLUSH, MPI\_WIN\_FLUSH\_ALL, MPI\_WIN\_FLUSH\_LOCAL, or

MPI\_WIN\_FLUSH\_LOCAL\_ALL, also indicates completion of the RMA operations. How ever, users must still wait or test on the request handle to allow the MPI implementation to
 clean up any resources associated with these requests; in such cases the wait operation will
 complete locally.

MPI_RPU	T(origin_addr, origin_count, target_count, target_d	origin_datatype, target_rank, target_disp, latatype, win, request)	$\frac{1}{2}$
IN	origin_addr	initial address of origin buffer (choice)	3 4
IN	origin_count	number of entries in origin buffer (non-negative integer)	4 5 6
IN	origin_datatype	datatype of each entry in origin buffer (handle)	7
IN	target_rank	rank of target (non-negative integer)	8
IN	target_disp	displacement from start of window to target buffer (non-negative integer)	9 10 11
IN	target_count	number of entries in target buffer (non-negative integer)	12 13
IN	target_datatype	datatype of each entry in target buffer (handle)	14
IN	win	window object used for communication (handle)	15 16
OUT	request	RMA request (handle)	17 18
	_		19
C bindin	0	_addr, int origin_count,	20
1110 111 1_		n_datatype, int target_rank,	21
		sp, int target_count,	22
	MPI_Datatype targe	t_datatype, MPI_Win win,	23
	MPI_Request *reque	est)	24 25
int MPI	Rput c(const void *orig	in_addr, MPI_Count origin_count,	25 26
_	-	.n_datatype, int target_rank,	27
		sp, MPI_Count target_count,	28
	MPI_Datatype targe	t_datatype, MPI_Win win,	29
	MPI_Request *reque	est)	30
Fortran	2008 binding		31
		unt, origin_datatype, target_rank,	32
_ 1		et_count, target_datatype, win, request,	33
	ierror)		34
		ENT(IN), ASYNCHRONOUS :: origin_addr	35 36
	Ū.	<pre>in_count, target_rank, target_count</pre>	30
		<pre>IN) :: origin_datatype, target_datatype</pre>	38
		ND), INTENT(IN) :: target_disp	39
	<pre>(MPI_Win), INTENT(IN) : (MPI_Request), INTENT(0)</pre>		40
	GER, OPTIONAL, INTENT(O	-	41
	dent, of flower, intent (o		42
MPI_Rput		unt, origin_datatype, target_rank,	43
		t_count, target_datatype, win, request,	44
ייינאיי	ierror)		45
		<pre>ENT(IN), ASYNCHRONOUS :: origin_addr ), INTENT(IN) :: origin_count, target_count</pre>	46 47
		IN) :: origin_datatype, target_datatype	47 48

1 2 3 4 5 6	INTE TYPE TYPE	GER, INTENT(IN) :: targe GER(KIND=MPI_ADDRESS_KIN (MPI_Win), INTENT(IN) :: (MPI_Request), INTENT(OU GER, OPTIONAL, INTENT(OU	D), INTENT(IN) :: target_disp win T) :: request
7 8 9 10 11 12 13 14	<typ INTE</typ 	CORIGIN_ADDR, ORIGIN_COU TARGET_DISP, TARGET IERROR) e> ORIGIN_ADDR(*) GER ORIGIN_COUNT, ORIGIN	NT, ORIGIN_DATATYPE, TARGET_RANK, T_COUNT, TARGET_DATATYPE, WIN, REQUEST, T_DATATYPE, TARGET_RANK, TARGET_COUNT, TIN, REQUEST, IERROR D) TARGET_DISP
15 16 17 18 19 20 21 22 23	nication of The comp dicates the not indicates quired, M	request object and associates oletion of an MPI_RPUT oper- nat the sender is now free to ate that the data is available	Γ (Section 12.3.1), except that it allocates a commu- it with the request handle (the argument request). ration (i.e., after the corresponding test or wait) in- o update the locations in the origin buffer. It does at the target window. If remote completion is re- FLUSH_ALL, MPI_WIN_UNLOCK, or
24 25	MPI_RGE	T(origin_addr, origin_count, c target_count, target_da	origin_datatype, target_rank, target_disp, atatype, win, request)
26	OUT	origin_addr	initial address of origin buffer (choice)
27 28 29	IN	origin_count	number of entries in origin buffer (non-negative integer)
30	IN	origin_datatype	datatype of each entry in origin buffer (handle)
31	IN	target_rank	rank of target (non-negative integer)
32 33 34	IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)
35 36	1N	target_count	number of entries in target buffer (non-negative integer)
37	IN	target_datatype	datatype of each entry in target buffer (handle)
38 39	IN	win	window object used for communication (handle)
40	OUT	request	RMA request (handle)
41 42 43 44 45 46 47 48	C bindin int MPI_	Rget(void *origin_addr, MPI_Datatype origin MPI_Aint target_dis	n_datatype, int target_rank, sp, int target_count, c_datatype, MPI_Win win,

```
1
int MPI_Rget_c(void *origin_addr, MPI_Count origin_count,
                                                                                    \mathbf{2}
              MPI_Datatype origin_datatype, int target_rank,
                                                                                    3
              MPI_Aint target_disp, MPI_Count target_count,
              MPI_Datatype target_datatype, MPI_Win win,
                                                                                    4
              MPI_Request *request)
                                                                                    5
                                                                                    6
Fortran 2008 binding
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
              target_disp, target_count, target_datatype, win, request,
                                                                                    9
              ierror)
                                                                                    10
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                    11
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                    12
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                    13
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                    14
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                    15
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    17
                                                                                    18
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                    19
              target_disp, target_count, target_datatype, win, request,
                                                                                    20
              ierror)
                                                                                    21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                    22
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
                                                                                    23
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                    24
    INTEGER, INTENT(IN) :: target_rank
                                                                                    25
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                    26
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                    27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    29
Fortran binding
                                                                                    30
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
                                                                                    31
              TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
                                                                                    32
              IERROR)
                                                                                    33
    <type> ORIGIN_ADDR(*)
                                                                                    34
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
                                                                                    35
               TARGET_DATATYPE, WIN, REQUEST, IERROR
                                                                                    36
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
                                                                                    37
                                                                                    38
    MPI_RGET is similar to MPI_GET (Section 12.3.2), except that it allocates a commu-
                                                                                    39
nication request object and associates it with the request handle (the argument request)
                                                                                    40
```

nication 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 operation indicates that the data is available in the origin buffer. If origin\_addr points to memory attached to a window, then the data becomes available in the private copy of this window.

44 45

41

42

43

```
CHAPTER 12. ONE-SIDED COMMUNICATIONS
1
     MPI_RACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
\mathbf{2}
                    target_count, target_datatype, op, win, request)
3
       IN
                origin_addr
                                            initial address of buffer (choice)
4
       IN
                origin_count
                                            number of entries in buffer (non-negative integer)
5
6
                origin_datatype
                                            datatype of each entry in origin buffer (handle)
       IN
7
       IN
                target_rank
                                            rank of target (non-negative integer)
8
       IN
                target_disp
                                            displacement from start of window to beginning of
9
                                            target buffer (non-negative integer)
10
11
       IN
                                            number of entries in target buffer (non-negative
                target_count
12
                                            integer)
13
       IN
                target_datatype
                                            datatype of each entry in target buffer (handle)
14
       IN
                                            reduce operation (handle)
                ор
15
16
       IN
                win
                                            window object (handle)
17
                                            RMA request (handle)
       OUT
                request
18
19
     C binding
20
     int MPI_Raccumulate(const void *origin_addr, int origin_count,
21
                    MPI_Datatype origin_datatype, int target_rank,
22
                    MPI_Aint target_disp, int target_count,
23
                    MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
24
                    MPI_Request *request)
25
26
     int MPI_Raccumulate_c(const void *origin_addr, MPI_Count origin_count,
27
                    MPI_Datatype origin_datatype, int target_rank,
                    MPI_Aint target_disp, MPI_Count target_count,
28
                    MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
29
30
                    MPI_Request *request)
^{31}
     Fortran 2008 binding
32
     MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
33
                    target_disp, target_count, target_datatype, op, win, request,
34
                    ierror)
35
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
36
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
37
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
38
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
39
         TYPE(MPI_Op), INTENT(IN) :: op
40
         TYPE(MPI_Win), INTENT(IN) :: win
41
         TYPE(MPI_Request), INTENT(OUT) :: request
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
45
                    target_disp, target_count, target_datatype, op, win, request,
46
                    ierror)
47
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
48
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
```

TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype INTEGER, INTENT(IN) :: target\_rank INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp TYPE(MPI\_Op), INTENT(IN) :: op TYPE(MPI\_Win), INTENT(IN) :: win TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding

MPI\_RACCUMULATE(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, REQUEST, IERROR) <type> ORIGIN\_ADDR(\*) INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, REQUEST, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP

MPI\_RACCUMULATE is similar to MPI\_ACCUMULATE (Section 12.3.4), 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.

1 MPI\_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 6 integer) 7 8 IN origin\_datatype datatype of each entry in origin buffer (handle) 9 OUT result\_addr initial address of result buffer (choice) 10 result\_count number of entries in result buffer (non-negative IN 11 integer) 1213IN result\_datatype datatype of entries in result buffer (handle) 14IN target\_rank rank of target (non-negative integer) 15IN target\_disp displacement from start of window to beginning of 16 target buffer (non-negative integer) 1718 number of entries in target buffer (non-negative IN target\_count 19 integer) 20IN target\_datatype datatype of each entry in target buffer (handle) 21IN ор reduce operation (handle) 22 23IN window object (handle) win 24OUT RMA request (handle) request 2526C binding 27int MPI\_Rget\_accumulate(const void \*origin\_addr, int origin\_count, 28MPI\_Datatype origin\_datatype, void \*result\_addr, 29 int result\_count, MPI\_Datatype result\_datatype, 30 int target\_rank, MPI\_Aint target\_disp, int target\_count,  $^{31}$ MPI\_Datatype target\_datatype, MPI\_Op op, MPI\_Win win, 32 MPI\_Request \*request) 33 34 int MPI\_Rget\_accumulate\_c(const void \*origin\_addr, MPI\_Count origin\_count, 35 MPI\_Datatype origin\_datatype, void \*result\_addr, 36 MPI\_Count result\_count, MPI\_Datatype result\_datatype, 37 int target\_rank, MPI\_Aint target\_disp, MPI\_Count target\_count, 38 MPI\_Datatype target\_datatype, MPI\_Op op, MPI\_Win win, 39 MPI\_Request \*request) 40 Fortran 2008 binding 41 MPI\_Rget\_accumulate(origin\_addr, origin\_count, origin\_datatype, 42result\_addr, result\_count, result\_datatype, target\_rank, 43 target\_disp, target\_count, target\_datatype, op, win, request, 44 ierror) 45 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr 46 INTEGER, INTENT(IN) :: origin\_count, result\_count, target\_rank, 47target\_count 48

```
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                   1
                                                                                   2
              target_datatype
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   6
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
                                                                                  10
              result_addr, result_count, result_datatype, target_rank,
                                                                                  11
              target_disp, target_count, target_datatype, op, win, request,
                                                                                  12
              ierror)
                                                                                  13
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  14
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
                                                                                  15
              target_count
                                                                                  16
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                  17
              target_datatype
                                                                                  18
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
                                                                                  19
    INTEGER, INTENT(IN) :: target_rank
                                                                                  20
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  21
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  22
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  23
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
                                                                                  26
Fortran binding
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
                                                                                  27
                                                                                  28
              RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
              TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
                                                                                  29
                                                                                  30
              IERROR)
                                                                                  31
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
                                                                                  32
                                                                                  33
              TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
                                                                                  34
              IERROR
                                                                                  35
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
                                                                                  36
```

MPI\_RGET\_ACCUMULATE is similar to MPI\_GET\_ACCUMULATE (Section 12.3.4), 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.

## 12.4 Memory Model

The memory semantics of RMA are best understood by using the concept of *public* and private window copies. We assume that systems have a public memory region that is 47 addressable by all processes (e.g., the shared memory in shared memory machines or the 48

## Unofficial Draft for Comment Only

37

38

39

40

41

42 43

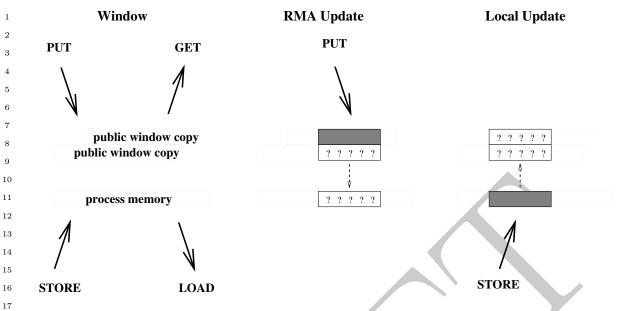


Figure 12.1: Schematic description of the public/private window operations in the MPI\_WIN\_SEPARATE memory model for two overlapping windows.

19 20

18

21exposed main memory in distributed memory machines). In addition, most machines have 22fast private buffers (e.g., transparent caches or explicit communication buffers) local to 23each process where copies of data elements from the main memory can be stored for faster  $^{24}$ access. Such buffers are either coherent, i.e., all updates to main memory are reflected in 25all private copies consistently, or non-coherent, i.e., conflicting accesses to main memory 26need to be synchronized and updated in all private copies explicitly. Coherent systems 27allow direct updates to remote memory without any participation of the remote side. Non-28coherent systems, however, need to call RMA functions in order to reflect updates to the 29public window in their private memory. Thus, in coherent memory, the public and the 30 private window are identical while they remain logically separate in the non-coherent case.  $^{31}$ MPI thus differentiates between two memory models called **RMA unified**, if public and 32 private window are logically identical, and **RMA** separate, otherwise.

33 In the RMA separate model, there is only one instance of each variable in process 34memory, but a distinct *public* copy of the variable for each window that contains it. A load 35 accesses the instance in process memory (this includes MPI sends). A local store accesses 36 and updates the instance in process memory (this includes MPI receives), but the update 37 may affect other public copies of the same locations. A get on a window accesses the public 38 copy of that window. A put or accumulate on a window accesses and updates the public 39 copy of that window, but the update may affect the private copy of the same locations 40in process memory, and public copies of other overlapping windows. This is illustrated in  $^{41}$ Figure **12.1**.

<sup>42</sup> In the RMA unified model, public and private copies are identical and updates via put <sup>43</sup> or accumulate calls are eventually observed by load operations without additional RMA <sup>44</sup> calls. A store access to a window is eventually visible to remote get or accumulate calls <sup>45</sup> without additional RMA calls. These stronger semantics of the RMA unified model allow <sup>46</sup> the user to omit some synchronization calls and potentially improve performance.

- 47
- 48

Advice to users. If accesses in the RMA unified model are not synchronized (with

locks or flushes, see Section 12.5.3), 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.

## 12.5 Synchronization Calls

RMA communications fall in two categories:

- active target communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.
- **passive target communication**, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI\_PUT, MPI\_GET or MPI\_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

#### Unofficial Draft for Comment Only

1 2

3

4

5

6

 $\overline{7}$ 

8

9 10 11

12 13

14

15

16

17

18

19 20

21

22

23

 $^{24}$ 

25

26

27

28 29

30

 $^{31}$ 

32 33

34

35 36

37

38

39

40 41

42

43 44

45

46

47

- 1. The MPI\_WIN\_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates with many others.
- This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI\_WIN\_FENCE. A process can access windows at all processes in the group of win during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.
- 2. The four functions MPI\_WIN\_START, MPI\_WIN\_COMPLETE, MPI\_WIN\_POST, and MPI\_WIN\_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly RMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.
- These calls are used for active target communication. An access epoch is started at the origin process by a call to MPI\_WIN\_START and is terminated by a call to MPI\_WIN\_COMPLETE. The start call has a group argument that specifies the group of target processes for that epoch. An exposure epoch is started at the target process by a call to MPI\_WIN\_POST and is completed by a call to MPI\_WIN\_WAIT. The post call has a group argument that specifies the set of origin processes for that epoch.
- <sup>27</sup> 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 "bulletin board" model, where processes can, at random times, access or update different parts of the bulletin board.
    - 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.

37 Figure 12.2 illustrates the general synchronization pattern for active target communi-38 cation. The synchronization between **post** and **start** ensures that the put call of the origin 39 process does not start until the target process exposes the window (with the **post** call); 40the target process will expose the window only after preceding local accesses to the window 41 have completed. The synchronization between complete and wait ensures that the put call 42of the origin process completes before the window is unexposed (with the wait call). The 43 target process will execute following local accesses to the target window only after the wait 44returned.

Figure 12.2 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 However, such strong synchronization is more than needed for

1

2

3

4

5

6

7

8

9

10

11

12

28

29 30

31 32

33

34

35

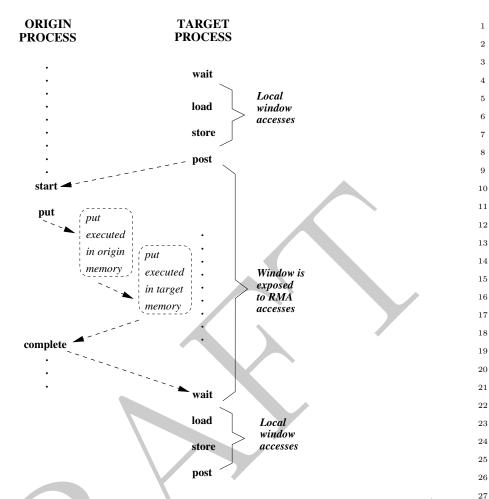


Figure 12.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

correct ordering of window accesses. The semantics of MPI calls allow **weak synchroniza-tion**, as illustrated in Figure 12.3. 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 12.4 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

28

29 30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

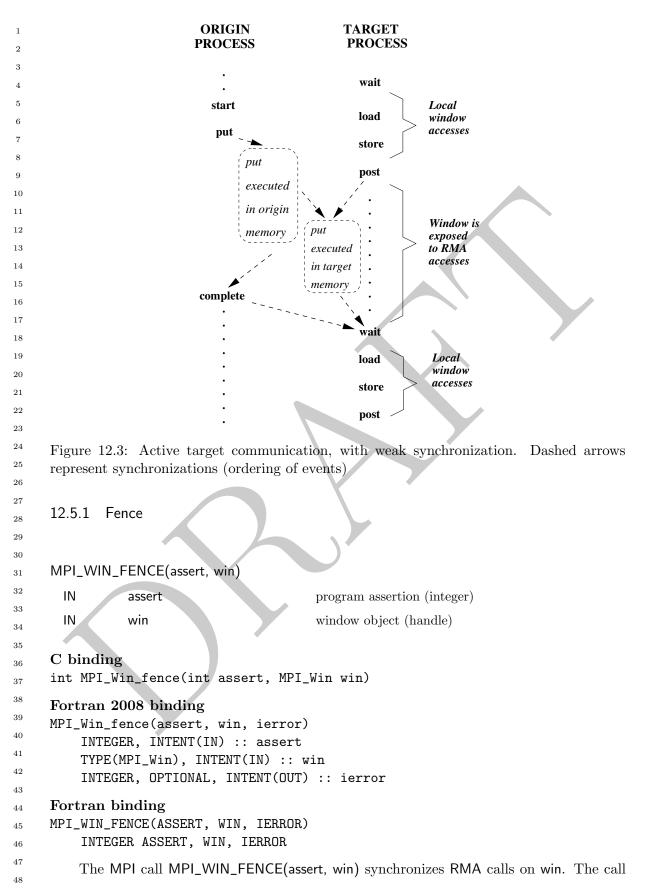
42

43 44

45

46

47



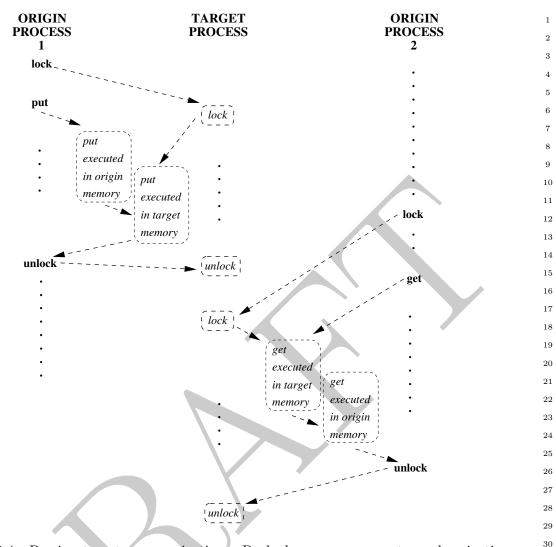


Figure 12.4: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

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 39 the local process issued RMA communication calls on win between these two calls. The call 40completes an RMA exposure epoch if it was preceded by another fence call and the local 41 window was the target of RMA accesses between these two calls. The call starts an RMA 42access epoch if it is followed by another fence call and by RMA communication calls issued 43 between these two fence calls. The call starts an exposure epoch if it is followed by another 44fence call and the local window is the target of RMA accesses between these two fence calls. 45Thus, the fence call is equivalent to calls to a subset of post, start, complete, wait. 46

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.

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

 $^{31}$ 

32 33

34

35

36

37

#### 588

1 However, a call to MPI\_WIN\_FENCE that is known not to end any epoch (in particular, a  $\mathbf{2}$ call with assert equal to MPI\_MODE\_NOPRECEDE) does not necessarily act as a barrier. 3 The assert argument is used to provide assertions on the context of the call that may 4 be used for various optimizations. This is described in Section 12.5.5. A value of assert =50 is always valid. 6 Advice to users. Calls to MPI\_WIN\_FENCE should both precede and follow calls to 7 RMA communication functions that are synchronized with fence calls. (End of advice 8 9 to users.) 10 1112.5.2 General Active Target Synchronization 1213 14MPI\_WIN\_START(group, assert, win) 15group of target processes (handle) IN group 16 17IN program assertion (integer) assert 18 IN window object (handle) win 19 20C binding 21int MPI\_Win\_start(MPI\_Group group, int assert, MPI\_Win win) 22 23Fortran 2008 binding  $^{24}$ MPI\_Win\_start(group, assert, win, ierror) 25TYPE(MPI\_Group), INTENT(IN) :: group 26INTEGER, INTENT(IN) :: assert 27TYPE(MPI\_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2829 Fortran binding 30 MPI\_WIN\_START(GROUP, ASSERT, WIN, IERROR)  $^{31}$ INTEGER GROUP, ASSERT, WIN, IERROR 32 33 Starts an RMA access epoch for win. RMA calls issued on win during this epoch must 34access only windows at processes in group. Each process in group must issue a matching 35 call to MPI\_WIN\_POST. RMA accesses to each target window will be delayed, if necessary, 36 until the target process executed the matching call to MPI\_WIN\_POST. MPI\_WIN\_START 37 is allowed to block until the corresponding MPI\_WIN\_POST calls are executed, but is not 38 required to. 39 The assert argument is used to provide assertions on the context of the call that may 40 be used for various optimizations. This is described in Section 12.5.5. A value of assert =41 0 is always valid. 4243 MPI\_WIN\_COMPLETE(win) 44 45IN window object (handle) win 46 47 C binding 48

int MPI\_Win\_complete(MPI\_Win win)

#### Fortran 2008 binding

```
MPI_Win_complete(win, ierror)
   TYPE(MPI_Win), INTENT(IN) :: win
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### Fortran binding

MPI\_WIN\_COMPLETE(WIN, IERROR) INTEGER WIN, IERROR

Completes an RMA access epoch on win started by a call to MPI WIN START. All RMA communication calls issued on win during this epoch will have completed at the origin when the call returns.

MPI\_WIN\_COMPLETE enforces completion of preceding RMA calls at the origin, but not at the target. A put or accumulate call may not have completed at the target when it has completed at the origin.

Consider the sequence of calls in the example below.

### **Example 12.4** Use of MPI\_WIN\_START and MPI\_WIN\_COMPLETE.

MPI\_Win\_start(group, flag, win); MPI\_Put(..., win); MPI\_Win\_complete(win);

The call to MPI\_WIN\_COMPLETE does not return until the put call has completed at the origin; and the target window will be accessed by the put operation only after the call to MPI\_WIN\_START has matched a call to MPI\_WIN\_POST by the target process. This still leaves much choice to implementors. The call to MPI\_WIN\_START can block until the matching call to MPI\_WIN\_POST occurs at all target processes. One can also have implementations where the call to MPI\_WIN\_START is nonblocking, but the call to 29 MPI\_PUT blocks until the matching call to MPI\_WIN\_POST occurs; or implementations 30 where the first two calls are nonblocking, but the call to MPI\_WIN\_COMPLETE blocks until the call to MPI\_WIN\_POST occurred; or even implementations where all three calls can complete before any target process has called MPI\_WIN\_POST—the data put must be buffered, in this last case, so as to allow the put to complete at the origin ahead of its completion at the target. However, once the call to MPI\_WIN\_POST is issued, the sequence above must complete, without further dependencies.

MPI_WIN_POST(group, assert, win)			38
			39
IN	group	group of origin processes (handle)	40
IN	assert	program assertion (integer)	41
IN	win	window object (handle)	42
			43
			44
C binding			45
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)			46
Fortran 2008 binding			47
MPI_Win_post(group, assert, win, ierror)			48

**Unofficial Draft for Comment Only** 

1

2

10

11

12

13

14

15

16

1718

19 20

21

22 23

24

25

26

27

28

31

32

33

34

35

1 2 3	INTEG	MPI_Group), INTENT(IN) :: ER, INTENT(IN) :: assert	
4		MPI_Win), INTENT(IN) :: v ER, OPTIONAL, INTENT(OUT)	
5 6 7 8		inding OST(GROUP, ASSERT, WIN, I ER GROUP, ASSERT, WIN, IE	
9 10 11 12	in group $\sinh$	ould access the window with	e local window associated with win. Only processes RMA calls on win during this epoch. Each process PI_WIN_START. MPI_WIN_POST does not block.
13 14	MPI_WIN_	WAIT(win)	
15 16	IN	win	window object (handle)
17 18 19	C binding int MPI_W	g 'in_wait(MPI_Win win)	
20	Fortran 2	008 binding	
21	MPI_Win_w	ait(win, ierror)	
22	TYPE(	MPI_Win), INTENT(IN) :: v	7in 🔪
23	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
24	Fortrop b	in dia m	
25	Fortran b	0	
26		AIT(WIN, IERROR)	
27	INTEG	ER WIN, IERROR	
28	Comp	letes an RMA exposure epoch	started by a call to MPI_WIN_POST on win. This
29	call matche	es calls to MPI_WIN_COMPLE	TE(win) issued by each of the origin processes that
30	were grante	ed access to the window during	this epoch. The call to MPI_WIN_WAIT will block
31			MPLETE have occurred. This guarantees that all
32			heir RMA accesses to the local window. When the
33			have completed at the target window.
34			hese four functions. Process 0 puts data in the
35			ess 3 puts data in the window of process 2. Each
36			whose windows will be accessed; each post call lists
37		-	he local window. The figure illustrates a possible
38	0		nchronization; in a weak synchronization, the start,
39	put or com	plete calls may occur ahead c	the matching post calls.
40		*	
41	MPI WIN	TEST(win, flag)	
42			window object (box 11-)
43 $44$	IN	win	window object (handle)
44 45	OUT	flag	success flag (logical)
46			
47	C binding	5	
48	int MPI_W	<pre>in_test(MPI_Win win, int</pre>	*flag)

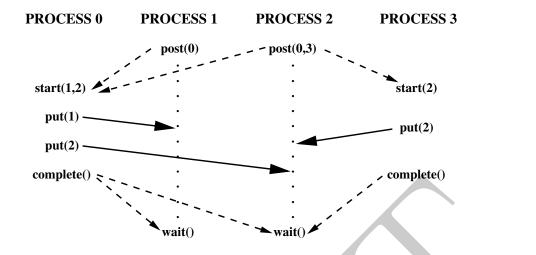


Figure 12.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

#### Fortran 2008 binding

```
MPI_Win_test(win, flag, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG
```

This is the nonblocking version of MPI\_WIN\_WAIT. It returns flag = true if all accesses to the local window by the group to which it was exposed by the corresponding MPI\_WIN\_POST call have been completed as signalled by matching MPI\_WIN\_COMPLETE calls, and flag = false otherwise. In the former case MPI\_WIN\_WAIT would have returned immediately. The effect of return of MPI\_WIN\_TEST with flag = true is the same as the effect of a return of MPI\_WIN\_WAIT. If flag = false is returned, then the call has no visible effect.

MPI\_WIN\_TEST should be invoked only where MPI\_WIN\_WAIT can be invoked. Once the call has returned flag = true, it must not be invoked anew, until the window is posted anew.

Assume that window win is associated with a "hidden" communicator wincomm, used for communication by the processes of win. The rules for matching of post and start calls and for matching complete and wait calls can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

# MPI\_WIN\_POST(group,0,win) initiates a nonblocking send with tag tag0 to each process in group, using wincomm. There is no need to wait for the completion of these sends.

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.

#### Unofficial Draft for Comment Only

- **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.*)

<sup>21</sup> <sup>22</sup> <sup>23</sup> <sup>24</sup> Advice to users. Assume a communication pattern that is represented by a directed graph  $G = \langle V, E \rangle$ , where  $V = \{0, ..., n-1\}$  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*<sub>i</sub>, ...), followed by a call to

<sup>25</sup> MPI\_WIN\_START( $outgroup_i, \ldots$ ), 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. <sup>27</sup> After the communications calls, each process that issued a start will issue a complete. <sup>28</sup> 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.*)

```
12.5.3 Lock
```

## MPI\_WIN\_LOCK(lock\_type, rank, assert, win)

37 38 39	IN	lock_type	either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED (state)
39 40	IN	rank	rank of locked window (non-negative integer)
41	IN	assert	program assertion (integer)
42 43	IN	win	window object (handle)
43 44 45 46	C binding		t rank, int assert, MPI_Win win)
47	Fortran 2	008 binding	

```
<sup>48</sup> MPI_Win_lock(lock_type, rank, assert, win, ierror)
```

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9 10

11

12

13

14

15

16

17

18

19

20

30

31 32

33 34 35

INTEGER, I	<pre>TENT(IN) :: lock_type, rank, assert</pre>		
TYPE(MPI_Win), INTENT(IN) :: win			
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
Fortran bindin	4	ł	
	CK_TYPE, RANK, ASSERT, WIN, IERROR)	<b>,</b>	
	(_TYPE, RANK, ASSERT, WIN, IERROR)	;	
INIEGER EO			
Starts an RM	A access epoch. The window at the process with rank rank can be accessed $^{8}$		
· *	is on win during that epoch. Multiple RMA access epochs (with calls $^{9}$		
	K) can occur simultaneously; however, each access epoch must target a		
different process.	11		
MPI_WIN_LOCK	All(assert_win)		
IN asser	program assertion (integer)		
IN win	window object (handle)		
	18		
C binding	19		
int MPI_Win_lo	x_all(int assert, MPI_Win win) 20	0	
Fortran 2008 b	ading 2	1	
	L(assert, win, ierror)	2	
	TENT(IN) :: assert	3	
TYPE(MPI_Win), INTENT(IN) :: win			
	CIONAL, INTENT(OUT) :: ierror	5	
	24	6	
Fortran bindin	2'	7	
	L(ASSERT, WIN, IERROR)	8	
INTEGER AS	ERT, WIN, IERROR 29	9	
Starts an R	A access epoch to all processes in win, with a lock type of <sup>30</sup>	0	
	D. During the epoch, the calling process can access the window memory on <sup>33</sup>		
all processes in win by using RMA operations. A window locked with MPI_WIN_LOCK_ALL			
must be unlocke	must be unlocked with MPI_WIN_UNLOCK_ALL. This routine is not collective—the ALL		
refers to a lock o	all members of the group of the window.		
	33		
Advice to u	i i i i i i i i i i i i i i i i i i i		
	OCK and MPI_WIN_LOCK_ALL concurrently on the same window. These		
	the avoided by specifying the assertion MPI_MODE_NOCHECK when		
possible (se	Section 12.5.5). (End of advice to users.) $33$		
	4		
	41		
MPI_WIN_UNLO			
IN rank	rank of window (non-negative integer) 44		
	41	5	
IN win	window object (handle) 40	6	
C binding			

1 int MPI\_Win\_unlock(int rank, MPI\_Win win) 2 Fortran 2008 binding 3 MPI\_Win\_unlock(rank, win, ierror) 4 INTEGER, INTENT(IN) :: rank 5TYPE(MPI\_Win), INTENT(IN) :: win 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 7 8 Fortran binding 9 MPI\_WIN\_UNLOCK(RANK, WIN, IERROR) 10 INTEGER RANK, WIN, IERROR 11 Completes an RMA access epoch started by a call to MPI\_WIN\_LOCK on window win. 12RMA operations issued during this period will have completed both at the origin and at the 13 target when the call returns. 14 1516MPI\_WIN\_UNLOCK\_ALL(win) 17window object (handle) IN win 18 19 C binding 2021int MPI\_Win\_unlock\_all(MPI\_Win win) 22Fortran 2008 binding 23MPI\_Win\_unlock\_all(win, ierror) 24TYPE(MPI\_Win), INTENT(IN) :: win 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2627Fortran binding MPI\_WIN\_UNLOCK\_ALL(WIN, IERROR) 28INTEGER WIN, IERROR 29 30 Completes a shared RMA access epoch started by a call to MPI\_WIN\_LOCK\_ALL on  $^{31}$ window win. RMA operations issued during this epoch will have completed both at the 32 origin and at the target when the call returns. 33 34Locks are used to protect accesses to the locked target window effected by RMA calls 35 issued between the lock and unlock calls, and to protect load/store accesses to a locked local 36 or shared memory window executed between the lock and unlock calls. Accesses that are 37 protected by an exclusive lock will not be concurrent at the window site with other accesses 38 to the same window that are lock protected. Accesses that are protected by a shared lock 39 will not be concurrent at the window site with accesses protected by an exclusive lock to 40 the same window. 41 It is erroneous to have a window locked and exposed (in an exposure epoch) concur-42rently. For example, a process may not call MPI\_WIN\_LOCK to lock a target window if 43 the target process has called MPI\_WIN\_POST and has not yet called MPI\_WIN\_WAIT; it 44is erroneous to call MPI\_WIN\_POST while the local window is locked. 4546Rationale. An alternative is to require MPI to enforce mutual exclusion between 47 exposure epochs and locking periods. But this would entail additional overheads 48 when locks or active target synchronization do not interact in support of those rare

interactions between the two mechanisms. The programming style that we encourage here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

Advice to users. Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (*End of advice to users.*)

Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI\_ALLOC\_MEM (Section 9.2), MPI\_WIN\_ALLOCATE (Section 12.2.2), MPI\_WIN\_ALLOCATE\_SHARED (Section 12.2.3), or attached with MPI\_WIN\_ATTACH (Section 12.2.4). Locks can be used portably only in such memory.

*Rationale.* The implementation of passive target communication when memory is not shared may require an asynchronous software agent. Such an agent can be implemented more easily, and can achieve better performance, if restricted to specially allocated memory. It can be avoided altogether if shared memory is used. It seems natural to impose restrictions that allows one to use shared memory for third party communication in shared memory machines.

(End of rationale.)

Consider the sequence of calls in the example below.

Example 12.5 Use of MPI\_WIN\_LOCK and MPI\_WIN\_UNLOCK.

MPI\_Win\_lock(MPI\_LOCK\_EXCLUSIVE, rank, assert, win); MPI\_Put(..., rank, ..., win);

MPI\_Win\_unlock(rank, win);

The call to MPI\_WIN\_UNLOCK will not return until the put transfer has completed at the origin and at the target. This still leaves much freedom to implementors. The call to MPI\_WIN\_LOCK may block until an exclusive lock on the window is acquired; or, the first two calls may not block, while MPI\_WIN\_UNLOCK blocks until a lock is acquired—the 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.

#### 12.5.4 Flush and Sync

All flush and sync functions can be called only within passive target epochs.

MPI\_WIN\_FLUSH(rank, win)

IN	rank	rank of target window (non-negative integer)
IN	win	window object (handle)

```
C binding
```

1

 $\mathbf{2}$ 

3

4

5

6

7

8 9

10

11

12

13

14 15

16

17

18

19

20 21

22 23

 $^{24}$ 

25 26

27

28

29

30

31

32

33

34

35

36

37 38

39 40

41 42 43

```
1
     int MPI_Win_flush(int rank, MPI_Win win)
\mathbf{2}
     Fortran 2008 binding
3
     MPI_Win_flush(rank, win, ierror)
4
          INTEGER, INTENT(IN) :: rank
5
          TYPE(MPI_Win), INTENT(IN) :: win
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     Fortran binding
9
     MPI_WIN_FLUSH(RANK, WIN, IERROR)
10
          INTEGER RANK, WIN, IERROR
11
          MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling
12
     process to the target rank on the specified window. The operations are completed both at
13
     the origin and at the target.
14
15
16
     MPI_WIN_FLUSH_ALL(win)
17
                                             window object (handle)
       IN
                 win
18
19
     C binding
20
21
     int MPI_Win_flush_all(MPI_Win win)
22
     Fortran 2008 binding
23
     MPI_Win_flush_all(win, ierror)
24
          TYPE(MPI_Win), INTENT(IN) :: win
25
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     Fortran binding
     MPI_WIN_FLUSH_ALL(WIN, IERROR)
28
          INTEGER WIN, IERROR
29
30
          All RMA operations issued by the calling process to any target on the specified window
^{31}
     prior to this call and in the specified window will have completed both at the origin and at
32
     the target when this call returns.
33
34
35
     MPI_WIN_FLUSH_LOCAL(rank, win)
36
       IN
                                             rank of target window (non-negative integer)
                 rank
37
       IN
                 win
                                             window object (handle)
38
39
40
     C binding
41
     int MPI_Win_flush_local(int rank, MPI_Win win)
42
     Fortran 2008 binding
43
     MPI_Win_flush_local(rank, win, ierror)
44
          INTEGER, INTENT(IN) :: rank
45
          TYPE(MPI_Win), INTENT(IN) :: win
46
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

Fortran binding MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR) INTEGER RANK, WIN, IERROR	1 2 3
Locally completes at the origin all outstanding RMA operations initiated by the calling process to the target process specified by rank on the specified window. For example, after this routine completes, the user may reuse any buffers provided to put, get, or accumulate operations.	4 5 6 7 8 9
MPI_WIN_FLUSH_LOCAL_ALL(win)	10
IN win window object (handle)	11
	12 13
C binding int MPI_Win_flush_local_all(MPI_Win win)	13 14 15
	16
Fortran 2008 binding MPI_Win_flush_local_all(win, ierror)	17
TYPE(MPI_Win), INTENT(IN) :: win	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
Fortron binding	20
Fortran binding MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)	21 22
INTEGER WIN, IERROR	23
	24
All RMA operations issued to any target prior to this call in this window will have	25
completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.	26
MPI_WIN_SYNC(win)	27 28
	29
IN win window object (handle)	30
C binding	31
int MPI_Win_sync(MPI_Win win)	32
	33
Fortran 2008 binding	34 35
MPI_Win_sync(win, ierror)	36
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
INTEGER, OPTIONAL, INTENI(001) TETTOT	38
Fortran binding	39
MPI_WIN_SYNC(WIN, IERROR)	40
INTEGER WIN, IERROR	41
The call MPI_WIN_SYNC synchronizes the private and public window copies of win.	42
For the purposes of synchronizing the private and public window, $MPI\_WIN\_SYNC$ has the	43
effect of ending and reopening an access and exposure epoch on the window (note that it	44 45
does not actually end an epoch or complete any pending MPI RMA operations).	45

## 12.5.5 Assertions

The assert argument in the calls MPI\_WIN\_POST, MPI\_WIN\_START, MPI\_WIN\_FENCE, MPI\_WIN\_LOCK, and MPI\_WIN\_LOCK\_ALL is used to provide assertions on the context of the call that may be used to optimize performance. The assert argument does not change program semantics if it provides correct information on the program—it is erroneous to provide incorrect information. Users may always provide assert = 0 to indicate a general case where no guarantees are made.

Advice to users. Many implementations may not take advantage of the information in assert; some of the information is relevant only for noncoherent shared memory machines. Users should consult their implementation's manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations whenever available. (End of advice to users.)

Advice to implementors. Implementations can always ignore the assert argument. Implementors should document which assert values are significant on their implementation. (End of advice to implementors.)

assert is the bit vector OR of zero or more of the following integer constants:

MPI\_MODE\_NOCHECK, MPI\_MODE\_NOSTORE, MPI\_MODE\_NOPUT,

MPI\_MODE\_NOPRECEDE, and MPI\_MODE\_NOSUCCEED. The significant options are listed
 below for each call.

Advice to users. C/C++ users can use bit vector OR(|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (End of advice to users.)

## MPI\_WIN\_START:

MPI\_MODE\_NOCHECK—the matching calls to MPI\_WIN\_POST have already completed on all target processes when the call to MPI\_WIN\_START is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit in the code. However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.

## MPI\_WIN\_POST:

- MPI\_MODE\_NOCHECK—the matching calls to MPI\_WIN\_START have not yet occurred on any origin processes when the call to MPI\_WIN\_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
- <sup>45</sup> 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.

1 2 3

4

5

6

7

8 9

10

11

12

13

14

15 16

17

18

19 20

21

25 26

27

28

29

30 31

32

33

34

35

36

37

38 39

40

41 42

43

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.)

#### 12.5.6 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.

## 12.6 Error Handling

#### 12.6.1 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 error handler MPI\_ERRORS\_ARE\_FATAL is associated with win during its creation. Users may change this default by explicitly associating a new error handler with win (see Section 9.3).

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

## 12.6.2 Error Classes

The error classes for one-sided communication are defined in Table 12.2. RMA routines may (and almost certainly will) use other MPI error classes, such as MPI\_ERR\_OP or MPI\_ERR\_RANK.

6	MPI_ERR_WIN	invalid win argument
7	MPI_ERR_BASE	invalid base argument
8	MPI_ERR_SIZE	invalid size argument
9	MPI_ERR_DISP	invalid disp argument
10	MPI_ERR_LOCKTYPE	invalid locktype argument
11	MPI_ERR_ASSERT	invalid assert argument
12	MPI_ERR_RMA_CONFLICT	conflicting accesses to window
13	MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls
14	MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case
15		of a window created with
16		MPI_WIN_CREATE_DYNAMIC, target memory is not
17		attached)
18	MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource
19		exhaustion)
20	MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the
21		group of the specified communicator cannot expose
22		shared memory)
23	MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called
24		function
25		

Table 12.2: Error classes in one-sided communication routines

## 12.7 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.

1. 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 23copy to private copy (6) is the same call that completes the put or accumulate operation in  $^{24}$ the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then 25the update of the public window copy is complete as soon as the updating process executed 26MPI\_WIN\_UNLOCK or MPI\_WIN\_UNLOCK\_ALL. In the RMA separate memory model, the 27update of a private copy in the process memory may be delayed until the target process 28executes a synchronization call on that window (6). Thus, updates to process memory can 29always be delayed in the RMA separate memory model until the process executes a suitable 30 synchronization call, while they must complete in the RMA unified model without additional 31synchronization calls. If fence or post-start-complete-wait synchronization is used, updates 32 to a public window copy can be delayed in both memory models until the window owner 33 executes a synchronization call. When passive target synchronization is used, it is necessary 34to update the public window copy even if the window owner does not execute any related 35 synchronization call. 36

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI\_WIN\_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI\_WIN\_FENCE(0, win2) makes these updates visible in the public copy of win2.

The behavior of some MPI RMA operations may be *undefined* in certain situations. For example, the result of several origin processes performing concurrent MPI\_PUT operations to the same target location is undefined. In addition, the result of a single origin process performing multiple MPI\_PUT operations to the same target location within the same access epoch is also undefined. The result at the target may have all of the data from one of the MPI\_PUT operations (the "last" one, in some sense), bytes from some of each of the 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

37

38

39

40

41

operations, or something else. In MPI-2, such operations were *erroneous*. That meant that
 an MPI implementation was permitted to raise an error. Thus, user programs or tools that
 used MPI RMA could not portably permit such operations, even if the application code could
 function correctly with such an undefined result. Starting with MPI-3, these operations are
 not erroneous, but do not have a defined behavior.

Rationale. As discussed in [7], requiring operations such as overlapping puts to be erroneous makes it difficult to use MPI RMA to implement programming models such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further, while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any implementation that enforces this rule, as it would require significant overhead. Thus, relaxing this condition does not impact existing implementations or applications. (*End* of rationale.)

Advice to implementors. Overlapping accesses are undefined. However, to assist users in debugging code, implementations may wish to provide a mode in which such operations are detected and reported to the user. Note, however, that starting with MPI-3, such operations must not raise an error. (*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 12.2.1.
  - 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.
- 40 The last constraint on correct RMA accesses may seem unduly restric-Rationale. 41 tive, as it forbids concurrent accesses to nonoverlapping locations in a window. The 42reason for this constraint is that, on some architectures, explicit coherence restor-43 ing operations may be needed at synchronization points. A different operation may 44be needed for locations that were updated by stores and for locations that were re-45motely updated by put or accumulate operations. Without this constraint, the MPI 46library would have to track precisely which locations in a window were updated by a 47 put or accumulate call. The additional overhead of maintaining such information is 48 considered prohibitive. (*End of rationale.*)

## Unofficial Draft for Comment Only

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

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 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 12.2.1.
- U5. A put or accumulate must not access a target window once a store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, 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	a store operation to a location in a window must not start once a put or accumulate
2	update to the same location in that target window has started and until the put or
3	accumulate update completes at the target.
4	
5	Advice to users. In the unified memory model, in the case where the window is
6	in shared memory, MPI_WIN_SYNC can be used to order store operations and make
7	store updates to the window visible to other processes and threads. Use of this
8	routine is necessary to ensure portable behavior when point-to-point, collective, or
9 10	shared memory synchronization is used in place of an RMA synchronization routine.
10	MPI_WIN_SYNC should be called by the writer before the non-RMA synchroniza- tion operation and by the reader after the non-RMA synchronization, as shown in
12	Example 12.21. (End of advice to users.)
12	Example 12.21. (End of davice to users.)
14	A program that violates these rules has undefined behavior.
15	I to a second and a second
16	Advice to users. A user can write correct programs by following the following rules:
17	fence: During each period between fence calls, each window is either updated by put
18	or accumulate calls, or updated by stores, but not both. Locations updated by
19	put or accumulate calls should not be accessed during the same period (with
20	the exception of concurrent updates to the same location by accumulate calls).
21 22	Locations accessed by get calls should not be updated during the same period.
22	<b>post-start-complete-wait:</b> A window should not be updated with store operations
24	while posted if it is being updated by put or accumulate calls. Locations updated
25	by put or accumulate calls should not be accessed while the window is posted
26	(with the exception of concurrent updates to the same location by accumulate
27	calls). Locations accessed by get calls should not be updated while the window
28	is posted.
29	With the post-start synchronization, the target process can tell the origin process
30	that its window is now ready for RMA access; with the complete-wait synchro-
31	nization, the origin process can tell the target process that it has finished its
32	RMA accesses to the window.
33	<b>lock:</b> Updates to the window are protected by exclusive locks if they may conflict.
34	Nonconflicting accesses (such as read-only accesses or accumulate accesses) are
35	protected by shared locks, both for load/store accesses and for RMA accesses.
36	changing window or synchronization mode: One can change synchronization
37	mode, or change the window used to access a location that belongs to two over-
38	lapping windows, when the process memory and the window copy are guaranteed
39	to have the same values. This is true after a local call to MPI_WIN_FENCE, if
40	RMA accesses to the window are synchronized with fences; after a local call
41	to MPI_WIN_WAIT, if the accesses are synchronized with post-start-complete-
42	wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK or
43	MPI_WIN_UNLOCK_ALL if the accesses are synchronized with locks.
44	
45	In addition, a process should not access the local buffer of a get operation until the
46	operation is complete, and should not update the local buffer of a put or accumulate
47	operation until that operation is complete.
48	

- 5:
- t у h ۱.
  - $\mathbf{s}$ d d e N
    - $\mathbf{s}$ )- $\mathbf{s}$
    - . e

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 12.6** 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.

		10
Process A:	Process B:	11
	window location X	12
		13
	MPI_Win_lock(EXCLUSIVE, B)	14
	store X /* local update to private copy of B */	15
	MPI_Win_unlock(B)	16
	/* now visible in public window copy */	17
		18
MPI_Barrier	MPI_Barrier	19
		20
MPI_Win_lock(EXCLUSIVE, ]	3)	21
MPI_Get(X) /* ok, read f:	rom public window */	22
MPI_Win_unlock(B)		23 24
		24 25
Example 12.7 In the RMA	unified model, although the public and private copies of the	26
-	aution must be used when combining load/stores and multi-	20
	hough the following example appears correct, the compiler or	28
	e to X after the barrier, possibly resulting in the MPI_GET	29
returning an incorrect value of		30
		31
Process A: P:	rocess B:	32
W	indow location X	33
		34
s	tore X /* update to private & public copy of B */	35
MPI_Barrier M	PI_Barrier	36
MPI_Win_lock_all		37
MPI_Get(X) /* ok, read f:	rom window */	38
MPI_Win_flush_local(B)		39
/* read value in X */		40
MPI_Win_unlock_all		41
		42
MPI_BARRIER provides proc	ess synchronization, but not memory synchronization. The	43

MPI\_BARRIER provides process synchronization, but not memory synchronization. The example could potentially be made safe through the use of compiler- and hardware-specific notations to ensure the store to X occurs before process B enters the MPI\_BARRIER. The use of one-sided synchronization calls, as shown in Example 12.6, also ensures the correct result.

 $\mathbf{2}$ 

Example 12.8 The following example demonstrates the reading of a memory location
 updated by a remote process (Rule 6) in the RMA separate memory model. Although
 the MPI\_WIN\_UNLOCK on process A and the MPI\_BARRIER ensure that the public copy
 on process B reflects the updated value of X, the call to MPI\_WIN\_LOCK by process B is
 necessary to synchronize the private copy with the public copy.

6 Process B: Process A: 7 window location X 8 9 MPI\_Win\_lock(EXCLUSIVE, B) 10 MPI\_Put(X) /\* update to public window \*/ 11 MPI\_Win\_unlock(B) 1213 MPI\_Barrier MPI\_Barrier 1415MPI\_Win\_lock(EXCLUSIVE, B) 16/\* now visible in private copy of B \*/ 17 load X 18 MPI\_Win\_unlock(B) 1920Note that in this example, the barrier is not critical to the semantic correctness. The 21use of exclusive locks guarantees a remote process will not modify the public copy after 22MPI\_WIN\_LOCK synchronizes the private and public copies. A polling implementation 23looking for changes in X on process B would be semantically correct. The barrier is required  $^{24}$ to ensure that process A performs the put operation before process B performs the load of 25Х. 26**Example 12.9** Similar to Example 12.7, the following example is unsafe even in the unified 27model, because the load of X can not be guaranteed to occur after the MPI\_BARRIER. While 28Process B does not need to explicitly synchronize the public and private copies through 29 MPI\_WIN\_LOCK as the MPI\_PUT will update both the public and private copies of the 30 window, the scheduling of the load could result in old values of X being returned. Compiler  $^{31}$ and hardware specific notations could ensure the load occurs after the data is updated, or 32 explicit one-sided synchronization calls can be used to ensure the proper result. 33 34Process A: Process B: 35 window location X 36 MPI\_Win\_lock\_all 37 MPI\_Put(X) /\* update to window \*/ 38 MPI\_Win\_flush(B) 39 40 MPI\_Barrier MPI\_Barrier 41 load X 42MPI\_Win\_unlock\_all 43 4445**Example 12.10** The following example further clarifies Rule 5. MPI\_WIN\_LOCK and 46MPI\_WIN\_LOCK\_ALL do not update the public copy of a window with changes to the 47private copy. Therefore, there is no guarantee that process A in the following sequence will 48 see the value of X as updated by the local store by process B before the lock.

Process A:	Process B:	1			
	window location X	2			
		3			
	store X /* update to private copy of B */	4			
	MPI_Win_lock(SHARED, B)	5			
MPI_Barrier	MPI_Barrier	6			
		7			
MPI_Win_lock(SHARED, B)		8			
MPI_Get(X) /* X may be the	X before the store */	9			
MPI_Win_unlock(B)		10			
	MPI_Win_unlock(B)	11			
	<pre>/* update on X now visible in public window */</pre>	12			
		13			
	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.					
		17			
<b>Example 12.11</b> Similar to the previous example, Rule 5 can have unexpected implications					
for general active target synchronization with the RMA separate memory model. It is <i>not</i>					
guaranteed that process B reads the value of X as per the local update by process A, because					
neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in					

the public window copy.

		23
Process A:	Process B:	24
window location X		25
window location Y		25 26
store Y		27
MPI_Win_post(A, B) /* Y vi	sible in public window */	28
MPI_Win_start(A)	MPI_Win_start(A)	29
		30
store X /* update to priva	te window */	31
boord n / apadoo oo priva		32
MPI_Win_complete	MPI_Win_complete	33
MPI_Win_wait	Mr1_win_compiete	34
	······································	35
/* update on X may not yet	visible in public window */	36
		37
MPI_Barrier	MPI_Barrier	38
		39
	MPI_Win_lock(EXCLUSIVE, A)	40
	MPI_Get(X) /* may return an obsolete value */	41
	MPI_Get(Y)	42
	MPI_Win_unlock(A)	43
		40

To allow process B to read the value of X stored by A the local store must be replaced by a local MPI\_PUT that updates the public window copy. Note that by this replacement X <sup>45</sup> may become visible in the private copy of process A only after the MPI\_WIN\_WAIT call in process A. The update to Y made before the MPI\_WIN\_POST call is visible in the public <sup>47</sup> window after the MPI\_WIN\_POST call and therefore process B will read the proper value <sup>48</sup>

#### Unofficial Draft for Comment Only

#### 1 of Y. The MPI\_GET(Y) call could be moved to the epoch started by the MPI\_WIN\_START $\mathbf{2}$ operation, and process B would still get the value stored by process A. 3 **Example 12.12** The following example demonstrates the interaction of general active 4 target synchronization with local read operations with the RMA separate memory model. 5Rules 5 and 6 do not guarantee that the private copy of X at process B has been updated 6 before the load takes place. 7 8 Process A: Process B: 9 window location X 10 11 MPI\_Win\_lock(EXCLUSIVE, B) 12MPI\_Put(X) /\* update to public window \*/ 13 MPI\_Win\_unlock(B) 14 15MPI\_Barrier MPI\_Barrier 1617MPI\_Win\_post(B) 18 MPI\_Win\_start(B) 19 20load X /\* access to private window \*/ 21/\* may return an obsolete value \*/ 22 23

MPI\_Win\_complete MPI\_Win\_wait

To ensure that the value put by process A is read, the local load must be replaced with a local MPI\_GET operation, or must be placed after the call to MPI\_WIN\_WAIT.

12.7.1 Atomicity 30

 $^{31}$ The outcome of concurrent accumulate operations to the same location with the same 32 predefined datatype is as if the accumulates were done at that location in some serial 33 order. Additional restrictions on the operation apply; see the info key accumulate\_ops in 34 Section 12.2.1. Concurrent accumulate operations with different origin and target pairs are 35 not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is 36 executed atomically. The effect of this lack of atomicity is limited: The previous correctness 37 conditions imply that a location updated by a call to an accumulate operation cannot be 38 accessed by a load or an RMA call other than accumulate until the accumulate operation has 39 completed (at the target). Different interleavings can lead to different results only to the 40 extent that computer arithmetics are not truly associative or commutative. The outcome 41 of accumulate operations with overlapping types of different sizes or target displacements 42is undefined.

43 44

45

24

2526

27

28 29

## 12.7.2 Ordering

46Accumulate calls enable element-wise atomic read and write to remote memory locations. 47MPI specifies ordering between accumulate operations from an origin process to the same 48 (or overlapping) memory locations at a target process on a per-datatype granularity. The

default ordering is strict ordering, which guarantees that overlapping updates from the same origin 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.

 $\overline{7}$ The default strict ordering may incur a significant performance penalty. MPI specifies the info key "accumulate\_ordering" to allow relaxation of the ordering semantics when specified 8 to any window creation function. The values for this key are as follows. If set to "none", 9 10 then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA in 11MPI-2 but has not been the default since MPI-3. The key can be set to a comma-separated 12list of required access orderings at the target. Allowed values in the comma-separated list  $^{13}$ are "rar", "war", "raw", and "waw" for read-after-read, write-after-read, read-after-write, and 14write-after-write ordering, respectively. These indicate whether operations of the specified 15type complete in the order they were issued. For example, "raw" means that any writes must 16complete at the target before subsequent reads. These ordering requirements apply only to 17 operations issued by the same origin process and targeting the same target process. The 18 default value for "accumulate\_ordering" is rar,raw,war,waw, which implies that writes complete 19at the target in the order in which they were issued, reads complete at the target before any 20writes that are issued after the reads, and writes complete at the target before any reads 21that are issued after the writes. Any subset of these four orderings can be specified. For example, if only read-after-read and write-after-write ordering is required, then the value of 22the "accumulate\_ordering" key could be set to rar, waw. The order of values is not significant. 23

Note that the above ordering semantics apply only to accumulate operations, not put and get. Put and get within an epoch are unordered.

### 12.7.3 Progress

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled it is guaranteed to complete. RMA calls must have local semantics, except when required for synchronization with other RMA calls.

There is some fuzziness in the definition of the time when a RMA communication becomes enabled. This fuzziness provides to the implementor more flexibility than with point-to-point communication. Access to a target window becomes enabled once the corresponding synchronization (such as MPI\_WIN\_FENCE or MPI\_WIN\_POST) has executed. On the origin process, an RMA communication may become enabled as soon as the corresponding put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. Once the communication is enabled both at the origin and at the target, the communication must complete.

Consider the code fragment in Example 12.4. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete. The data transfer may start as soon as the put call occurs, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 12.5. 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 12.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are

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

 $^{24}$ 

25

26 27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

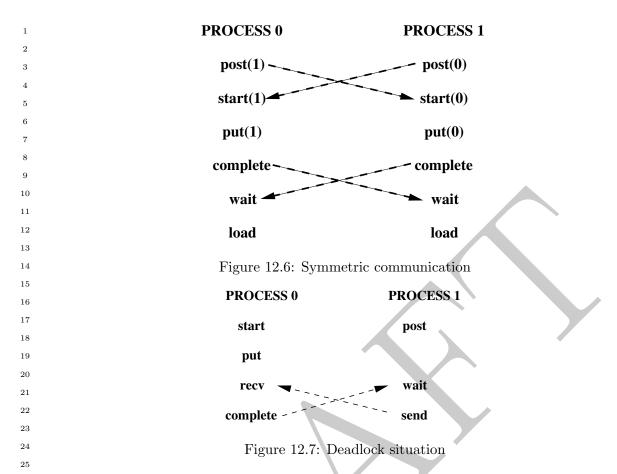
43

44

45

46

47



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 com-35plete and wait is not symmetric: the wait call blocks until the complete executes, but not 36 vice versa. Consider the code illustrated in Figure 12.7. This code will deadlock: the wait 37 of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until 38 process 1 calls send. Consider, on the other hand, the code illustrated in Figure 12.8. This 39 code will not deadlock. Once process 1 calls post, then the sequence start, put, complete 40 on process 0 can proceed to completion. Process 0 will reach the send call, allowing the 41 receive call of process 1 to complete. 42

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 12.8, 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.

### Unofficial Draft for Comment Only

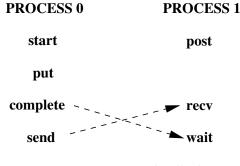


Figure 12.8: No deadlock

A similar issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call. Suppose that in the last example the send-receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not specify whether deadlock is avoided. Suppose that the blocking receive of process 1 is replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious. (*End of rationale*.)

### 12.7.4 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:

Source of Process 1	Source of Process 2	Executed in Process 2	40
bbbb = 777	buff = 999	reg_A:=999	41
call MPI_WIN_FENCE	call MPI_WIN_FENCE		42
call MPI_PUT(bbbb		stop appl.thread	43
into buff of process 2)		buff:=777 in PUT handler	44
		continue appl.thread	45
call MPI_WIN_FENCE	call MPI_WIN_FENCE		46
	ccc = buff	ccc:=reg_A	47
			48

1 2

Z

5 6

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

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.

3 This problem, which also afflicts in some cases send/receive communication, is discussed 4 more at length in Section 19.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran 6 compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should 8 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 10 compilers, please note the hints in Sections 19.1.10–19.1.20. Sections 19.1.17 to 19.1.17 11discuss several solutions for the problem in this example.

- 12.8 Examples
- 14 1516

17

18

19

29

39

**Example 12.13** 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.

```
20
     . . .
21
     while (!converged(A)) {
       update(A);
22
       MPI_Win_fence(MPI_MODE_NOPRECEDE,
                                             win);
23
24
       for(i=0; i < toneighbors; i++)</pre>
         MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
25
26
                                 todisp[i], 1, totype[i], win);
       MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
27
     }
28
```

The same code could be written with get rather than put. Note that, during the commu-30 nication phase, each window is concurrently read (as origin buffer of puts) and written (as  $^{31}$ target buffer of puts). This is OK, provided that there is no overlap between the target 32 buffer of a put and another communication buffer. 33

34**Example 12.14** Same generic example, with more computation/communication overlap. 35 We assume that the update phase is broken into two subphases: the first, where the "bound-36 ary," which is involved in communication, is updated, and the second, where the "core," 37 which neither uses nor provides communicated data, is updated. 38

```
. . .
40
     while (!converged(A)) {
41
       update_boundary(A);
42
       MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
43
       for(i=0; i < fromneighbors; i++)</pre>
44
         MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
45
                           fromdisp[i], 1, fromtype[i], win);
46
       update_core(A);
47
       MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
48
     }
```

1

 $\mathbf{2}$ 

5

 $\overline{7}$ 

9

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 12.15 Same code as in Example 12.13, rewritten using post-start-complete-wait.

```
...
while (!converged(A)) {
    update(A);
    MPI_Win_post(fromgroup, 0, win);
    MPI_Win_start(togroup, 0, win);
    for(i=0; i < toneighbors; i++)
        MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
            todisp[i], 1, totype[i], win);
    MPI_Win_complete(win);
    MPI_Win_wait(win);
}</pre>
```

Example 12.16 Same example, with split phases, as in Example 12.14.

```
while (!converged(A)) {
  update_boundary(A);
  MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
  MPI_Win_start(fromgroup, 0, win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
            fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_complete(win);
  MPI_Win_wait(win);
}</pre>
```

**Example 12.17** A checkerboard, or double buffer communication pattern, that allows more computation/communication overlap. Array A0 is updated using values of array A1, and vice versa. We assume that communication is symmetric: if process A gets data from process B, then process B gets data from process A. Window wini consists of array Ai.

```
...
if (!converged(A0,A1))
    MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
MPI_Barrier(comm0);
/* the barrier is needed because the start call inside the
loop uses the nocheck option */
while (!converged(A0, A1)) {
```

 $^{24}$ 

```
1
       /* communication on AO and computation on A1 */
\mathbf{2}
       update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
3
       MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
4
       for(i=0; i < fromneighbors; i++)</pre>
5
         MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
6
                      fromdisp0[i], 1, fromtype0[i], win0);
7
       update1(A1); /* local update of A1 that is
8
                         concurrent with communication that updates AO */
9
       MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
10
       MPI_Win_complete(win0);
11
       MPI_Win_wait(win0);
12
       /* communication on A1 and computation on A0 */
13
14
       update2(A0, A1); /* local update of A0 that depends on A1 (and A0) */
15
       MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
16
       for(i=0; i < fromneighbors; i++)</pre>
17
         MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
18
                       fromdisp1[i], 1, fromtype1[i], win1);
19
       update1(A0); /* local update of A0 that depends on A0 only,
                        concurrent with communication that updates A1 */
20
21
       if (!converged(A0,A1))
22
         MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
23
       MPI_Win_complete(win1);
24
       MPI_Win_wait(win1);
25
     }
26
         A process posts the local window associated with win0 before it completes RMA accesses
27
     to the remote windows associated with win1. When the wait(win1) call returns, then all
28
     neighbors of the calling process have posted the windows associated with win0. Conversely,
29
     when the wait(win0) call returns, then all neighbors of the calling process have posted the
30
     windows associated with win1. Therefore, the nocheck option can be used with the calls to
^{31}
     MPI_WIN_START.
32
         Put calls can be used, instead of get calls, if the area of array AO (resp. A1) used by
33
     the update(A1, A0) (resp. update(A0, A1)) call is disjoint from the area modified by the
34
     RMA communication. On some systems, a put call may be more efficient than a get call,
35
     as it requires information exchange only in one direction.
36
         In the next several examples, for conciseness, the expression
37
38
```

```
z = MPI_Get_accumulate(...)
```

39

44

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 and MPI\_COMM\_SIZE.
 Process B... refers to any process other than A.

Example 12.18 The following example implements a naive, non-scalable counting sema phore. 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

### Unofficial Draft for Comment Only

public and private copies of windows, MPI\_ACCUMULATE and MPI\_GET\_ACCUMULATE are used to write to or read from the local public copy.

<pre>Process A: MPI_Win_lock_all window location X X=MPI_Comm_size() MPI_Win_sync</pre>	Process B: MPI_Win_lock_all
MPI_Barrier	MPI_Barrier
MPI_Accumulate(X, MPI_SUM, -1)	MPI_Accumulate(X, MPI_SUM, -1)
stack variable z	stack variable z
do	do
<pre>z = MPI_Get_accumulate(X,</pre>	<pre>z = MPI_Get_accumulate(X,</pre>
MPI_NO_OP, 0)	MPI_NO_OP, 0)
MPI_Win_flush(A)	MPI_Win_flush(A)
while(z!=0)	while(z!=0)
MPI_Win_unlock_all	MPI_Win_unlock_all

**Example 12.19** 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.

operations can be used to read from or write	to the public copy of the window.	
		28
Process A:	Process B:	29
window location X	window location Y	30
window location T	P	31
		32
MPI_Win_lock_all	MPI_Win_lock_all	33
X=1	Y=1	34
MPI_Win_sync	MPI_Win_sync	35
MPI_Barrier	MPI_Barrier	36
MPI_Accumulate(T, MPI_REPLACE, 1)	<pre>MPI_Accumulate(T, MPI_REPLACE, 0)</pre>	37
stack variables t,y	stack variable t,x	38
t=1	t=0	39
y=MPI_Get_accumulate(Y,	<pre>x=MPI_Get_accumulate(X,</pre>	40
MPI_NO_OP, 0)	MPI_NO_OP, O)	41
while(y==1 && t==1) do	while(x==1 && t==0) do	42
<pre>y=MPI_Get_accumulate(Y,</pre>	<pre>x=MPI_Get_accumulate(X,</pre>	43
MPI_NO_OP, 0)	MPI_NO_OP, 0)	44
<pre>t=MPI_Get_accumulate(T,</pre>	t=MPI_Get_accumulate(T,	45
MPI_NO_OP, 0)	MPI_NO_OP, O)	46
MPI_Win_flush_all	MPI_Win_flush(A)	47
done	done	48

Unofficial Draft for Comment Only

 $^{24}$ 

```
1
     // critical region
                                                // critical region
\mathbf{2}
     MPI_Accumulate(X, MPI_REPLACE, 0)
                                                MPI_Accumulate(Y, MPI_REPLACE, 0)
3
     MPI_Win_unlock_all
                                                MPI_Win_unlock_all
4
5
     Example 12.20 Implementing a critical region between multiple processes with compare
6
     and swap. The call to MPI_WIN_SYNC is necessary on Process A after local initialization
7
     of A to guarantee the public copy has been updated with the initialization value found in
8
     the private copy. It would also be valid to call MPI_ACCUMULATE with MPI_REPLACE to
9
     directly initialize the public copy. A call to MPI_WIN_FLUSH would be necessary to assure
10
     A in the public copy of Process A had been updated before the barrier.
11
12
     Process A:
                                                 Process B...:
     MPI_Win_lock_all
                                                 MPI_Win_lock_all
13
14
     atomic location A
15
     A=0
16
     MPI_Win_sync
17
                                                 MPI_Barrier
     MPI_Barrier
                                                 stack variable r=1
18
     stack variable r=1
19
     while(r != 0) do
                                                 while(r != 0) do
                                                    r = MPI_Compare_and_swap(A, 0, 1)
       r = MPI_Compare_and_swap(A, 0, 1)
20
       MPI_Win_flush(A)
                                                    MPI_Win_flush(A)
21
22
     done
                                                 done
                                                 // critical region
23
     // critical region
^{24}
     r = MPI_Compare_and_swap(A, 1, 0)
                                                 r = MPI_Compare_and_swap(A, 1, 0)
     MPI_Win_unlock_all
                                                 MPI_Win_unlock_all
25
26
27
     Example 12.21 The following example demonstrates the proper synchronization in the
28
     unified memory model when a data transfer is implemented with load and store in the case
29
     of windows in shared memory (instead of MPI_PUT or MPI_GET) and the synchronization
30
     between processes is performed using point-to-point communication. The synchronization
^{31}
     between processes must be supplemented with a memory synchronization through calls to
32
     MPI_WIN_SYNC, which act locally as a processor-memory barrier. In Fortran, if
33
     MPI_ASYNC_PROTECTS_NONBLOCKING is .FALSE. or the variable X is not declared as
34
     ASYNCHRONOUS, reordering of the accesses to the variable X must be prevented with
35
     MPI_F_SYNC_REG operations. (No equivalent function is needed in C.)
36
          The variable X is contained within a shared memory window and X corresponds to
37
     the same memory location at both processes. The MPI_WIN_SYNC operation performed
38
     by process A ensures completion of the load/store operations issued by process A. The
39
     MPI_WIN_SYNC operation performed by process B ensures that process A's updates to X
40
     are visible to process B.
41
42
     Process A:
                                          Process B:
43
     MPI_WIN_LOCK_ALL(
                                          MPI_WIN_LOCK_ALL(
44
            MPI_MODE_NOCHECK,win)
                                                 MPI_MODE_NOCHECK,win)
45
46
     DO ...
                                          DO ...
47
       X=...
48
```

MPI_F_SYNC_REG(X)		1
MPI_WIN_SYNC(win)		2
MPI_SEND	MPI_RECV	3
	MPI_WIN_SYNC(win)	4
	MPI_F_SYNC_REG(X)	5
		6
	print X	7
	-	8
	MPI_F_SYNC_REG(X)	9
MPI_RECV	MPI_SEND	10
MPI_F_SYNC_REG(X)		11
END DO	END DO	12
		13
MPI_WIN_UNLOCK_ALL(win)	MPI_WIN_UNLOCK_ALL(win)	14
		15
Example 12.22 The following exam	ple shows how request-based operations can be used	16
	itation. Each process fetches, processes, and writes	17
	Instead of a single buffer, M local buffers are used to	18
allow up to M communication operation		19
		20
int i i.		21

```
int
             i, j;
                                                                                     22
MPI_Win
             win;
MPI_Request put_req[M] = { MPI_REQUEST_NULL };
                                                                                     23
MPI_Request get_req;
                                                                                      24
                                                                                      25
double
             *baseptr;
                                                                                      26
double
             data[M][N];
                                                                                     27
MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
                                                                                     28
  MPI_COMM_WORLD, &baseptr, &win);
                                                                                     29
                                                                                      30
                                                                                      ^{31}
MPI_Win_lock_all(0, win);
                                                                                      32
                                                                                      33
for (i = 0; i < NSTEPS; i+
                                                                                     34
 if (i<M)
   j=i;
                                                                                     35
                                                                                     36
 else
                                                                                     37
   MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
                                                                                      38
                                                                                      39
 MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
                                                                                      40
           &get_req);
                                                                                      ^{41}
 MPI_Wait(&get_req,MPI_STATUS_IGNORE);
                                                                                     42
 compute(i, data[j], ...);
                                                                                     43
 MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
                                                                                      44
           &put_req[j]);
}
                                                                                      45
                                                                                      46
                                                                                      47
MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
                                                                                      48
MPI_Win_unlock_all(win);
```

Example 12.23 The following example constructs a distributed shared linked list using dynamic windows. Initially process 0 creates the head of the list, attaches it to the window, and broadcasts the pointer to all processes. All processes then concurrently append N new elements to the list. When a process attempts to attach its element to the tail of the list it may discover that its tail pointer is stale and it must chase ahead to the new tail before the element can be attached. This example requires some modification to work in an environment where the layout of the structures is different on different processes.

```
9
     . . .
10
     #define NUM_ELEMS 10
11
     #define LLIST_ELEM_NEXT_RANK ( offsetof(llist_elem_t, next) + \
12
                                      offsetof(llist_ptr_t, rank) )
13
     #define LLIST_ELEM_NEXT_DISP ( offsetof(llist_elem_t, next) )
14
                                      offsetof(llist_ptr_t,
15
                                                              disp) )
16
17
     /* Linked list pointer */
     typedef struct {
18
19
       MPI_Aint disp;
20
       int
                rank;
     } llist_ptr_t;
21
22
23
     /* Linked list element */
24
     typedef struct {
25
       llist_ptr_t next;
26
       int value;
     } llist_elem_t;
27
28
     const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM, -1 };
29
30
     /* List of locally allocated list elements. */
^{31}
     static llist_elem_t **my_elems = NULL;
32
     static int my_elems_size = 0;
33
34
     static int my_elems_count = 0;
35
     /* Allocate a new shared linked list element */
36
     MPI_Aint alloc_elem(int value, MPI_Win win) {
37
38
       MPI_Aint disp;
       llist_elem_t *elem_ptr;
39
40
       /* Allocate the new element and register it with the window */
41
42
       MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
       elem_ptr->value = value;
43
       elem_ptr->next = nil;
44
       MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
45
46
47
       /* Add the element to the list of local elements so we can free
48
          it later. */
```

```
1
 if (my_elems_size == my_elems_count) {
                                                                                     \mathbf{2}
    my_elems_size += 100;
                                                                                     3
   my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
 }
                                                                                     4
 my_elems[my_elems_count] = elem_ptr;
                                                                                     5
                                                                                     6
 my_elems_count++;
                                                                                     7
 MPI_Get_address(elem_ptr, &disp);
                                                                                     9
 return disp;
                                                                                     10
}
                                                                                     11
int main(int argc, char *argv[]) {
                                                                                     12
                                                                                     13
 int
                 procid, nproc, i;
                                                                                     14
 MPI_Win
                llist_win;
                                                                                     15
 llist_ptr_t
                head_ptr, tail_ptr;
                                                                                     16
                                                                                     17
 MPI_Init(&argc, &argv);
                                                                                     18
                                                                                     19
 MPI_Comm_rank(MPI_COMM_WORLD, &procid);
 MPI_Comm_size(MPI_COMM_WORLD, &nproc);
                                                                                     20
                                                                                     21
 MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
                                                                                     22
                                                                                     23
 /* Process 0 creates the head node */
                                                                                     24
                                                                                     25
 if (procid == 0)
                                                                                     26
    head_ptr.disp = alloc_elem(-1, llist_win);
                                                                                     27
 /* Broadcast the head pointer to everyone */
                                                                                     28
                                                                                     29
 head_ptr.rank = 0;
                                                                                     30
 MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
                                                                                     31
 tail_ptr = head_ptr;
                                                                                     32
  /* Lock the window for shared access to all targets */
                                                                                     33
                                                                                     34
 MPI_Win_lock_all(0, llist_win);
                                                                                     35
  /* All processes concurrently append NUM_ELEMS elements to the list */
                                                                                     36
                                                                                     37
 for (i = 0; i < NUM_ELEMS; i++) {</pre>
                                                                                     38
    llist_ptr_t new_elem_ptr;
                                                                                     39
    int success;
                                                                                     40
                                                                                     41
    /* Create a new list element and attach it to the window */
                                                                                     42
    new_elem_ptr.rank = procid;
    new_elem_ptr.disp = alloc_elem(procid, llist_win);
                                                                                     43
                                                                                     44
                                                                                     45
    /* Append the new node to the list. This might take multiple
                                                                                     46
       attempts if others have already appended and our tail pointer
                                                                                     47
       is stale. */
                                                                                     48
    do {
```

```
1
           llist_ptr_t next_tail_ptr = nil;
2
3
           MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
4
                (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
5
                MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
6
                llist_win);
7
8
           MPI_Win_flush(tail_ptr.rank, llist_win);
9
           success = (next_tail_ptr.rank == nil.rank);
10
11
           if (success) {
12
             MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
13
                  MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
14
                  MPI_AINT, MPI_REPLACE, llist_win);
15
16
             MPI_Win_flush(tail_ptr.rank, llist_win);
17
             tail_ptr = new_elem_ptr;
18
19
           } else {
20
             /* Tail pointer is stale, fetch the displacement.
                                                                    May take
21
                 multiple tries if it is being updated. */
22
             do {
23
                MPI_Get_accumulate(NULL, 0, MPI_AINT, &next_tail_ptr.disp,
24
                    1, MPI_AINT, tail_ptr.rank,
                    MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
25
26
                    1, MPI_AINT, MPI_NO_OP, llist_win);
27
28
                MPI_Win_flush(tail_ptr.rank, llist_win);
29
             } while (next_tail_ptr.disp == nil.disp);
30
             tail_ptr = next_tail_ptr;
31
           }
32
         } while (!success);
33
       }
34
       MPI_Win_unlock_all(llist_win);
35
36
       MPI_Barrier(MPI_COMM_WORLD);
37
38
       /* Free all the elements in the list */
39
       for ( ; my_elems_count > 0; my_elems_count--) {
40
         MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
41
         MPI_Free_mem(my_elems[my_elems_count-1]);
42
       }
43
       MPI_Win_free(&llist_win);
44
     . . .
45
46
47
```

CHAPTER 12. ONE-SIDED COMMUNICATIONS

620

### Chapter 13

## **External Interfaces**

### 13.1 Introduction

This chapter contains 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 13.3 deals with setting the information found in status. This functionality is needed for generalized requests.

### 13.2 Generalized Requests

The goal of generalized requests is to allow users to define new nonblocking operations. Such an outstanding nonblocking operation is represented by a (generalized) request. A fundamental property of nonblocking operations is that progress toward the completion of this operation occurs asynchronously, i.e., concurrently with normal program execution. Typically, this requires execution of code concurrently with the execution of the user code, e.g., in a separate thread or in a signal handler. Operating systems provide a variety of mechanisms in support of concurrent execution. MPI does not attempt to standardize or to replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as MPI\_WAIT or MPI\_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

*Rationale.* It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the application. For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI through a call to

 $45 \\ 46$ 

```
1
     MPI_GREQUEST_COMPLETE when the operation completes. MPI maintains the "comple-
\mathbf{2}
     tion" status of generalized requests. Any other request state has to be maintained by the
3
     user.
4
          A new generalized request is started with
5
6
     MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)
7
8
       IN
                                             callback function invoked when request status is
                 query_fn
9
                                             queried (function)
10
       IN
                 free_fn
                                             callback function invoked when request is freed
11
                                             (function)
12
                                             callback function invoked when request is cancelled
       IN
                 cancel_fn
13
                                             (function)
14
15
       IN
                 extra_state
                                             extra state
16
       OUT
                                             generalized request (handle)
                 request
17
18
     C binding
19
     int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
20
                    MPI_Grequest_free_function *free_fn,
21
                    MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
22
                    MPI_Request *request)
23
^{24}
     Fortran 2008 binding
25
     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
26
                     ierror)
27
          PROCEDURE(MPI_Grequest_query_function), INTENT(IN) :: query_fn
28
          PROCEDURE(MPI_Grequest_free_function), INTENT(IN) :: free_fn
29
          PROCEDURE(MPI_Grequest_cancel_function), INTENT(IN) :: cancel_fn
30
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
31
          TYPE(MPI_Request), INTENT(OUT) :: request
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     Fortran binding
34
     MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
35
                     IERROR)
36
          EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
37
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
38
          INTEGER REQUEST, IERROR
39
40
41
           Advice to users.
                              Note that a generalized request is of the same type as regular
42
           requests, in C and Fortran. (End of advice to users.)
43
         The call starts a generalized request and returns a handle to it in request.
44
          The syntax and meaning of the callback functions are listed below. All callback func-
45
     tions are passed the extra_state argument that was associated with the request by the
46
47
     starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined
48
     state for the request.
```

```
1
    In C, the query function is
                                                                                          2
typedef int MPI_Grequest_query_function(void *extra_state,
               MPI_Status *status);
in Fortran with the mpi_f08 module
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
    TYPE(MPI_Status) :: status
                                                                                          9
    INTEGER :: ierror
                                                                                          10
                                                                                          11
in Fortran with the mpi module and mpif.h
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
                                                                                          12
                                                                                          13
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                          14
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
                                                                                          15
    The query_fn function computes the status that should be returned for the generalized
                                                                                          16
request. The status also includes information about successful/unsuccessful cancellation of
                                                                                          17
the request (result to be returned by MPI_TEST_CANCELLED).
                                                                                          18
    The query_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that
                                                                                          19
completed the generalized request associated with this callback. The callback function is
                                                                                          20
also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when
                                                                                          21
the call occurs. In both cases, the callback is passed a reference to the corresponding
                                                                                          22
status variable passed by the user to the MPI call; the status set by the callback function
                                                                                          23
is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or
                                                                                          ^{24}
MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI
                                                                                          25
will pass a valid status object to query fn, and this status will be ignored upon return of the
                                                                                          26
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE
                                                                                          27
is called on the request; it may be invoked several times for the same generalized request,
                                                                                          28
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also
                                                                                          29
that a call to MPI_{WAIT|TEST}{SOME|ALL} may cause multiple invocations of query_fn
                                                                                          30
callback functions, one for each generalized request that is completed by the MPI call. The
                                                                                          ^{31}
order of these invocations is not specified by MPI.
                                                                                          32
    In C, the free function is
                                                                                          33
typedef int MPI_Grequest_free_function(void *extra_state);
                                                                                          34
                                                                                          35
in Fortran with the mpi_f08 module
                                                                                          36
ABSTRACT INTERFACE
                                                                                          37
  SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
                                                                                          38
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                          39
    INTEGER :: ierror
                                                                                          40
in Fortran with the mpi module and mpif.h
                                                                                          41
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
                                                                                          42
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                          43
    INTEGER IERROR
                                                                                          44
                                                                                          45
The free_fn function is invoked to clean up user-allocated resources when the generalized
                                                                                          46
request is freed.
                                                                                          47
```

<sup>1</sup> The free\_fn callback is invoked by the MPI\_{WAIT|TEST}{ANY|SOME|ALL} call that <sup>2</sup> completed the generalized request associated with this callback. free\_fn is invoked after <sup>3</sup> the call to query\_fn for the same request. However, if the MPI call completed multiple <sup>4</sup> generalized requests, the order in which free\_fn callback functions are invoked is not specified <sup>5</sup> by MPI.

6 The free\_fn callback is also invoked for generalized requests that are freed by a call 7to MPI\_REQUEST\_FREE (no call to MPI\_{WAIT|TEST}{ANY|SOME|ALL} will occur for 8 such a request). In this case, the callback function will be called either in the MPI call 9 MPI\_REQUEST\_FREE(request), or in the MPI call MPI\_GREQUEST\_COMPLETE(request), 10 whichever happens last, i.e., in this case the actual freeing code is executed as soon as both 11calls MPI\_REQUEST\_FREE and MPI\_GREQUEST\_COMPLETE have occurred. The request 12is not deallocated until after free\_fn completes. Note that free\_fn will be invoked only once 13per request by a correct program.

Advice to users. Calling MPI\_REQUEST\_FREE(request) will cause the request handle 15to be set to MPI\_REQUEST\_NULL. This handle to the generalized request is no longer 16valid. However, user copies of this handle are valid until after free\_fn completes since 17 MPI does not deallocate the object until then. Since free\_fn is not called until after 18 MPI\_GREQUEST\_COMPLETE, the user copy of the handle can be used to make this 19 call. Users should note that MPI will deallocate the object after free\_fn executes. At 20this point, user copies of the request handle no longer point to a valid request. MPI will 21not set user copies to MPI\_REQUEST\_NULL in this case, so it is up to the user to avoid 22 accessing this stale handle. This is a special case in which MPI defers deallocating the 23object until a later time that is known by the user. (End of advice to users.)  $^{24}$ 

In C, the cancel function is

typedef int MPI\_Grequest\_cancel\_function(void \*extra\_state, int complete);

```
^{28} in Fortran with the mpi_f08 module
```

<sup>29</sup> ABSTRACT INTERFACE

<sup>30</sup> SUBROUTINE MPI\_Grequest\_cancel\_function(extra\_state, complete, ierror)
 <sup>31</sup> INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state
 <sup>32</sup> LOGICAL :: complete
 <sup>33</sup> INTEGER :: ierror

34 INTEGER :: Terror

```
in Fortran with the mpi module and mpif.h
```

```
    SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    LOGICAL COMPLETE
```

39 INTEGER IERROR

The cancel\_fn function is invoked to start the cancelation of a generalized request. It is called by MPI\_CANCEL(request). MPI passes complete = true to the callback function if MPI\_GREQUEST\_COMPLETE was already called on the request, and complete = false otherwise.

<sup>44</sup> All callback functions return an error code. The code is passed back and dealt with as <sup>45</sup> appropriate for the error code by the MPI function that invoked the callback function. For <sup>46</sup> example, if error codes are returned then the error code returned by the callback function <sup>47</sup> will be returned by the MPI function that invoked the callback function. In the case of <sup>48</sup> an MPI\_{WAIT|TEST}{ANY} call that invokes both query\_fn and free\_fn, the MPI call will

### Unofficial Draft for Comment Only

14

25

26

return the error code returned by the last callback, namely free\_fn. If one or more of the requests in a call to MPI\_{WAIT|TEST}{SOME|ALL} failed, then the MPI call will return MPI\_ERR\_IN\_STATUS. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its free\_fn callback function. However, if the MPI function was passed MPI\_STATUSES\_IGNORE, then the individual error codes returned by each callback functions will be lost.

Advice to users. query\_fn must not set the error field of status since query\_fn may be called by MPI\_WAIT or MPI\_TEST, in which case the error field of status should not change. The MPI library knows the "context" in which query\_fn is invoked and can decide correctly when to put the returned error code in the error field of status. (End of advice to users.)

MPI\_GREQUEST\_COMPLETE(request)

INOUT request

generalized request (handle)

### C binding

int MPI\_Grequest\_complete(MPI\_Request request)

### Fortran 2008 binding

```
MPI_Grequest_complete(request, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### Fortran binding

MPI\_GREQUEST\_COMPLETE(REQUEST, IERROR)

INTEGER REQUEST, IERROR

The call informs MPI that the operations represented by the generalized request request are complete (see definitions in Section 2.4). A call to MPI\_WAIT(request, status) will return and a call to MPI\_TEST(request, flag, status) will return flag = true only after a call to MPI\_GREQUEST\_COMPLETE has declared that these operations are complete.

MPI imposes no restrictions on the code executed by the callback functions. However, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as MPI\_TEST, MPI\_REQUEST\_FREE, or MPI\_CANCEL still hold. For example, these calls are supposed to be local and nonblocking. Therefore, the callback functions query\_fn, free\_fn, or cancel\_fn should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once MPI\_CANCEL is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired "local" semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

Advice to implementors. A call to MPI\_GREQUEST\_COMPLETE may unblock a blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (*End of advice to implementors.*)

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\frac{44}{45}$ 

```
13.2.1 Examples
```

**Example 13.1** 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 \rightarrow 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
      to MPI_PROC_NULL if does not exist */
   /* code not shown */
   . . .
  MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
  MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
  MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
                                                                                    a
                                                                                   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 */
   status->MPI_SOURCE = MPI_UNDEFINED;
                                                                                   25
                                                                                   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
ł
   /* 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;
}
```

### 13.3 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 according count (Status)	
28	INCOT	status	status with which to associate count (Status)	
29	IN	datatype	datatype associated with count (handle)	
30	IN	count	number of elements to associate with status (integer)	
31				
32	C binding	<b>o</b> r		
33			Status *status, MPI_Datatype datatype,	
34	IIIC MFI_C	int count)	Status *status, Hri_Datatype datatype,	
35		Int count)		
36	Fortran 2	008 binding		
37	MPI_Statu	s_set_elements(status,	datatype, count, ierror)	
38	TYPE(MPI_Status), INTENT(INOUT) :: status			
39	TYPE(	[MPI_Datatype), INTENT(I]	N) :: datatype	
40	INTEG	ER, INTENT(IN) :: count		
41	INTEG	ER, OPTIONAL, INTENT(OU	T) :: ierror	
42	Fortran b	anding		
43		0		
44	MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR			
45	INIEG	ER SIAIUS (MFI_SIAIUS_SI	ZE), DAIAIIFE, COONI, IERROR	
46				
47				
48				

1

 $\mathbf{2}$ 

3

4 5

> 6 7

8

9

10

11

MPI_STAT	US_SET_ELEMENTS_X(stat	us, datatype, count)	1
INOUT	status	status with which to associate count (Status)	2
IN	datatype	datatype associated with count (handle)	3
IN	count	* <del>-</del>	4 5
IIN	count	number of elements to associate with status (integer)	6
C bindin	σ		7
	•	_Status *status, MPI_Datatype datatype,	8
_	MPI_Count count)		9
Fortran (	2008 binding		10 11
	0	datatype, count, ierror)	11
	(MPI_Status), INTENT(INOU		13
TYPE	(MPI_Datatype), INTENT(IN	) :: datatype	14
	GER(KIND=MPI_COUNT_KIND),		15
INTEC	GER, OPTIONAL, INTENT(OUT	) :: ierror	16
Fortran l	oinding		17
		DATATYPE, COUNT, IERROR)	18 19
	GER STATUS(MPI_STATUS_SIZ		20
INTEC	GER(KIND=MPI_COUNT_KIND)	COUNT	21
	· · · ·	ne part of status so that a call to	22
		EMENTS_X will return count. MPI_GET_COUNT	23
will return	a compatible value.		24
Rati	onale. The number of elem-	ents is set instead of the count because the former	25 26
		er of datatypes. (End of rationale.)	20 27
	0		28
	_	OUNT(status, datatype, count),	29
	ELEMENTS status, datatype,		30
	MPI_GET_ELEMENTS_X(status, datatype, count) must use a datatype argument that has		
	the same type signature as the datatype argument that was used in the call to MPI_STATUS_SET_ELEMENTS or MPI_STATUS_SET_ELEMENTS_X.		
			33 34
Rati	onale. The requirement of	matching type signatures for these calls is similar	35
		a count is set by a receive operation: in that case,	36
		PI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X	37
	t use a datatype with the sam <i>l</i> of rationale.)	e signature as the datatype used in the receive call.	38
( <i>Em</i>	i of fationale.)		39
	*		40
		(lag)	41 42
	TUS_SET_CANCELLED(status	_,	43
INOUT	status	status with which to associate cancel flag (Status)	44
IN	flag	if true, indicates request was cancelled (logical)	45
			46
C bindin	•		47
int MPI_S	Status_set_cancelled(MPI_	Status *status, int flag)	48

Unofficial Draft for Comment Only

1 2 3 4 5	<pre>Fortran 2008 binding MPI_Status_set_cancelled(status, flag, ierror)     TYPE(MPI_Status), INTENT(INOUT) :: status     LOGICAL, INTENT(IN) :: flag     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
6 7 8 9 10	Fortran binding MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG
11 12 13	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.
14 15 16 17 18 19 20	Advice to users. Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI_GET_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable
21 22 23	results and is strongly discouraged. (End of advice to users.)
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

# I/O

### 14.1 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 [54], collective buffering [8, 16, 55, 59, 66], and disk-directed I/O [48]) 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.

### 14.1.1 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 14.1 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).

18	etype
19	
20	filetype
21	holes —
22	tiling a file with the filetype:
23	timig a me with the metype.
24	
25	displacement accessible data
26	
27	Figure 14.1: Etypes and filetypes
28	A meun of processes can use complementary views to achieve a global data distribution
29	A group of processes can use complementary views to achieve a global data distribution
30	such as a scatter/gather pattern (see Figure $14.2$ ).
31	etype
32	process 0 filetype
33	
34	process 1 filetype
35	process 2 filetype
36	tiling a file with the filetomest
37	tiling a file with the filetypes:
38	
39	displacement
40	
41	Figure 14.2: Partitioning a file among parallel processes
42	

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 14.2 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.

#### 14.2 File Manipulation

14.2.1 Opening a File

			10
MPI_FILE_OPEN(comm, filename, amode, info, fh)		20	
IN	comm	communicator (handle)	21
IN	filename	name of file to open (string)	22 23
IN	amode	file access mode (integer)	23
IN	info	info object (handle)	25
		into object (nandic)	26
OUT	fh	new file handle (handle)	27
			28
C binding			29
<pre>int MPI_File_open(MPI_Comm comm, const char *filename, int amode,</pre>		30	
	MPI_Info i	nfo, MPI_File *fh)	31
Fortran	2008 binding		32
	MPI_File_open(comm, filename, amode, info, fh, ierror)		
TYPE(MPI_Comm), INTENT(IN) :: comm		34	
CHARACTER(LEN=*), INTENT(IN) :: filename		35	
INTEGER, INTENT(IN) :: amode		36	
TYPE(MPI_Info), INTENT(IN) :: info		37	
	C(MPI_File), INT		38
			39
	GER, UPIIUNAL,	INTENT(OUT) :: ierror	40

### Fortran binding

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    INTEGER COMM, AMODE, INFO, FH, IERROR
    CHARACTER*(*) FILENAME
```

MPI\_FILE\_OPEN opens the file identified by the file name filename on all processes in the comm communicator group. MPI\_FILE\_OPEN is a collective routine: all processes must provide the same value for amode, and all processes must provide filenames that reference 

 $\mathbf{2}$ 

 $\overline{7}$ 

1 the same file. (Values for info may vary.) comm must be an intra-communicator; it is  $\mathbf{2}$ erroneous to pass an inter-communicator to MPI\_FILE\_OPEN. Errors in MPI\_FILE\_OPEN 3 are raised using the default file error handler (see Section 14.7). When using the World 4 Model (Section 11.1), a process can open a file independently of other processes by using 5the MPI\_COMM\_SELF communicator. Applications using the Sessions Model (Section 11.3) 6 can achieve the same result using communicators created from the "mpi://SELF" process  $\overline{7}$ set. The file handle returned, fh, can be subsequently used to access the file until the file is 8 closed using MPI\_FILE\_CLOSE. Before calling MPI\_FINALIZE, the user is required to close 9 (via MPI\_FILE\_CLOSE) all files that were opened with MPI\_FILE\_OPEN. Note that the 10 communicator comm is unaffected by MPI\_FILE\_OPEN and continues to be usable in all 11MPI routines (e.g., MPI\_SEND). Furthermore, the use of comm will not interfere with I/O 12behavior.

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.*)

27 28

29

30

 $^{31}$ 

32

33

34

35 36

37 38

39

40 41

42 43

44

 $45 \\ 46$ 

47 48

15

16

17

18

19

20 21

22

23

24

25

26

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.

### Unofficial Draft for Comment Only

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.)

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 [44]. 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 14.2.8). 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 14.6.1). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI\_FILE\_SET\_ATOMICITY.

Unofficial Draft for Comment Only

 $^{24}$ 

```
1
     14.2.2 Closing a File
\mathbf{2}
3
4
     MPI_FILE_CLOSE(fh)
5
       INOUT
                 fh
                                              file handle (handle)
6
7
     C binding
8
     int MPI_File_close(MPI_File *fh)
9
10
     Fortran 2008 binding
11
     MPI_File_close(fh, ierror)
12
          TYPE(MPI_File), INTENT(INOUT) :: fh
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     Fortran binding
15
16
     MPI_FILE_CLOSE(FH, IERROR)
17
          INTEGER FH, IERROR
18
          MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
19
     MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
20
     opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
21
     MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.
22
23
           Advice to users. If the file is deleted on close, and there are other processes currently
24
           accessing the file, the status of the file and the behavior of future accesses by these
25
           processes are implementation dependent. (End of advice to users.)
26
27
          The user is responsible for ensuring that all outstanding nonblocking requests and
28
     split collective operations associated with fh made by a process have completed before that
29
     process calls MPI_FILE_CLOSE.
30
          The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
^{31}
     MPI_FILE_NULL.
32
33
     14.2.3 Deleting a File
34
35
36
     MPI_FILE_DELETE(filename, info)
37
       IN
                 filename
                                              name of file to delete (string)
38
39
       IN
                 info
                                              info object (handle)
40
41
     C binding
42
     int MPI_File_delete(const char *filename, MPI_Info info)
43
     Fortran 2008 binding
44
     MPI_File_delete(filename, info, ierror)
45
46
          CHARACTER(LEN=*), INTENT(IN) :: filename
47
          TYPE(MPI_Info), INTENT(IN) :: info
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

#### Fortran binding

```
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
CHARACTER*(*) FILENAME
INTEGER INFO, IERROR
```

MPI\_FILE\_DELETE deletes the file identified by the file name filename. If the file does not exist, MPI\_FILE\_DELETE raises an error in the class MPI\_ERR\_NO\_SUCH\_FILE.

The info argument can be used to provide information regarding file system specifics (see Section 14.2.8). The constant MPI\_INFO\_NULL refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class MPI\_ERR\_FILE\_IN\_USE or MPI\_ERR\_ACCESS will be raised. Errors are raised using the default file error handler (see Section 14.7).

### 14.2.4 Resizing a File

MPI_FILE	_SET_SIZE(fh, siz	e)	
INOUT	fh	file handle (handle)	
IN	size	size to truncate or expand file (integer)	
C binding int MPI_F	0	PI_File fh, MPI_Offset size)	
MPI_File_	2008 binding _set_size(fh, s (MPI_File), INT		

### Fortran binding

MPI_FILE_SET_SIZE(FH,	SIZE, IERRON	R)
INTEGER FH, IERRO	R	
INTEGER(KIND=MPI_	OFFSET_KIND)	SIZE

INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: size

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

### Unofficial Draft for Comment Only

2

3

 $\frac{4}{5}$ 

6

7

8

9

10

11

12

13

14

15 16

30

31

32

38

39

40

41

42

43

44

45

46

47

1 pointer. If MPI\_MODE\_SEQUENTIAL mode was specified when the file was opened, it is  $\mathbf{2}$ erroneous to call this routine. 3 Advice to users. It is possible for the file pointers to point beyond the end of file 4 after a MPI\_FILE\_SET\_SIZE operation truncates a file. This is valid, and equivalent 5to seeking beyond the current end of file. (End of advice to users.) 6 7 All nonblocking requests and split collective operations on fh must be completed before 8 calling MPI\_FILE\_SET\_SIZE. Otherwise, calling MPI\_FILE\_SET\_SIZE is erroneous. As far 9 as consistency semantics are concerned, MPI\_FILE\_SET\_SIZE is a write operation that 10 conflicts with operations that access bytes at displacements between the old and new file 11 sizes (see Section 14.6.1). 121314.2.5 Preallocating Space for a File 14151617MPI\_FILE\_PREALLOCATE(fh, size) 18 file handle (handle) INOUT fh 19 IN size to preallocate file (integer) size 202122 C binding 23int MPI\_File\_preallocate(MPI\_File fh, MPI\_Offset size)  $^{24}$ Fortran 2008 binding 25MPI\_File\_preallocate(fh, size, ierror) 26TYPE(MPI\_File), INTENT(IN) :: fh 27INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: size 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2930 Fortran binding  $^{31}$ MPI\_FILE\_PREALLOCATE(FH, SIZE, IERROR) 32 INTEGER FH, IERROR 33 INTEGER(KIND=MPI\_OFFSET\_KIND) SIZE 34MPI\_FILE\_PREALLOCATE ensures that storage space is allocated for the first size bytes 35 of the file associated with fh. MPI\_FILE\_PREALLOCATE is collective; all processes in the 36 group must pass identical values for size. Regions of the file that have previously been 37 written are unaffected. For newly allocated regions of the file, MPI\_FILE\_PREALLOCATE 38 has the same effect as writing undefined data. If size is larger than the current file size, the 39 file size increases to size. If size is less than or equal to the current file size, the file size is 40 unchanged.

unchanged.
 The treatment of file pointers, pending nonblocking accesses, and file consistency is the
 same as with MPI\_FILE\_SET\_SIZE. If MPI\_MODE\_SEQUENTIAL mode was specified when
 the file was opened, it is erroneous to call this routine.

Advice to users. In some implementations, file preallocation may be time-consuming. (End of advice to users.)

47 48

45

14.2.6 Querying the Size of a File	1
	2
	3
MPI_FILE_GET_SIZE(fh, size)	4
IN fh file handle (handle)	5
	6
OUTsizesize of the file in bytes (integer)	7
	8 9
C binding	10
<pre>int MPI_File_get_size(MPI_File fh, MPI_Offset *size)</pre>	11
Fortran 2008 binding	12
MPI_File_get_size(fh, size, ierror)	13
TYPE(MPI_File), INTENT(IN) :: fh	14
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
Fortran binding	17
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)	18
INTEGER FH, IERROR	19 20
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	20 21
MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with	22
the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a	23
data access operation (see Section 14.6.1).	24
	25
14.2.7 Querying File Parameters	26
	27
	28
MPI_FILE_GET_GROUP(fh, group)	29
IN fh file handle (handle)	30
OUT group group which opened the file (handle)	31 32
group which opened the file (number)	33
C binding	34
int MPI_File_get_group(MPI_File fh, MPI_Group *group)	35
	36
Fortran 2008 binding	37
<pre>MPI_File_get_group(fh, group, ierror)     TYPE(MPI_File), INTENT(IN) :: fh</pre>	38
TYPE(MPI_FILE), INTENT(IN) :: In TYPE(MPI_Group), INTENT(OUT) :: group	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
	41
Fortran binding	42
MPI_FILE_GET_GROUP(FH, GROUP, IERROR)	43 44
INTEGER FH, GROUP, IERROR	44 45
$MPI\_FILE\_GET\_GROUP$ returns a duplicate of the group of the communicator used to	40
open the file associated with fh. The group is returned in group. The user is responsible for	47

freeing group.

```
1
     MPI_FILE_GET_AMODE(fh, amode)
2
       IN
                fh
                                            file handle (handle)
3
       OUT
                amode
                                            file access mode used to open the file (integer)
4
5
6
     C binding
7
     int MPI_File_get_amode(MPI_File fh, int *amode)
8
     Fortran 2008 binding
9
     MPI_File_get_amode(fh, amode, ierror)
10
         TYPE(MPI_File), INTENT(IN) :: fh
11
         INTEGER, INTENT(OUT) :: amode
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
     MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
15
16
         INTEGER FH, AMODE, IERROR
17
         MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with
18
     fh.
19
20
     Example 14.1 In Fortran 77, decoding an amode bit vector will require a routine such as
21
     the following:
22
23
     SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
^{24}
     !
25
     !
         TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
26
         IF SET, RETURN 1 IN BIT_FOUND, O OTHERWISE
     !
27
     i.
28
         INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
29
         BIT_FOUND = 0
30
         CP_AMODE = AMODE
^{31}
     100 CONTINUE
32
         LBIT = 0
33
         HIFOUND = 0
34
         DO L = MAX_BIT, 0, -1
35
             MATCHER = 2**L
36
             IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
37
                 HIFOUND = 1
38
                LBIT = MATCHER
39
                CP_AMODE = CP_AMODE - MATCHER
40
             END IF
41
         END DO
42
         IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
43
         IF (BIT_FOUND .EQ. O .AND. HIFOUND .EQ. 1 .AND. &
44
              CP_AMODE .GT. 0) GO TO 100
45
     END
46
47
         This routine could be called successively to decode amode, one bit at a time. For
```

```
<sup>48</sup> example, the following code fragment would check for MPI_MODE_RDONLY.
```

```
CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
IF (BIT_FOUND .EQ. 1) THEN
PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
ELSE
PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
END IF
```

### 14.2.8 File Info

Hints specified via info (see Chapter 10) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. 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\_FILE\_GET\_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per file basis, in MPI\_FILE\_OPEN, MPI\_FILE\_DELETE, MPI\_FILE\_SET\_VIEW, and MPI\_FILE\_SET\_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI\_FILE\_SET\_VIEW or MPI\_FILE\_SET\_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

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

		29	
MPI_FILE_SET_INFO(fh, info)			
, ,		31	
INOUT fh	file handle (handle)	32	
IN info	info object (handle)	33	
		34	
C binding		35	
<pre>int MPI_File_set_info(MPI_File fh, MPI_Info info)</pre>			
		37	
Fortran 2008 binding			
MPI_File_set_info(fh, info, ierror)			
TYPE(MPI_File), INTENT(IN) :: fh			
TYPE(MPI_Info), INTENT(IN) :: info			
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	42	
Fortran binding			
MPI_FILE_SET_INFO(FH, INFO, IERROR)			
INTEGER FH, INFO, IERROR			
		46	
	hints of the file essentiated with the using the hints		

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

 $^{24}$ 

1 specified by info. It also has no effect on previously set or defaulted hints that are specified  $\mathbf{2}$ by info, but are ignored by the MPI implementation in this call to MPI\_FILE\_SET\_INFO. 3 MPI\_FILE\_SET\_INFO is a collective routine. The info object may be different on each 4 process, but any info entries that an implementation requires to be the same on all processes 5must appear with the same value in each process's info object. 6

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.)

file handle (handle)

new info object (handle)

16

 $\overline{7}$ 

8

9

10

11

12

13 1415

19

2021

22

27

28

MPI\_FILE\_GET\_INFO(fh, info\_used) 17IN 18

OUT

C binding int MPI\_File\_get\_info(MPI\_File fh, MPI\_Info \*info\_used)

```
23
      Fortran 2008 binding
^{24}
```

fh

MPI\_File\_get\_info(fh, info\_used, ierror) 2526

info\_used

```
TYPE(MPI_File), INTENT(IN) :: fh
```

TYPE(MPI\_Info), INTENT(OUT) :: info\_used INTEGER, OPTIONAL, INTENT(OUT) :: ierror

29Fortran binding

```
30
     MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
^{31}
         INTEGER FH, INFO_USED, IERROR
32
```

33 MPI\_FILE\_GET\_INFO returns a new info object containing the hints of the file associ-34ated with fh. The current setting of all hints related to this file is returned in info\_used. An MPI implementation is required to return all hints that are supported by the implementa-35 36 tion and have default values specified; any user-supplied hints that were not ignored by the 37 implementation; and any additional hints that were set by the implementation. If no such 38hints exist, a handle to a newly created info object is returned that contains no (key, value) 39 pairs. The user is responsible for freeing info\_used via MPI\_INFO\_FREE.

- 40
- $^{41}$ Reserved File Hints

42Some potentially useful hints (info key values) are outlined below. The following key values 43 are reserved. An implementation is not required to interpret these key values, but if it does 44interpret the key value, it must provide the functionality described. (For more details on 45"info," see Chapter 10.) 46

These hints mainly affect access patterns and the layout of data on parallel I/O devices. 47For each hint name introduced, we describe the purpose of the hint, and the type of the hint 48

value. The "[**SAME**]" annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., "file\_perm" is only useful during file creation).

- "access\_style" (comma separated list of strings): This hint specifies the manner in which the file will be accessed until the file is closed or until the "access\_style" key value is altered. The hint value is a comma separated list of the following: "read\_once", "write\_once", "read\_mostly", "write\_mostly", "sequential", "reverse\_sequential", and "random".
- "collective\_buffering" (boolean) [SAME]: This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Valid values for this key are "true" and "false". Collective buffering parameters are further directed via additional hints: "cb\_block\_size", "cb\_buffer\_size", and "cb\_nodes".
- "cb\_block\_size" (integer) [SAME]: This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (cyclic) pattern.
- "cb\_buffer\_size" (integer) [SAME]: This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of "cb\_block\_size".
- "cb\_nodes" (integer) [SAME]: This hint specifies the number of target nodes to be used for collective buffering.
- "chunked" (comma separated list of integers) [SAME]: This hint specifies that the file consists of a multidimentional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).
- "chunked\_item" (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.
- "chunked\_size" (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.
- "filename" (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI\_FILE\_GET\_INFO. This key is ignored when passed to MPI\_FILE\_OPEN, MPI\_FILE\_SET\_VIEW, MPI\_FILE\_SET\_INFO, and MPI\_FILE\_DELETE.
- "file\_perm" (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI\_FILE\_OPEN with an amode

### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{31}$ 

1 2		that includes MPI_MODE_CREATE. The set of valid values for this key is implementation dependent.		
3 4 5 6	"io_node_list" (comma separated list of strings) [SAME]: This hint specifies the of I/O devices that should be used to store the file. This hint is most relevant we the file is created.			
7 8 9 10	"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.			
11 12	"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.			
13 14 15		· · · -	<b>ME</b> ]: This hint specifies the number of I/O devices that cross, and is relevant only when the file is created.	
16 17 18 19 20	used I/O o	for this file. The strip device before progress	<b>E</b> ]: This hint specifies the suggested striping unit to be ing unit is the amount of consecutive data assigned to one ing to the next device, when striping across a number of bytes. This hint is relevant only when the file is created.	
21 22 23 24	14.3 Fi	le Views		
25	MPI_FILE_	_SET_VIEW(fh, disp, e	type, filetype, datarep, info)	
26 27	INOUT	fh	file handle (handle)	
28	IN	disp	displacement (integer)	
29	IN	etype	elementary datatype (handle)	
30 31	IN	filetype	filetype (handle)	
32	IN	datarep	data representation (string)	
33 34	IN	info	info object (handle)	
35 36 37 38	C binding int MPI_F	ile_set_view(MPI_F	ile fh, MPI_Offset disp, MPI_Datatype etype, Tiletype, const char *datarep, MPI_Info info)	
39 40		008 binding set_view(fh, disp,	etype, filetype, datarep, info, ierror)	
41 42 43	INTEG		_KIND), INTENT(IN) :: disp	
44	TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype CHARACTER(LEN=*), INTENT(IN) :: datarep			
45	TYPE(MPI_Info), INTENT(IN) :: info			
46	INTEG	ER, OPTIONAL, INTE	NT(OUT) :: ierror	
47 48	Fortran b	inding		

# MPI\_FILE\_SET\_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER(KIND=MPI\_OFFSET\_KIND) DISP CHARACTER\*(\*) DATAREP

The MPI\_FILE\_SET\_VIEW routine changes the process's view of the data in the file. The start of the view is set to disp; the type of data is set to etype; the distribution of data to processes is set to filetype; and the representation of data in the file is set to datarep. In addition, MPI\_FILE\_SET\_VIEW resets the individual file pointers and the shared file pointer to zero. MPI\_FILE\_SET\_VIEW is collective; the values for datarep and the extents of etype in the file data representation must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be committed.

The etype always specifies the data layout in the file. If etype is a portable datatype (see Section 2.4), the extent of etype is computed by scaling any displacements in the datatype to match the file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section 14.5.1 for further details.

If MPI\_MODE\_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI\_DISPLACEMENT\_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer. MPI\_DISPLACEMENT\_CURRENT is invalid unless the amode for the file has MPI\_MODE\_SEQUENTIAL set.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. MPI\_DISPLACEMENT\_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to MPI\_FILE\_SET\_VIEW will immediately follow MPI\_FILE\_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 14.3). Separate views, each using a different displacement and filetype, can be used to access each segment.



Figure 14.3: Displacements

(End of advice to users.)

# Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 

 $45 \\ 46$ 

1 An etype (elementary datatype) is the unit of data access and positioning. It can be  $\mathbf{2}$ any MPI predefined or derived datatype. Derived etypes can be constructed by using any 3 of the MPI datatype constructor routines, provided all resulting typemap displacements are 4 non-negative and monotonically nondecreasing. Data access is performed in etype units,  $\mathbf{5}$ reading or writing whole data items of type etype. Offsets are expressed as a count of 6 etypes; file pointers point to the beginning of etypes.

> In order to ensure interoperability in a heterogeneous environ-Advice to users. ment, additional restrictions must be observed when constructing the etype (see Section 14.5). (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 16contain overlapping regions. This restriction is equivalent to the "datatype used in a receive 17cannot specify overlapping regions" restriction for communication. Note that filetypes from 18 different processes may still overlap each other. 19

If a filetype has holes in it, then the data in the holes is inaccessible to the calling 20process. However, the disp, etype, and filetype arguments can be changed via future calls to 21MPI\_FILE\_SET\_VIEW to access a different part of the file. 22

It is erroneous to use absolute addresses in the construction of the etype and filetype. 23The info argument is used to provide information regarding file access patterns and file 24system specifics to direct optimization (see Section 14.2.8). The constant MPI\_INFO\_NULL 25refers to the null info and can be used when no info needs to be specified. 26

The datarep argument is a string that specifies the representation of data in the file. 27See the file interoperability section (Section 14.5) for details and a discussion of valid values. 28

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI\_FILE\_SET\_VIEW—otherwise, the call to MPI\_FILE\_SET\_VIEW is erroneous.

 $^{31}$ 32 33

34

29

30

MPI\_FILE\_GET\_VIEW(fh, disp, etype, filetype, datarep)

35	1N	fh	file handle (handle)
36	OUT	disp	displacement (integer)
37 38	OUT	etype	elementary datatype (handle)
39	OUT	filetype	filetype (handle)
40	OUT	datarep	data representation (string)
41			

C binding

```
43
     int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
44
                   MPI_Datatype *filetype, char *datarep)
45
```

```
Fortran 2008 binding
46
```

```
47
     MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
         TYPE(MPI_File), INTENT(IN) :: fh
48
```

**Unofficial Draft for Comment Only** 

7

8

9

10 11

12

13

14

```
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
CHARACTER(LEN=*), INTENT(OUT) :: datarep
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

# Fortran binding

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
INTEGER FH, ETYPE, FILETYPE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) DISP
CHARACTER*(*) DATAREP
```

MPI\_FILE\_GET\_VIEW returns the process's view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI\_MAX\_DATAREP\_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI\_FILE\_GET\_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

# 14.4 Data Access

## 14.4.1 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 14.1.

positioning	synchronism	coo	ordination
		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
	, i i i i i i i i i i i i i i i i i i i	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 14.1: Data access routines

# Unofficial Draft for Comment Only

 $\mathbf{2}$ 

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.

<sup>9</sup> Positioning 10

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 14.4.2.

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 14.4.3. The data access routines that use shared file pointers contain \_SHARED or \_ORDERED in their name (e.g., MPI\_FILE\_WRITE\_SHARED). Operations with shared file pointers are described in Section 14.4.4.

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,

29 30 31

32

37

 $new_file_offset = old_file_offset + \frac{elements(datatype)}{elements(etupe)} \times count$ 

where *count* is the number of *datatype* items to be accessed, elements(X) is the number of predefined datatypes in the typemap of X, and *old\_file\_offset* is the value of the implicit offset before the call. The file position, *new\_file\_offset*, is in terms of a count of etypes relative to the current view.

<sup>38</sup> Synchronism

 $_{40}^{39}$  MPI supports blocking and nonblocking I/O routines.

A blocking I/O call will not return until the I/O request is completed.

A nonblocking I/O call with not return tilter the I/O request is completed. A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete. Given suitable hardware, this allows the transfer of data out of and 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.

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18 19

20 21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

42

43 44

45

46

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 14.4.5).

#### 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 14.6.4 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.

#### 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 5.1.11. The data is accessed from those parts of the file specified by the current view (Section 14.3). 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 19.1.10–19.1.20. (End of advice to users.)

# $47 \\ 48$

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

<sup>5</sup> MPI\_GET\_ELEMENTS\_X), respectively. The interpretation of the MPI\_ERROR field is the <sup>6</sup> same as for other operations—normally undefined, but meaningful if an MPI routine returns <sup>7</sup> MPI\_ERR\_IN\_STATUS. The user can pass (in C and Fortran) MPI\_STATUS\_IGNORE in the <sup>8</sup> status argument if the return value of this argument is not needed. The status can be <sup>9</sup> passed to MPI\_TEST\_CANCELLED to determine if the operation was cancelled. All other <sup>10</sup> fields of status are undefined.

<sup>11</sup> When reading, a program can detect the end of file by noting that the amount of data <sup>12</sup> read is less than the amount requested. Writing past the end of file increases the file size. <sup>13</sup> The amount of data accessed will be the amount requested, unless an error is raised (or a <sup>14</sup> read reaches the end of file).

<sup>16</sup> 14.4.2 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.

MPI\_FILE\_READ\_AT(fh, offset, buf, count, datatype, status)

```
22
                 fh
       IN
                                            file handle (handle)
23
24
       IN
                offset
                                            file offset (integer)
25
       OUT
                 buf
                                            initial address of buffer (choice)
26
       IN
                 count
                                            number of elements in buffer (integer)
27
       IN
                                            datatype of each buffer element (handle)
                 datatype
28
29
       OUT
                status
                                            status object (Status)
30
^{31}
     C binding
32
     int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
33
                    MPI_Datatype datatype, MPI_Status *status)
34
35
     int MPI_File_read_at_c(MPI_File fh, MPI_Offset offset, void *buf,
36
                    MPI_Count count, MPI_Datatype datatype, MPI_Status *status)
37
     Fortran 2008 binding
38
     MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
39
          TYPE(MPI_File), INTENT(IN) :: fh
40
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
41
          TYPE(*), DIMENSION(..) :: buf
42
          INTEGER, INTENT(IN) :: count
43
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
          TYPE(MPI_Status) :: status
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
48
          TYPE(MPI_File), INTENT(IN) :: fh
```

15

		FFSET_KIND), INTENT(IN) :: offset	1
	(*), DIMENSION		2 3
		DUNT_KIND), INTENT(IN) :: count , INTENT(IN) :: datatype	4
	(MPI_Datatype) (MPI_Status) :		5
		INTENT(OUT) :: ierror	6
			7
Fortran			8
		FFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	9
		FSET_KIND) OFFSET	10
	e> BUF(*)		11 12
• -		useds a file having ing at the position specified by effect	12
IVIPI_	FILE_READ_AL	reads a file beginning at the position specified by offset.	14
			15
MPI_FILE	E_READ_AT_ALL(	fh, offset, buf, count, datatype, status)	16
IN	fh	file handle (handle)	17
IN	offset	file offset (integer)	18
OUT	buf	initial address of buffer (choice)	19 20
IN	count	number of elements in buffer (integer)	20
			22
IN	datatype	datatype of each buffer element (handle)	23
OUT	status	status object (Status)	24
<b>~</b>			25
C bindir	0	11 (MDT Dile from MDT Offert offert unit thuf	26 27
int MP1_		ll(MPI_File fh, MPI_Offset offset, void *buf, , MPI_Datatype datatype, MPI_Status *status)	28
			29
int MPI_		<pre>Ll_c(MPI_File fh, MPI_Offset offset, void *buf,</pre>	30
	MP1_Count	<pre>count, MPI_Datatype datatype, MPI_Status *status)</pre>	31
	2008 binding		32
		n, offset, buf, count, datatype, status, ierror)	33
	C(MPI_File), IN		34 35
	GER(KIND=MP1_U (*), DIMENSION	FFSET_KIND), INTENT(IN) :: offset	36
	GER, INTENT(IN)		37
		, INTENT(IN) :: datatype	38
	C(MPI_Status) :	• •	39
INTE	GER, OPTIONAL,	INTENT(OUT) :: ierror	40
MPT File	read at all(fl	n, offset, buf, count, datatype, status, ierror)	41
	C(MPI_File), IN		42 43
INTE	GER(KIND=MPI_O	FFSET_KIND), INTENT(IN) :: offset	43
	C(*), DIMENSION		45
		DUNT_KIND), INTENT(IN) :: count	46
		, INTENT(IN) :: datatype	47
IYPE	C(MPI_Status) :	: StatuS	48

```
652
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     Fortran binding
3
     MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
4
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
5
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
6
         <type> BUF(*)
7
8
         MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT
9
     interface.
10
11
     MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)
12
13
       INOUT
                fh
                                           file handle (handle)
14
       IN
                offset
                                           file offset (integer)
15
       IN
                buf
                                           initial address of buffer (choice)
16
17
       IN
                count
                                           number of elements in buffer (integer)
18
       IN
                                           datatype of each buffer element (handle)
                datatype
19
       OUT
                status
                                           status object (Status)
20
21
     C binding
22
     int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
23
^{24}
                    int count, MPI_Datatype datatype, MPI_Status *status)
25
     int MPI_File_write_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
26
                    MPI_Count count, MPI_Datatype datatype, MPI_Status *status)
27
28
     Fortran 2008 binding
     MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
29
30
         TYPE(MPI_File), INTENT(IN) :: fh
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
31
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
32
33
         INTEGER, INTENT(IN) :: count
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Status) :: status
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
40
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         TYPE(MPI_Status) :: status
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
48
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

	EGER(KIND=MPI_OFFSET_KIND pe> BUF(*)	) OFFSET	1 2
MP	$I_FILE_WRITE_AT$ writes a file	e beginning at the position specified by offset.	3 4 5
MPI_FIL	E_WRITE_AT_ALL(fh, offset,	buf, count, datatype, status)	6
INOUT		file handle (handle)	7
IN	offset	file offset (integer)	8 9
IN	buf	initial address of buffer (choice)	10
IN	count	number of elements in buffer (integer)	11
			12
	datatype	datatype of each buffer element (handle)	13 14
OUT	status	status object (Status)	15
C bindi	'nσ		16
	5	le fh, MPI_Offset offset, const void *buf,	17
	int count, MPI_Data	type datatype, MPI_Status *status)	18 19
int MPI	_File_write_at_all_c(MPI_	File fh, MPI_Offset offset,	20
		PI_Count count, MPI_Datatype datatype,	21
	MPI_Status *status)		22
Fortran	2008 binding		23 24
		, buf, count, datatype, status, ierror)	24 25
	E(MPI_File), INTENT(IN) :		26
	EGER(KIND=MPI_OFFSET_KIND E(*), DIMENSION(), INTE		27
	EGER, INTENT(IN) :: count		28
	E(MPI_Datatype), INTENT(I	N) :: datatype	29 30
	E(MPI_Status) :: status		31
INT	EGER, OPTIONAL, INTENT(OU	T) :: ierror	32
		, buf, count, datatype, status, ierror)	33
	E(MPI_File), INTENT(IN) :		34
	EGER(KIND=MPI_OFFSET_KIND E(*), DIMENSION(), INTE	• • •	35 36
	EGER(KIND=MPI_COUNT_KIND)		37
	E(MPI_Datatype), INTENT(I	• • • •	38
	E(MPI_Status) :: status		39
INT	EGER, OPTIONAL, INTENT(OU	T) :: ierror	40 41
Fortran	ı binding		42
		, BUF, COUNT, DATATYPE, STATUS, IERROR)	43
	EGER FH, COUNT, DATATYPE, EGER(KIND=MPI_OFFSET_KIND	STATUS(MPI_STATUS_SIZE), IERROR ) OFFSET	44
	pe> BUF(*)		45
·	-	collective version of the blocking	46 47
	E_WRITE_AT_ALL is a of E_WRITE_AT interface.	concentre version of the blocking	48

```
1
     MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)
2
       IN
                fh
                                           file handle (handle)
3
                offset
       IN
                                           file offset (integer)
4
5
       OUT
                buf
                                           initial address of buffer (choice)
6
       IN
                count
                                           number of elements in buffer (integer)
7
       IN
                datatype
                                           datatype of each buffer element (handle)
8
9
       OUT
                request
                                           request object (handle)
10
11
     C binding
12
     int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
13
                    MPI_Datatype datatype, MPI_Request *request)
14
     int MPI_File_iread_at_c(MPI_File fh, MPI_Offset offset, void *buf,
15
                    MPI_Count count, MPI_Datatype datatype, MPI_Request *request)
16
17
     Fortran 2008 binding
18
     MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
19
         TYPE(MPI_File), INTENT(IN) :: fh
20
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
22
         INTEGER, INTENT(IN) :: count
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
27
         TYPE(MPI_File), INTENT(IN) :: fh
28
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
29
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
30
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Request), INTENT(OUT) :: request
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     Fortran binding
36
     MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
37
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
38
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
39
         <type> BUF(*)
40
         MPI_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.
41
42
43
44
45
46
47
48
```

MPI_FILE	_IREAD_AT_ALL(fh, offset, bu	f, count, datatype, request)	1
IN	fh	file handle (handle)	2 3
IN	offset	file offset (integer)	4
OUT	buf	initial address of buffer (choice)	5
IN	count	number of elements in buffer (integer)	6 7
IN	datatype	datatype of each buffer element (handle)	8
OUT	request	request object (handle)	9
			10 11
C binding			11
int MPI_F		e fh, MPI_Offset offset, void *buf, ype datatype, MPI_Request *request)	13
int MDT I		ile fh, MPI_Offset offset, void *buf,	14
IIIC MFI_f		_Datatype datatype, MPI_Request *request)	15 16
Fortran 2	2008 binding		17
	0	buf, count, datatype, request, ierror)	18
	<pre>MPI_File), INTENT(IN) ::</pre>		19 20
	<pre>ER(KIND=MPI_OFFSET_KIND)   (*), DIMENSION(), ASYNCH</pre>		21
	ER, INTENT(IN) :: count	IKUNUUS :: Dul	22
TYPE	MPI_Datatype), INTENT(IN)		23 24
	(MPI_Request), INTENT(OUT)		24 25
INTEC	ER, OPTIONAL, INTENT(OUT)	:: ierror	26
		buf, count, datatype, request, ierror)	27
	<pre>[MPI_File), INTENT(IN) :: ER(KIND=MPI_OFFSET_KIND)</pre>		28 29
	(*), DIMENSION(), ASYNCH		30
	<pre>ER(KIND=MPI_COUNT_KIND),</pre>		31
	(MPI_Datatype), INTENT(IN)	• •	32
	MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)	-	$33 \\ 34$
Fortran k			35
	<u>j</u>	BUF, COUNT, DATATYPE, REQUEST, IERROR)	36
	ER FH, COUNT, DATATYPE, F		37 38
	<pre>ER(KIND=MPI_OFFSET_KIND)</pre>	OFFSET	39
<type< td=""><td>&gt; BUF(*)</td><td></td><td>40</td></type<>	> BUF(*)		40
		blocking version of MPI_FILE_READ_AT_ALL. See	41
Section 14	.6.5 for semantics of nonblock	ing collective file operations.	42 43
			44
			45
			46
			47

```
1
     MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
2
       INOUT
                fh
                                           file handle (handle)
3
                offset
       IN
                                           file offset (integer)
4
5
                buf
                                           initial address of buffer (choice)
       IN
6
       IN
                count
                                           number of elements in buffer (integer)
7
       IN
                datatype
                                           datatype of each buffer element (handle)
8
9
       OUT
                request
                                           request object (handle)
10
11
     C binding
12
     int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
13
                    int count, MPI_Datatype datatype, MPI_Request *request)
14
     int MPI_File_iwrite_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
15
                    MPI_Count count, MPI_Datatype datatype, MPI_Request *request)
16
17
     Fortran 2008 binding
18
     MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
19
         TYPE(MPI_File), INTENT(IN) :: fh
20
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
21
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
22
         INTEGER, INTENT(IN) :: count
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
27
         TYPE(MPI_File), INTENT(IN) :: fh
28
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
30
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Request), INTENT(OUT) :: request
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     Fortran binding
36
     MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
37
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
38
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
39
         <type> BUF(*)
40
         MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.
41
42
43
44
45
46
47
48
```

MPI_FILE	_IWRITE_AT_ALL(fh, offset, b	uf, count, datatype, request)	1
INOUT	fh	file handle (handle)	2 3
IN	offset	file offset (integer)	4
IN	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 8
OUT	request	request object (handle)	9
			10
C bindin	g		11
int MPI_		<pre>le fh, MPI_Offset offset, const void *buf,</pre>	12 13
	int count, MP1_Datat	ype datatype, MPI_Request *request)	14
int MPI_		File fh, MPI_Offset offset,	15
	const void *buf, MPI MPI_Request *request	_Count count, MPI_Datatype datatype,	16
		,	17 18
	2008 binding	, buf, count, datatype, request, ierror)	19
	(MPI_File), INTENT(IN) ::		20
	GER(KIND=MPI_OFFSET_KIND)		21 22
		I(IN), ASYNCHRONOUS :: buf	22
	GER, INTENT(IN) :: count (MPI_Datatype), INTENT(IN)	) ··· datatuna	24
	(MPI_Batatype), INTENT(IN) (MPI_Request), INTENT(OUT)		25
	GER, OPTIONAL, INTENT(OUT)	-	26
MPI File	iwrite at all(fh. offset	, buf, count, datatype, request, ierror)	27 28
	(MPI_File), INTENT(IN) ::		29
	GER(KIND=MPI_OFFSET_KIND)		30
		r(IN), ASYNCHRONOUS :: buf	31
	GER(KIND=MPI_COUNT_KIND), (MPI_Datatype), INTENT(IN)		32 33
	(MPI_Request), INTENT(OUT)	V1	34
INTE	GER, OPTIONAL, INTENT(OUT)	) :: ierror	35
Fortran	binding		36
MPI_FILE	_IWRITE_AT_ALL(FH, OFFSET	, BUF, COUNT, DATATYPE, REQUEST, IERROR)	37 38
	GER FH, COUNT, DATATYPE, I		39
	GER(KIND=MPI_OFFSET_KIND) e> BUF(*)	OFFSEI	40
			41
MPI_	FILE_IVVRITE_AT_ALL is a no	nblocking version of MPI_FILE_WRITE_AT_ALL.	42 43
14.4.3 C	ata Access with Individual Fi	le Pointers	44
			45
	-	er per process per file handle. The current value offset in the data access routines described in this	46
po			47 48

```
1
     section. These routines only use and update the individual file pointers maintained by MPI.
\mathbf{2}
     The shared file pointer is not used nor updated.
3
          The individual file pointer routines have the same semantics as the data access with
4
     explicit offset routines described in Section 14.4.2, with the following modification:
5
        • the offset is defined to be the current value of the MPI-maintained individual file
6
           pointer.
7
8
     After an individual file pointer operation is initiated, the individual file pointer is updated
9
     to point to the next etype after the last one that will be accessed. The file pointer is updated
10
     relative to the current view of the file.
11
         If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
12
     to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.
13
14
15
     MPI_FILE_READ(fh, buf, count, datatype, status)
16
       INOUT
                 fh
                                              file handle (handle)
17
       OUT
                 buf
                                              initial address of buffer (choice)
18
19
                                              number of elements in buffer (integer)
       IN
                 count
20
       IN
                                              datatype of each buffer element (handle)
                 datatype
21
       OUT
                 status
                                              status object (Status)
22
23
^{24}
     C binding
     int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
25
26
                    MPI_Status *status)
27
     int MPI_File_read_c(MPI_File fh, void *buf, MPI_Count count,
28
                    MPI_Datatype datatype, MPI_Status *status)
29
30
     Fortran 2008 binding
     MPI_File_read(fh, buf, count, datatype, status, ierror)
31
32
          TYPE(MPI_File), INTENT(IN) :: fh
33
          TYPE(*), DIMENSION(..) :: buf
34
          INTEGER, INTENT(IN) :: count
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
36
          TYPE(MPI_Status) :: status
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_File_read(fh, buf, count, datatype, status, ierror)
39
          TYPE(MPI_File), INTENT(IN) :: fh
40
          TYPE(*), DIMENSION(..) :: buf
41
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
          TYPE(MPI_Status) :: status
44
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
48
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
1
    <type> BUF(*)
                                                                                          \mathbf{2}
    MPI_FILE_READ reads a file using the individual file pointer.
                                                                                          3
                                                                                         4
Example 14.2 The following Fortran code fragment is an example of reading a file until
                                                                                          5
the end of file is reached:
                                                                                          6
Ţ
    Read a preexisting input file until all data has been read.
                                                                                          7
    Call routine "process_input" if all requested data is read.
!
!
    The Fortran 90 "exit" statement exits the loop.
                                                                                          9
                                                                                         10
           bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
                                                                                         11
integer
parameter (bufsize=100)
                                                                                         12
           localbuffer(bufsize)
                                                                                         13
real
                                                                                         14
integer (kind=MPI_OFFSET_KIND) zero
                                                                                         15
                                                                                         16
zero = 0
                                                                                         17
                                                                                         18
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
                     MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
                                                                                         19
call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
                                                                                         20
                         MPI_INFO_NULL, ierr)
                                                                                         21
totprocessed = 0
                                                                                         22
                                                                                         23
do
   call MPI_FILE_READ(myfh, localbuffer, bufsize, MPI_REAL, &
                                                                                         ^{24}
                                                                                         25
                        status, ierr)
                                                                                         26
   call MPI_GET_COUNT(status, MPI_REAL, numread, ierr)
   call process_input(localbuffer, numread)
                                                                                         27
   totprocessed = totprocessed + numread
                                                                                         28
   if (numread < bufsize) exit
                                                                                         29
end do
                                                                                         30
                                                                                         31
write(6, 1001) numread, bufsize, totprocessed
                                                                                         32
                                                                                         33
1001 format("No more data: read", I3, "and expected", I3, &
              "Processed total of", I6, "before terminating job.")
                                                                                         34
                                                                                         35
call MPI_FILE_CLOSE(myfh, ierr)
                                                                                         36
                                                                                         37
                                                                                         38
                                                                                         39
MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
                                                                                         40
 INOUT
           fh
                                       file handle (handle)
                                                                                         41
 OUT
                                                                                         42
           buf
                                       initial address of buffer (choice)
                                                                                         43
 IN
           count
                                       number of elements in buffer (integer)
                                                                                         44
           datatype
 IN
                                       datatype of each buffer element (handle)
                                                                                         45
 OUT
                                       status object (Status)
           status
                                                                                         46
                                                                                         47
                                                                                         48
```

```
C binding
```

```
int MPI_File_read_all(MPI_File fh, void *buf, int count,
\mathbf{2}
                    MPI_Datatype datatype, MPI_Status *status)
3
     int MPI_File_read_all_c(MPI_File fh, void *buf, MPI_Count count,
4
                    MPI_Datatype datatype, MPI_Status *status)
5
6
     Fortran 2008 binding
\overline{7}
     MPI_File_read_all(fh, buf, count, datatype, status, ierror)
8
         TYPE(MPI_File), INTENT(IN) :: fh
9
         TYPE(*), DIMENSION(..) :: buf
10
         INTEGER, INTENT(IN) :: count
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         TYPE(MPI_Status) :: status
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_File_read_all(fh, buf, count, datatype, status, ierror)
15
         TYPE(MPI_File), INTENT(IN) :: fh
16
         TYPE(*), DIMENSION(..) :: buf
17
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
18
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
         TYPE(MPI_Status) :: status
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     Fortran binding
23
     MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
24
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
25
         <type> BUF(*)
26
         MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
27
28
29
     MPI_FILE_WRITE(fh, buf, count, datatype, status)
30
                                            file handle (handle)
       INOUT
                fh
^{31}
32
       IN
                buf
                                            initial address of buffer (choice)
33
       IN
                                            number of elements in buffer (integer)
                count
34
       1N
                datatype
                                            datatype of each buffer element (handle)
35
36
       OUT
                status
                                            status object (Status)
37
38
     C binding
39
     int MPI_File_write(MPI_File fh, const void *buf, int count,
40
                    MPI_Datatype datatype, MPI_Status *status)
41
     int MPI_File_write_c(MPI_File fh, const void *buf, MPI_Count count,
42
                    MPI_Datatype datatype, MPI_Status *status)
43
44
     Fortran 2008 binding
45
     MPI_File_write(fh, buf, count, datatype, status, ierror)
46
         TYPE(MPI_File), INTENT(IN) :: fh
47
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
48
```

```
1
    INTEGER, INTENT(IN) :: count
                                                                                       2
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_write(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                      10
    TYPE(MPI_Status) :: status
                                                                                      11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      12
                                                                                      13
Fortran binding
                                                                                      14
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
                                                                                      15
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                      16
    <type> BUF(*)
                                                                                      17
    MPI_FILE_WRITE writes a file using the individual file pointer.
                                                                                      18
                                                                                      19
                                                                                      20
MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
                                                                                      21
  INOUT
           fh
                                      file handle (handle)
                                                                                      22
                                      initial address of buffer (choice)
  IN
           buf
                                                                                      23
                                                                                      ^{24}
                                      number of elements in buffer (integer)
  IN
           count
                                                                                      25
  IN
                                      datatype of each buffer element (handle)
           datatype
                                                                                      26
  OUT
           status
                                      status object (Status)
                                                                                      27
                                                                                      28
                                                                                      29
C binding
                                                                                      30
int MPI_File_write_all(MPI_File fh, const void *buf, int count,
                                                                                      31
              MPI_Datatype datatype, MPI_Status *status)
                                                                                      32
int MPI_File_write_all_c(MPI_File fh, const void *buf, MPI_Count count,
                                                                                      33
              MPI_Datatype datatype, MPI_Status *status)
                                                                                      34
                                                                                      35
Fortran 2008 binding
                                                                                      36
MPI_File_write_all(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
MPI_File_write_all(fh, buf, count, datatype, status, ierror)
                                                                                      44
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                      45
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                      46
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                      47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                      48
```

```
1
         TYPE(MPI_Status) :: status
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     Fortran binding
4
     MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
5
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
6
         <type> BUF(*)
7
8
         MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
9
     face.
10
11
     MPI_FILE_IREAD(fh, buf, count, datatype, request)
12
13
       INOUT
                fh
                                            file handle (handle)
14
       OUT
                buf
                                           initial address of buffer (choice)
15
       IN
                                           number of elements in buffer (integer)
                count
16
17
       IN
                                           datatype of each buffer element (handle)
                datatype
18
       OUT
                                           request object (handle)
                request
19
20
     C binding
21
     int MPI_File_iread(MPI_File fh, void *buf, int count,
22
                    MPI_Datatype datatype, MPI_Request *request)
23
24
     int MPI_File_iread_c(MPI_File fh, void *buf, MPI_Count count,
25
                    MPI_Datatype datatype, MPI_Request *request)
26
     Fortran 2008 binding
27
     MPI_File_iread(fh, buf, count, datatype, request, ierror)
28
         TYPE(MPI_File), INTENT(IN) :: fh
29
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
30
         INTEGER, INTENT(IN) :: count
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Request), INTENT(OUT) :: request
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_File_iread(fh, buf, count, datatype, request, ierror)
36
         TYPE(MPI_File), INTENT(IN) :: fh
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
38
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
39
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     Fortran binding
43
     MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
44
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
45
         <type> BUF(*)
46
47
         MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.
48
```

**Example 14.3** The following Fortran code fragment illustrates file pointer update semantics:

```
!
    Read the first twenty real words in a file into two local
                                                                                       4
    buffers. Note that when the first MPI_FILE_IREAD returns,
!
                                                                                       5
!
    the file pointer has been updated to point to the
                                                                                       6
Т
    eleventh real word in the file.
          bufsize, req1, req2
integer
                                                                                       9
integer, dimension(MPI_STATUS_SIZE) :: status1, status2
                                                                                       10
parameter (bufsize=10)
                                                                                       11
           buf1(bufsize), buf2(bufsize)
real
                                                                                       12
integer (kind=MPI_OFFSET_KIND) zero
                                                                                       13
                                                                                       14
zero = 0
                                                                                       15
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
                                                                                       16
                     MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
                                                                                       17
call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
                                                                                       18
                         MPI_INFO_NULL, ierr)
                                                                                       19
call MPI_FILE_IREAD(myfh, buf1, bufsize, MPI_REAL, &
                                                                                       20
                      req1, ierr)
                                                                                       21
call MPI_FILE_IREAD(myfh, buf2, bufsize, MPI_REAL,
                                                                                       22
                      req2, ierr)
                                                                                       23
                                                                                       ^{24}
call MPI_WAIT(req1, status1, ierr)
                                                                                       25
call MPI_WAIT(req2, status2, ierr)
                                                                                       26
                                                                                       27
call MPI_FILE_CLOSE(myfh, ierr)
                                                                                       28
                                                                                       29
                                                                                       30
                                                                                       31
MPI_FILE_IREAD_ALL(fh, buf, count, datatype, request)
                                                                                       32
 INOUT
                                      file handle (handle)
           fh
                                                                                       33
  OUT
           buf
                                      initial address of buffer (choice)
                                                                                       34
                                                                                       35
 ĬΝ
                                      number of elements in buffer (integer)
           count
                                                                                       36
 IN
                                      datatype of each buffer element (handle)
           datatype
                                                                                       37
 OUT
           request
                                      request object (handle)
                                                                                       38
                                                                                       39
C binding
                                                                                       40
                                                                                       41
int MPI_File_iread_all(MPI_File fh, void *buf, int count,
                                                                                       42
              MPI_Datatype datatype, MPI_Request *request)
                                                                                       43
int MPI_File_iread_all_c(MPI_File fh, void *buf, MPI_Count count,
                                                                                       44
              MPI_Datatype datatype, MPI_Request *request)
                                                                                       45
                                                                                       46
Fortran 2008 binding
                                                                                       47
MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
                                                                                       48
    TYPE(MPI_File), INTENT(IN) :: fh
```

1 2

```
1
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
2
         INTEGER, INTENT(IN) :: count
3
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
         TYPE(MPI_Request), INTENT(OUT) :: request
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
7
         TYPE(MPI_File), INTENT(IN) :: fh
8
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
9
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
10
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
16
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
17
         <type> BUF(*)
18
         MPI_FILE_IREAD_ALL is a nonblocking version of MPI_FILE_READ_ALL.
19
20
21
     MPI_FILE_IWRITE(fh, buf, count, datatype, request)
22
       INOUT
                fh
                                           file handle (handle)
23
^{24}
       IN
                buf
                                           initial address of buffer (choice)
25
       IN
                count
                                           number of elements in buffer (integer)
26
       IN
                                           datatype of each buffer element (handle)
                datatype
27
28
       OUT
                request
                                           request object (handle)
29
30
     C binding
^{31}
     int MPI_File_iwrite(MPI_File fh, const void *buf, int count,
32
                   MPI_Datatype datatype, MPI_Request *request)
33
     int MPI_File_iwrite_c(MPI_File fh, const void *buf, MPI_Count count,
34
                   MPI_Datatype datatype, MPI_Request *request)
35
36
     Fortran 2008 binding
37
     MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
40
         INTEGER, INTENT(IN) :: count
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         TYPE(MPI_Request), INTENT(OUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
45
         TYPE(MPI_File), INTENT(IN) :: fh
46
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
47
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       \mathbf{2}
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       3
Fortran binding
                                                                                       5
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
                                                                                       6
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    <type> BUF(*)
    MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.
                                                                                       9
                                                                                       10
                                                                                       11
MPI_FILE_IWRITE_ALL(fh, buf, count, datatype, request)
                                                                                       12
                                                                                       13
  INOUT
           fh
                                      file handle (handle)
                                                                                       14
  IN
           buf
                                      initial address of buffer (choice)
                                                                                       15
                                      number of elements in buffer (integer)
  IN
           count
                                                                                       16
                                                                                       17
  IN
                                      datatype of each buffer element (handle)
           datatype
                                                                                       18
  OUT
           request
                                      request object (handle)
                                                                                       19
                                                                                       20
C binding
                                                                                      21
int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count,
                                                                                       22
              MPI_Datatype datatype, MPI_Request *request)
                                                                                       23
                                                                                       ^{24}
int MPI_File_iwrite_all_c(MPI_File fh, const void *buf, MPI_Count count,
                                                                                       25
              MPI_Datatype datatype, MPI_Request *request)
                                                                                       26
Fortran 2008 binding
                                                                                       27
MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
                                                                                       28
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                       29
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                       30
    INTEGER, INTENT(IN) :: count
                                                                                       31
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       32
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                       33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      34
                                                                                      35
MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
                                                                                      36
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                      37
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                       38
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                       39
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       40
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                       41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       42
Fortran binding
                                                                                       43
MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
                                                                                       44
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
                                                                                       45
    <type> BUF(*)
                                                                                       46
                                                                                       47
    MPI_FILE_IWRITE_ALL is a nonblocking version of MPI_FILE_WRITE_ALL.
                                                                                       48
```

```
1
     MPI_FILE_SEEK(fh, offset, whence)
2
       INOUT
                 fh
                                             file handle (handle)
3
                 offset
       IN
                                             file offset (integer)
4
5
       IN
                 whence
                                             update mode (state)
6
7
     C binding
8
     int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
9
     Fortran 2008 binding
10
     MPI_File_seek(fh, offset, whence, ierror)
11
          TYPE(MPI_File), INTENT(IN) :: fh
12
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
13
          INTEGER, INTENT(IN) :: whence
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     Fortran binding
17
     MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
18
          INTEGER FH, WHENCE, IERROR
19
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
20
          MPI_FILE_SEEK updates the individual file pointer according to whence, which has the
21
     following possible values:
22
23
         • MPI_SEEK_SET: the pointer is set to offset
24
25
         • MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
26
         • MPI_SEEK_END: the pointer is set to the end of file plus offset
27
28
         The offset can be negative, which allows seeking backwards. It is erroneous to seek to
29
     a negative position in the view.
30
^{31}
32
     MPI_FILE_GET_POSITION(fh, offset)
33
       IN
                 fh
                                             file handle (handle)
34
       OUT
                 offset
                                             offset of individual pointer (integer)
35
36
37
     C binding
38
     int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
39
     Fortran 2008 binding
40
     MPI_File_get_position(fh, offset, ierror)
41
          TYPE(MPI_File), INTENT(IN) :: fh
42
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
43
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     Fortran binding
46
     MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
47
          INTEGER FH, IERROR
48
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

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

MPI_FILE	_GET_BYTE_OFFSET(fh, offse	et, disp)
IN	fh	file handle (handle)
IN	offset	offset (integer)
OUT	disp	absolute byte position of offset (integer)
C bindin int MPI_H	0	File fh, MPI_Offset offset,
MPI_File_ TYPE INTEC	2008 binding _get_byte_offset(fh, offse (MPI_File), INTENT(IN) :: GER(KIND=MPI_OFFSET_KIND), GER(KIND=MPI_OFFSET_KIND), GER, OPTIONAL, INTENT(OUT)	fh INTENT(IN) :: offset INTENT(OUT) :: disp
INTEC	Dinding _GET_BYTE_OFFSET(FH, OFFSE GER FH, IERROR GER(KIND=MPI_OFFSET_KIND)	
position.		nverts a view-relative offset into an absolute byte m the beginning of the file) of <b>offset</b> relative to the

# 14.4.4 Data Access with Shared File Pointers

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.

The shared file pointer routines have the same semantics as the data access with explicit offset routines described in Section 14.4.2, with the following modifications:

- the offset is defined to be the current value of the MPI-maintained shared file pointer,
- the effect of multiple calls to shared file pointer routines is defined to behave as if the calls were serialized, and

# Unofficial Draft for Comment Only

 $\overline{7}$ 

 $46 \\ 47$ 

```
1
        • the use of shared file pointer routines is erroneous unless all processes use the same
\mathbf{2}
           file view.
3
     For the noncollective shared file pointer routines, the serialization ordering is not determin-
4
     istic. The user needs to use other synchronization means to enforce a specific order.
5
          After a shared file pointer operation is initiated, the shared file pointer is updated to
6
     point to the next etype after the last one that will be accessed. The file pointer is updated
7
     relative to the current view of the file.
8
9
     Noncollective Operations
10
11
12
13
     MPI_FILE_READ_SHARED(fh, buf, count, datatype, status)
14
       INOUT
                 fh
                                             file handle (handle)
15
       OUT
                 buf
                                             initial address of buffer (choice)
16
17
       IN
                 count
                                             number of elements in buffer (integer)
18
       IN
                 datatype
                                             datatype of each buffer element (handle)
19
       OUT
                 status
                                             status object (Status)
20
21
     C binding
22
     int MPI_File_read_shared(MPI_File fh, void *buf, int count,
23
                    MPI_Datatype datatype, MPI_Status *status)
^{24}
25
     int MPI_File_read_shared_c(MPI_File fh, void *buf, MPI_Count count,
26
                    MPI_Datatype datatype, MPI_Status *status)
27
     Fortran 2008 binding
28
     MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
29
30
          TYPE(MPI_File), INTENT(IN) :: fh
          TYPE(*), DIMENSION(..) :: buf
31
          INTEGER, INTENT(IN) :: count
32
33
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
          TYPE(MPI_Status) :: status
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
37
          TYPE(MPI_File), INTENT(IN) :: fh
38
          TYPE(*), DIMENSION(..) :: buf
39
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
40
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
          TYPE(MPI_Status) :: status
42
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     Fortran binding
45
     MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
46
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
47
          <type> BUF(*)
48
```

Ν	IPI_FILE_READ_SHAR	RED reads a file using the shared file pointer.	1
MDI I		(fh, buf, count, datatype, status)	3
			4
INO		file handle (handle)	5 6
IN	buf	initial address of buffer (choice)	7
IN	count	number of elements in buffer (integer)	8
IN	datatype	datatype of each buffer element (handle)	9
OUT	status	status object (Status)	10 11
			12
C bir int M	PI_File_write_share	ed(MPI_File fh, const void *buf, int count, pe datatype, MPI_Status *status)	13 14 15
int M		ed_c(MPI_File fh, const void *buf, MPI_Count count pe datatype, MPI_Status *status)	17
Fortr	an 2008 binding		18
		n, buf, count, datatype, status, ierror)	19 20
	YPE(MPI_File), INTE		21
		), INTENT(IN) :: buf	22
	NTEGER, INTENT(IN)		23
	YPE(MPI_Datatype), YPE(MPI_Status) ::	INTENT(IN) :: datatype	24
		INTENT(OUT) :: ierror	25
			26 27
		n, buf, count, datatype, status, ierror)	21
	YPE(MPI_File), INTE	LNI(IN) :: in , INTENT(IN) :: buf	29
		JNT_KIND), INTENT(IN) :: count	30
		INTENT(IN) :: datatype	31
	YPE(MPI_Status) ::		32
I	NTEGER, OPTIONAL, I	INTENT(OUT) :: ierror	33
Fortr	an binding		34
	U U	I, BUF, COUNT, DATATYPE, STATUS, IERROR)	35 36
		DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	37
<	type> BUF(*)		38
N	1 PL FILE WRITE SHA	RED writes a file using the shared file pointer.	39
IV		KED writes a me using the shared me pointer.	40
			41
			42
			43
			44 45
			45
			47

```
1
     MPI_FILE_IREAD_SHARED(fh, buf, count, datatype, request)
2
       INOUT
                fh
                                            file handle (handle)
3
       OUT
                buf
                                           initial address of buffer (choice)
4
5
       IN
                                           number of elements in buffer (integer)
                count
6
       IN
                datatype
                                           datatype of each buffer element (handle)
7
       OUT
                request
                                           request object (handle)
8
9
10
     C binding
     int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
11
                    MPI_Datatype datatype, MPI_Request *request)
12
13
     int MPI_File_iread_shared_c(MPI_File fh, void *buf, MPI_Count count,
14
                    MPI_Datatype datatype, MPI_Request *request)
15
16
     Fortran 2008 binding
17
     MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
18
         TYPE(MPI_File), INTENT(IN) :: fh
19
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
         INTEGER, INTENT(IN) :: count
20
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
         TYPE(MPI_Request), INTENT(OUT) :: request
22
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
27
         INTEGER(KIND=MPI_COUNT_KIND), 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
     Fortran binding
33
     MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
34
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
35
          <type> BUF(*)
36
         MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
37
     interface.
38
39
40
41
42
43
44
45
46
47
48
```

MPI_FILE	_IWRITE_SHARED(fh, buf, co	unt, datatype, request)	1
INOUT	fh	file handle (handle)	2
IN	buf	initial address of buffer (choice)	3 4
IN	count	number of elements in buffer (integer)	5
IN			6
	datatype	datatype of each buffer element (handle)	7
OUT	request	request object (handle)	8
Chindin	~		9 10
C bindin	-	le fh, const void *buf, int count,	10
1110 III 1_1		e, MPI_Request *request)	12
· ·			13
int MPI_B		File fh, const void *buf, MPI_Count count, e, MPI_Request *request)	14
		e, mri_wequest *request)	15
	2008 binding		16
	_iwrite_shared(fh, buf, c (MPI_File), INTENT(IN) ::	ount, datatype, request, ierror)	17 18
	· – · · · · ·	III T(IN), ASYNCHRONOUS :: buf	19
	GER, INTENT(IN) :: count		20
	(MPI_Datatype), INTENT(IN	) :: datatype	21
TYPE	(MPI_Request), INTENT(OUT	) :: request	22
INTEC	GER, OPTIONAL, INTENT(OUT	) :: ierror	23
MPI_File_	_iwrite_shared(fh, buf, c	ount, datatype, request, ierror)	24 25
	(MPI_File), INTENT(IN) ::		25 26
		T(IN), ASYNCHRONOUS :: buf	27
	GER(KIND=MPI_COUNT_KIND),		28
	(MPI_Datatype), INTENT(IN (MPI_Request), INTENT(OUT		29
	GER, OPTIONAL, INTENT(OUT	•	30
			31
Fortran l	<u> </u>		32 33
	_IWRITE_SHARED(FH, BUF, C GER FH, COUNT, DATATYPE, I	OUNT, DATATYPE, REQUEST, IERROR)	34
	<pre>sek FII, COONI, DATATIFE, 1 &gt;&gt; BUF(*)</pre>		35
			36
	FILE_IWRITE_SHARED is a	nonblocking version of the	37
WPI_FILE	WRITE_SHARED interface.		38
Collective	Operations		39
Concerne			40

The semantics of a collective access using a shared file pointer is that the accesses to the <sup>41</sup> file will be in the order determined by the ranks of the processes within the group. For each process, the location in the file at which data is accessed is the position at which the shared file pointer would be after all processes whose ranks within the group less than that of this process had accessed their data. In addition, in order to prevent subsequent shared offset accesses by the same processes from interfering with this collective access, the call might return only after all the processes within the group have initiated their accesses. When the 48

1 2 3		s, the shared file pointer point by all processes, after the last	is to the next etype accessible, according to the file setype requested.
4 5 6 7 8 9	need <i>quire</i> share MPI_	to access the file using the that data be accessed in or d ordered routines (e.g., MP	some programs in which all processes in the group shared file pointer, but the program may not <i>re</i> - der of process rank. In such programs, using the I_FILE_WRITE_ORDERED rather than enable an implementation to optimize access, im- <i>ice to users.</i> )
10 11 12 13 14	to be for al	serialized. Once all processes	s to the data requested by all processes do not have have issued their requests, locations within the file and accesses can proceed independently from each f advice to implementors.)
15 16			
17	MPL FILE	READ_ORDERED(fh, buf, co	unt, datatype, status)
18	INOUT	fh	file handle (handle)
19 20	OUT	buf	initial address of buffer (choice)
21	IN	count	number of elements in buffer (integer)
22	IN	datatype	datatype of each buffer element (handle)
23 24	OUT	status	status object (Status)
25 26 27 28 29 30		ile_read_ordered(MPI_File MPI_Datatype datatyp ile_read_ordered_c(MPI_F	e fh, void *buf, int count, e, MPI_Status *status) ile fh, void *buf, MPI_Count count, e, MPI_Status *status)
31 32	Fortran 2	008 binding	
33 33 34 35 36 37 38 39 40	MPI_File_ TYPE( TYPE( INTEG TYPE( TYPE(	U U	) :: datatype
40 41 42 43 44 45 46 47 48	TYPE () TYPE ( INTEG TYPE () TYPE ()	<pre>MPI_File), INTENT(IN) :: *), DIMENSION() :: buf ER(KIND=MPI_COUNT_KIND), MPI_Datatype), INTENT(IN) MPI_Status) :: status ER, OPTIONAL, INTENT(OUT)</pre>	INTENT(IN) :: count ) :: datatype

MPI\_FILE\_READ\_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI\_STATUS\_SIZE), IERROR <type> BUF(\*) MPI\_FILE\_READ\_ORDERED is a collective version of the MPI\_FILE\_READ\_SHARED interface. MPI\_FILE\_WRITE\_ORDERED(fh, buf, count, datatype, status) INOUT file handle (handle) fh 10 IN buf initial address of buffer (choice) 11 12IN count number of elements in buffer (integer) 13 IN datatype datatype of each buffer element (handle) 14OUT status status object (Status) 151617 C binding 18 int MPI\_File\_write\_ordered(MPI\_File fh, const void \*buf, int count, 19 MPI\_Datatype datatype, MPI\_Status \*status) 20int MPI\_File\_write\_ordered\_c(MPI\_File fh, const void \*buf, MPI\_Count count, 21MPI\_Datatype datatype, MPI\_Status \*status) 22 23Fortran 2008 binding  $^{24}$ MPI\_File\_write\_ordered(fh, buf, count, datatype, status, ierror) 25TYPE(MPI\_File), INTENT(IN) :: fh 26TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 27INTEGER, INTENT(IN) :: count 28TYPE(MPI\_Datatype), INTENT(IN) :: datatype 29 TYPE(MPI\_Status) :: status 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 31MPI\_File\_write\_ordered(fh, buf, count, datatype, status, ierror) 32 TYPE(MPI\_File), INTENT(IN) :: fh 33 TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 34 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 35TYPE(MPI\_Datatype), INTENT(IN) :: datatype 36 TYPE(MPI\_Status) :: status 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 Fortran binding 40 MPI\_FILE\_WRITE\_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 41 INTEGER FH, COUNT, DATATYPE, STATUS(MPI\_STATUS\_SIZE), IERROR 42<type> BUF(\*) 43 MPI\_FILE\_WRITE\_ORDERED is a collective version of the MPI\_FILE\_WRITE\_SHARED 44interface. 454647

1

 $\mathbf{2}$ 

3

 $\mathbf{4}$ 

5

6 7

9

```
1
     Seek
\mathbf{2}
     If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
3
     to call the following two routines (MPI_FILE_SEEK_SHARED and
4
     MPI_FILE_GET_POSITION_SHARED).
5
6
\overline{7}
     MPI_FILE_SEEK_SHARED(fh, offset, whence)
8
       INOUT
                 fh
                                              file handle (handle)
9
       IN
                 offset
                                              file offset (integer)
10
11
       IN
                 whence
                                              update mode (state)
12
13
     C binding
14
     int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
15
16
     Fortran 2008 binding
17
     MPI_File_seek_shared(fh, offset, whence, ierror)
18
          TYPE(MPI_File), INTENT(IN) :: fh
19
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
20
          INTEGER, INTENT(IN) :: whence
21
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     Fortran binding
23
     MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
24
          INTEGER FH, WHENCE, IERROR
25
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
26
27
          MPI_FILE_SEEK_SHARED updates the shared file pointer according to whence, which
28
     has the following possible values:
29
         • MPI_SEEK_SET: the pointer is set to offset
30
31
         • MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
32
33
         • MPI_SEEK_END: the pointer is set to the end of file plus offset
34
         MPI_FILE_SEEK_SHARED is collective; all the processes in the communicator group
35
     associated with the file handle fh must call MPI_FILE_SEEK_SHARED with the same values
36
     for offset and whence.
37
          The offset can be negative, which allows seeking backwards. It is erroneous to seek to
38
     a negative position in the view.
39
40
41
     MPI_FILE_GET_POSITION_SHARED(fh, offset)
42
       IN
                 fh
                                              file handle (handle)
43
44
       OUT
                 offset
                                              offset of shared pointer (integer)
45
46
     C binding
47
     int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
48
```

## Fortran 2008 binding

<pre>ition_shared(fh, offset, ierror)</pre>
e), INTENT(IN) :: fh
=MPI_OFFSET_KIND), INTENT(OUT) :: offset
IONAL, INTENT(OUT) :: ierror
e), INTENT(IN) :: fh =MPI_OFFSET_KIND), INTENT(OUT) :: offset

#### Fortran binding

MPI\_FILE\_GET\_POSITION\_SHARED(FH, OFFSET, IERROR)
 INTEGER FH, IERROR
 INTEGER(KIND=MPI\_OFFSET\_KIND) OFFSET

MPI\_FILE\_GET\_POSITION\_SHARED returns, in offset, the current position of the shared file pointer in etype units relative to the current view.

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.*)

# 14.4.5 Split Collective Data Access Routines

MPI provides a restricted form of "nonblocking collective" I/O operations for all data accesses using split collective data access routines. These routines are referred to as "split" collective routines because a single collective operation is split in two: a begin routine and an end routine. The begin routine begins the operation, much like a nonblocking data access (e.g., MPI\_FILE\_IREAD). The end routine completes the operation, much like the matching test or wait (e.g., MPI\_WAIT). As with nonblocking data access operations, the user must not use the buffer passed to a begin routine while the routine is outstanding; the operation must be completed with an end routine before it is safe to free buffers, etc.

Split collective data access operations on a file handle fh are subject to the semantic rules given below.

- On any MPI process, each file handle may have at most one active split collective operation at any time.
- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.
- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an "end" call is made, exactly one unmatched "begin" call for the same operation must precede it.
- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., MPI\_FILE\_READ\_ALL\_BEGIN) or the end call (e.g., MPI\_FILE\_READ\_ALL\_END) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.

1 2 3	According to the definitions in Section 2.4.2, 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.
4 5 6	Advice to users. This is one of the exceptions in which incomplete procedures are non-local and therefore blocking. (End of advice to users.)
7 8 9 10 11	<ul> <li>Split collective operations do not match the corresponding regular collective opera- tion. 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.</li> </ul>
12 13 14 15 16	• Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid the problems described in "A Problem with Code Movements and Register Optimization," Section 19.1.17, but not all of the problems, such as those described in Sections 19.1.12, 19.1.13, and 19.1.16.
17 18 19 20	• 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
21 22	<pre>MPI_File_read_all_begin(fh,);</pre>
23	
24	<pre>MPI_File_read_all(fh,);</pre>
25	
26	<pre>MPI_File_read_all_end(fh,);</pre>
27	
28	is erroneous.
29	• In a multithreaded implementation, any split collective begin and end operation called
30	by a process must be called from the same thread. This restriction is made to simplify
31 32	the implementation in the multithreaded case. (Note that we have already disallowed
33	having two threads begin a split collective operation on the same file handle since only
34	one split collective operation can be active on a file handle at any time.)
35	The arguments for these routines have the same meaning as for the equivalent collective
36	versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and
37	MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL).
38	The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation
39	that, when completed with the matching end routine (i.e., MPI_FILE_READ_ALL_END)
40	produces the result as defined for the equivalent collective routine (i.e.,
41 42	MPI_FILE_READ_ALL).
42	For the purpose of consistency semantics (Section 14.6.1), a matched pair of split
44	collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compare a single data access
	collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access.
44	
44 45	

MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype) <sup>1</sup>					
IN	fh	file handle (handle)	2 3		
IN	offset	file offset (integer)	3		
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		
IIN	datatype	datatype of each buner element (nandie)	8 9		
C bindin	C binding				
int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,					
	int count, MPI_Datat	ype datatype)	12		
<pre>int MPI_File_read_at_all_begin_c(MPI_File fh, MPI_Offset offset, void *buf,</pre>					
MPI_Count count, MPI_Datatype datatype)					
Fortran 2008 binding					
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)					
TYPE(MPI_File), INTENT(IN) :: fh					
<pre>INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset</pre>					
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf					
INTEGER, INTENT(IN) :: count					
TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, OPTIONAL, INTENT(OUT) :: ierror			22 23		
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)					
TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset					
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf					
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count					
TYPE(MPI_Datatype), INTENT(IN) :: datatype					
INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
Fortran binding					
		FSET, BUF, COUNT, DATATYPE, IERROR)	33 34		
INTEGER FH, COUNT, DATATYPE, IERROR					
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type>					
(c) her por(*)					
			38		
MPI_FILE_READ_AT_ALL_END(fh, buf, status)			39		
IN	fh	file handle (handle)	40		
OUT	buf	initial address of buffer (choice)	41 42		
			43		
OUT	status	status object (Status)	44		
C bindin	C binding				
	int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)				
Fortran 2008 binding					

```
678
```

```
1
     MPI_File_read_at_all_end(fh, buf, status, ierror)
\mathbf{2}
         TYPE(MPI_File), INTENT(IN) :: fh
3
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
4
         TYPE(MPI_Status) :: status
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     Fortran binding
7
     MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
8
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
9
         <type> BUF(*)
10
11
12
     MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
13
14
       INOUT
                fh
                                           file handle (handle)
15
       IN
                offset
                                           file offset (integer)
16
                buf
                                           initial address of buffer (choice)
       IN
17
18
                                           number of elements in buffer (integer)
       IN
                count
19
                                           datatype of each buffer element (handle)
       IN
                datatype
20
21
     C binding
22
     int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
23
                    const void *buf, int count, MPI_Datatype datatype)
24
25
     int MPI_File_write_at_all_begin_c(MPI_File fh, MPI_Offset offset,
26
                    const void *buf, MPI_Count count, MPI_Datatype datatype)
27
     Fortran 2008 binding
28
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
29
         TYPE(MPI_File), INTENT(IN) :: fh
30
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
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_at_all_begin(fh, offset, buf, count, datatype, ierror)
37
         TYPE(MPI_File), INTENT(IN) :: fh
38
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
39
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
40
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
     Fortran binding
44
     MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
45
         INTEGER FH, COUNT, DATATYPE, IERROR
46
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
47
         <type> BUF(*)
48
```

MPI_FILE	_WRITE_AT_ALL_	END(fh, buf, status)	1	
INOUT	fh	file handle (handle)	2	
IN	buf	initial address of buffer (choice)	3 4	
OUT	status	status object (Status)	5	
001	514145		6	
C bindin	ıg		7	
int MPI_File_write_at_all_end(MPI_File fh, const void *buf,				
	MPI_Status	*status)	9	
Fortran	2008 binding		10 11	
		d(fh, buf, status, ierror)	12	
TYPE	(MPI_File), INTE	NT(IN) :: fh	13	
		.), INTENT(IN), ASYNCHRONOUS :: buf	14	
	(MPI_Status) ::		15	
	GER, UPIIUNAL, I	NTENT(OUT) :: ierror	16	
Fortran	•		17 18	
MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)				
	GER FH, STATUS(M e> BUF(*)	PI_STATUS_SIZE), IERROR	20	
Cup	e> Dor(*)		21	
			22	
MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype)				
INOUT		file handle (handle)	24 25	
			26	
OUT	buf	initial address of buffer (choice)	27	
IN	count	number of elements in buffer (integer)	28	
IN	datatype	datatype of each buffer element (handle)	29	
			30	
C bindin	-		31 32	
int MPI_		gin(MPI_File fh, void *buf, int count,	33	
	MP1_Datatyp	be datatype)	34	
int MPI_		gin_c(MPI_File fh, void *buf, MPI_Count count,	35	
	MPI_Datatyp	be datatype)	36	
Fortran 2008 binding				
<pre>MPI_File_read_all_begin(fh, buf, count, datatype, ierror)</pre>				
TYPE(MPI_File), INTENT(IN) :: fh TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count				
	• -	NTENT(OUT) :: ierror	43	
			$44 \\ 45$	
<pre>MPI_File_read_all_begin(fh, buf, count, datatype, ierror)</pre>				
		.), ASYNCHRONOUS :: buf	46 47	
		NT_KIND), INTENT(IN) :: count	48	

```
680
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     Fortran binding
4
     MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
5
          INTEGER FH, COUNT, DATATYPE, IERROR
6
         <type> BUF(*)
7
8
9
     MPI_FILE_READ_ALL_END(fh, buf, status)
10
11
       INOUT
                fh
                                            file handle (handle)
12
       OUT
                                            initial address of buffer (choice)
                 buf
13
       OUT
                                            status object (Status)
                status
14
15
16
     C binding
17
     int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
18
     Fortran 2008 binding
19
     MPI_File_read_all_end(fh, buf, status, ierror)
20
         TYPE(MPI_File), INTENT(IN) :: fh
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
22
         TYPE(MPI_Status) :: status
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
     Fortran binding
26
     MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
27
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
28
          <type> BUF(*)
29
30
31
     MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
32
       INOUT
                                            file handle (handle)
                 fh
33
34
       IN
                 buf
                                            initial address of buffer (choice)
35
       IN
                count
                                            number of elements in buffer (integer)
36
       IN
                 datatype
                                            datatype of each buffer element (handle)
37
38
     C binding
39
     int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
40
                    MPI_Datatype datatype)
41
42
     int MPI_File_write_all_begin_c(MPI_File fh, const void *buf,
43
                    MPI_Count count, MPI_Datatype datatype)
44
     Fortran 2008 binding
45
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
46
47
         TYPE(MPI_File), INTENT(IN) :: fh
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
48
```

	ER, INTENT(IN) :: count		1
	<pre>MPI_Datatype), INTENT(IN)</pre>		2
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	3
MPI_File_write_all_begin(fh, buf, count, datatype, ierror)			4 5
TYPE(	MPI_File), INTENT(IN) ::	fh	6
TYPE(	*), DIMENSION(), INTENT	C(IN), ASYNCHRONOUS :: buf	7
	ER(KIND=MPI_COUNT_KIND),		8
	MPI_Datatype), INTENT(IN)		9
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	10
Fortran b	binding		11
	WRITE_ALL_BEGIN(FH, BUF,	COUNT, DATATYPE, IERROR)	12
	ER FH, COUNT, DATATYPE, ]		13
<type< td=""><td>&gt; BUF(*)</td><td></td><td>14</td></type<>	> BUF(*)		14
			15
			16
MPI_FILE_	WRITE_ALL_END(fh, buf, sta	itus)	17
INOUT	fh	file handle (handle)	18 19
IN	buf	initial address of buffer (choice)	20
OUT	status	status object (Status)	21
001	Status	status object (status)	22
C binding			23
int MPI_File_write_all_end(MPI_File fh, const void *buf,			24
1110 PH 1_1	MPI_Status *status)		25
_			26
	008 binding		27
	write_all_end(fh, buf, st		28 29
	<pre>MPI_File), INTENT(IN) :: **) DIMENSION( ) INTENT</pre>		30
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf TYPE(MPI_Status) :: status			31
	ER, OPTIONAL, INTENT(OUT)	···ierror	32
INTEG			33
Fortran b	2		34
	WRITE_ALL_END(FH, BUF, ST		35
	ER FH, STATUS(MPI_STATUS	SIZE), IERROR	36
<type< td=""><td>&gt; BUF(*)</td><td></td><td>37</td></type<>	> BUF(*)		37
			38
			39
WPI_FILE_	READ_ORDERED_BEGIN(fh,	but, count, datatype)	40
INOUT	fh	file handle (handle)	41 42
OUT	buf	initial address of buffer (choice)	42
IN	count	number of elements in buffer (integer)	44
IN	datatype	datatype of each buffer element (handle)	45
IIN	uuuuype	davatype of each burier cientent (nanule)	46
C binding			47
C binding	6		48

```
1
     int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
\mathbf{2}
                   MPI_Datatype datatype)
3
     int MPI_File_read_ordered_begin_c(MPI_File fh, void *buf, MPI_Count count,
4
                   MPI_Datatype datatype)
5
6
     Fortran 2008 binding
\overline{7}
     MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
8
         TYPE(MPI_File), INTENT(IN) :: fh
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
10
         INTEGER, INTENT(IN) :: count
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_File_read_ordered_begin(fh, buf, count, datatype,
                                                                ierror)
14
         TYPE(MPI_File), INTENT(IN) :: fh
15
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     Fortran binding
21
     MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
22
         INTEGER FH, COUNT, DATATYPE, IERROR
23
         <type> BUF(*)
^{24}
25
26
     MPI_FILE_READ_ORDERED_END(fh, buf, status)
27
       INOUT
                                           file handle (handle)
                fh
28
29
                buf
       OUT
                                           initial address of buffer (choice)
30
       OUT
                                           status object (Status)
                status
^{31}
32
     C binding
33
     int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
34
35
     Fortran 2008 binding
36
     MPI_File_read_ordered_end(fh, buf, status, ierror)
37
         TYPE(MPI_File), INTENT(IN) :: fh
38
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
39
         TYPE(MPI_Status) :: status
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     Fortran binding
42
     MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
43
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
44
         <type> BUF(*)
45
46
47
48
```

MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype) <sup>1</sup>			
INOUT	fh	file handle (handle)	2
IN	buf	initial address of buffer (choice)	3 4
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6
			7 8
C binding	g		9
int MPI_F	ile_write_ordered_begin() MPI_Datatype datatyp	MPI_File fh, const void *buf, int count, e)	10 11
int MPI_F	Sile_write_ordered_begin_	c(MPI_File fh, const void *buf,	12
	MPI_Count count, MPI	_Datatype datatype)	13 14
Fortran 2	2008 binding		14 15
	0	buf, count, datatype, ierror)	16
	<pre>(MPI_File), INTENT(IN) ::</pre>	fh T(IN), ASYNCHRONOUS :: buf	17
	ER, INTENT(IN) :: count	I(IN), ASINCHRUNUUS :: DUI	18 19
	(MPI_Datatype), INTENT(IN)	) :: datatype	19 20
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	21
MPI_File_	write_ordered_begin(fh,	buf, count, datatype, ierror)	22
	(MPI_File), INTENT(IN) ::		23
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf			24
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype			25 26
	MPI_Datatype), INTENT(IN ER, OPTIONAL, INTENT(OUT)		27
			28
Fortran k			29
	ER FH, COUNT, DATATYPE,	BUF, COUNT, DATATYPE, IERROR)	30
	> BUF(*)		31 32
51			33
			34
MPI_FILE	WRITE_ORDERED_END(fh,	buf, status)	35
INOUT	fh	file handle (handle)	36
IN	buf	initial address of buffer (choice)	37 38
OUT	status	status object (Status)	39
			40
C binding	5		41
int MPI_F	lile_write_ordered_end(MP	I_File fh, const void *buf,	42
	MPI_Status *status)		43
Fortran 2008 binding			44 45
MPI_File_write_ordered_end(fh, buf, status, ierror)			
	<pre>MPI_File), INTENT(IN) ::</pre>		47
TYPE(	*), DIMENSION(), INTEN	T(IN), ASYNCHRONOUS :: buf	48

```
1
          TYPE(MPI_Status) :: status
2
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
      Fortran binding
4
      MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
5
          INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
6
          <type> BUF(*)
7
8
9
      14.5
              File Interoperability
10
11
      At the most basic level, file interoperability is the ability to read the information previously
12
      written to a file—not just the bits of data, but the actual information the bits represent.
13
      MPI guarantees full interoperability within a single MPI environment, and supports in-
14
      creased interoperability outside that environment through the external data representation
15
      (Section 14.5.2) as well as the data conversion functions (Section 14.5.3).
16
          Interoperability within a single MPI environment (which could be considered "oper-
17
      ability") ensures that file data written by one MPI process can be read by any other MPI
18
      process, subject to the consistency constraints (see Section 14.6.1), provided that it would
19
      have been possible to start the two processes simultaneously and have them reside in a
20
      single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at
21
      every absolute byte offset in the file for which data was written.
22
          This single environment file interoperability implies that file data is accessible regardless
23
      of the number of processes.
24
          There are three aspects to file interoperability:
25
26
         • transferring the bits.
27
         • converting between different file structures, and
28
29
         • converting between different machine representations.
30
^{31}
          The first two aspects of file interoperability are beyond the scope of this standard,
32
      as both are highly machine dependent. However, transferring the bits of a file into and
33
      out of the MPI environment (e.g., by writing a file to tape) is required to be supported
34
      by all MPI implementations. In particular, an implementation must specify how familiar
35
      operations similar to POSIX cp, rm, and mv can be performed on the file. Furthermore, it
36
      is expected that the facility provided maintains the correspondence between absolute byte
37
      offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the
38
      MPI environment are at byte offset 102 outside the MPI environment). As an example,
39
      a simple off-line conversion utility that transfers and converts files between the native file
40
      system and the MPI environment would suffice, provided it maintained the offset coherence
41
      mentioned above. In a high-quality implementation of MPI, users will be able to manipulate
42
      MPI files using the same or similar tools that the native file system offers for manipulating
43
      its files.
44
          The remaining aspect of file interoperability, converting between different machine
45
      representations, is supported by the typing information specified in the etype and filetype.
46
      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 14.5.3). 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 application is responsible for specifying the same data representation as was used to create the file. (*End of advice to users.*)

"native" Data in this representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.

Advice to users. This data representation should only be used in a homogeneous MPI environment, or when the MPI application is capable of performing the data type conversions itself. (*End of advice to users.*)

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be typed as MPI\_BYTE to ensure that the message routines do not perform any type conversions on the data. (End of advice to implementors.)

"internal" This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file. The environment in which the resulting file can be reused is implementationdefined and must be documented by the implementation.

Rationale. This data representation allows the implementation to perform I/O efficiently in a heterogeneous environment, though with implementation-defined restrictions on how the file can be reused. (*End of rationale.*)

Advice to implementors. Since "external32" is a superset of the functionality provided by "internal", an implementation may choose to implement "internal" as "external32". (End of advice to implementors.)

"external32" This data representation states that read and write operations convert all data from and to the "external32" representation defined in Section 14.5.2. The data conversion rules for communication also apply to these conversions (see Section 3.3.2). 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.

This data representation has several advantages. First, all processes reading the file <sup>46</sup> in a heterogeneous MPI environment will automatically have the data converted to <sup>47</sup> their respective native representations. Second, the file can be exported from one MPI <sup>48</sup>

#### 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

environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file.

The disadvantage of this data representation is that data precision and I/O performance may be lost in data type conversions.

> Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be converted to and from the "external32" representation in the client, and sent as type MPI\_BYTE. This will avoid possible double data type conversions and the associated further loss of precision and performance. (End of advice to implementors.)

12

#### 14.5.1Datatypes for File Interoperability

If the file data representation is other than "native", care must be taken in constructing 14etypes and filetypes. Any of the datatype constructor functions may be used; however, 15for those functions that accept displacements in bytes, the displacements must be specified 1617in 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 18

MPI\_FILE\_GET\_TYPE\_EXTENT can be used to calculate the extents of datatypes in the 19file. For etypes and filetypes that are portable datatypes (see Section 2.4), MPI will scale 2021any 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 22always be constructed using displacements corresponding to displacements in memory. 23

 $^{24}$ 25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

41

One can logically think of the file as if it were stored in the Advice to users. 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 MPL\_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 40 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 42portable, and has a typemap or extent that is architecture dependent, then the data 43 layout specified by it on a file is implementation dependent. 44
- 45File data representations other than "native" may be different from corresponding 46data representations in memory. Therefore, for these file data representations, it is 47 important not to use hardwired byte offsets for file positioning, including the initial 48 displacement that specifies the view. When a portable datatype (see Section 2.4) is

1

2

3

4

56

7

8

9

10

11

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.)

			9
			10
MPI_FILE_	GET_TYPE_EXTENT(fh, data	atype, extent)	11
IN	fh	file handle (handle)	12
IN	datatype	datatype (handle)	13
OUT	extent	datatype extent (integer)	14
001	extent	datatype extent (meger)	15
Chindina	_		16
C binding	•	Cile fb MDT Deteture deteture	17
int MPI_F	<pre>IIe_get_type_extent(MPI_F</pre>	File fh, MPI_Datatype datatype,	18
	MPI_AINt *extent)		19
int MPI_F	<pre>ile_get_type_extent_c(MP)</pre>	_File fh, MPI_Datatype datatype,	20 21
	MPI_Count *extent)		21 22
Fortran 2	008 binding		22
	0	vpe extent jerror)	24
<pre>MPI_File_get_type_extent(fh, datatype, extent, ierror)     TYPE(MPI_File), INTENT(IN) :: fh</pre>			25
TYPE(MPI_Datatype), INTENT(IN) :: datatype			26
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent			27
	ER, OPTIONAL, INTENT(OUT)		28
NDT D:1.			29
<pre>MPI_File_get_type_extent(fh, datatype, extent, ierror)     TYPE(MPI_File), INTENT(IN) :: fh</pre>			30
			31
	<pre>MPI_Datatype), INTENT(IN) ER(KIND=MPI_COUNT_KIND),</pre>		32
	ER, OPTIONAL, INTENT(OUT)		33
INTEG	ER, OFFICIARE, INTENT(001)		34
Fortran b	inding		35
MPI_FILE_	GET_TYPE_EXTENT(FH, DATA)	TYPE, EXTENT, IERROR)	36
	ER FH, DATATYPE, IERROR		37
INTEG	ER(KIND=MPI_ADDRESS_KIND)	EXTENT	38
Return	ns the extent of datatype in	the file fh. This extent will be the same for all	39
		rrent view uses a user-defined data representation	40

Returns the extent of datatype in the file fh. This extent will be the same for all processes accessing the file fh. If the current view uses a user-defined data representation (see Section 14.5.3), MPI uses the dtype\_file\_extent\_fn callback to calculate the extent.

If the datatype extent cannot be represented in extent, it is set to MPI\_UNDEFINED.

Advice to implementors. In the case of user-defined data representations, the extent of a derived datatype can be calculated by first determining the extents of the predefined datatypes in this derived datatype using dtype\_file\_extent\_fn (see Section 14.5.3). (End of advice to implementors.) 1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8 9

41

42

43 44

45

46

47

#### 1 External Data Representation: "external32" 14.5.2 $\mathbf{2}$ All MPI implementations are required to support the data representation defined in this 3 section. Support of optional datatypes (e.g., MPI\_INTEGER2) is not required. 4 All floating point values are in big-endian IEEE format [42] of the appropriate size. 5Floating point values are represented by one of three IEEE formats. These are the IEEE 6 "Single (binary32)," "Double (binary64)," and "Double Extended (binary128)" formats, 7 requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended 8 (binary128)" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, 9 bias = +16383, 112 fraction bits, and an encoding analogous to the "Double (binary64)" 10 format. All integral values are in two's complement big-endian format. Big-endian means 11 most significant byte at lowest address byte. For C $\_Bool$ , Fortran LOGICAL, and C++ bool, 120 implies false and nonzero implies true. C float \_Complex, double \_Complex, and long 13 double \_Complex, Fortran COMPLEX and DOUBLE COMPLEX, and other complex types are 14represented by a pair of floating point format values for the real and imaginary components. 15Characters are in ISO 8859-1 format [43]. Wide characters (of type MPI\_WCHAR) are in 16Unicode format [68]. 17All signed numerals (e.g., MPI\_INT, MPI\_REAL) have the sign bit at the most significant 18 bit. MPI\_COMPLEX and MPI\_DOUBLE\_COMPLEX have the sign bit of the real and imaginary 19parts at the most significant bit of each part. 20According to IEEE specifications [42], the "NaN" (not a number) is system dependent. 21It should not be interpreted within MPI as anything other than "NaN." 22 23Advice to implementors. The MPI treatment of "NaN" is similar to the approach $^{24}$ used in XDR [65]. (End of advice to implementors.) 2526All data is byte aligned, regardless of type. All data items are stored contiguously in 27the file (if the file view is contiguous). 2829Advice to implementors. All bytes of LOGICAL and bool must be checked to determine 30 the value. (End of advice to implementors.) 3132 Advice to users. The type MPI\_PACKED is treated as bytes and is not converted. 33 The user should be aware that MPI\_PACK has the option of placing a header in the 34 beginning of the pack buffer. (End of advice to users.) 35 36 The sizes of the predefined datatypes returned from MPI\_TYPE\_CREATE\_F90\_REAL, 37 MPI\_TYPE\_CREATE\_F90\_COMPLEX, and MPI\_TYPE\_CREATE\_F90\_INTEGER are defined in Section 19.1.9, page 801. 3839 Advice to implementors. When converting a larger size integer to a smaller size 40 integer, only the least significant bytes are moved. Care must be taken to preserve 41 the sign bit value. This allows no conversion errors if the data range is within the 42range of the smaller size integer. (End of advice to implementors.) 43 44Table 14.2, 14.3, and 14.4 specify the sizes of predefined, optional, and C++ datatypes 45in "external32" format, respectively. 464748

Predefined Type	Length	1
MPI_PACKED	1	2
MPI_BYTE	1	3
MPI_CHAR	1	4
MPI_UNSIGNED_CHAR	1	5
MPI_SIGNED_CHAR	1	6
MPI_WCHAR	2	7
MPI_SHORT	$\frac{2}{2}$	8
	$\frac{2}{2}$	9
MPI_UNSIGNED_SHORT		10
MPI_INT	4	10
MPI_LONG	4	
MPI_UNSIGNED	4	12
MPI_UNSIGNED_LONG	4	13
MPI_LONG_LONG_INT	8	14
MPI_UNSIGNED_LONG_LONG	8	15
MPI_FLOAT	4	16
MPI_DOUBLE	8	17
MPI_LONG_DOUBLE	16	18
MPI_C_BOOL	1	19
MPI_INT8_T	1	20
MPI_INT16_T	2	21
MPI_INT32_T	4	22
MPI_INT64_T	8	23
MPI_UINT8_T	1	24
MPI_UINT16_T	2	25
MPI_UINT32_T	4	26
MPI_UINT64_T	8	27
MPI_AINT	8	28
MPI_COUNT	8	29
MPI_OFFSET	8	30
MPI_C_COMPLEX	2*4	31
MPI_C_FLOAT_COMPLEX	2*4	32
MPI_C_DOUBLE_COMPLEX	2*8	33
MPI_C_LONG_DOUBLE_COMPLEX	2*16	34
MPI_CHARACTER	1	35
MPI_LOGICAL	4	36
MPI_INTEGER	4	37
MPI_REAL	4	38
MPI_DOUBLE_PRECISION	8	39
MPI_COMPLEX	2*4	40
MPI_DOUBLE_COMPLEX	2*8	41
MPI_CXX_BOOL	1	42
MPI_CXX_BOOL MPI_CXX_FLOAT_COMPLEX	$2^{*}4$	43
MPI_CXX_PEOAT_COMPLEX MPI_CXX_DOUBLE_COMPLEX	$2^{4}$ $2^{*8}$	44
MPI_CXX_DOUBLE_COMPLEX MPI_CXX_LONG_DOUBLE_COMPLEX	$2^{*}8$ $2^{*}16$	45
	2 10	46
		47

Table 14.2: "external32" sizes of predefined datatypes

**Unofficial Draft for Comment Only** 

1	Predefined Type	Length
2	MPI_INTEGER1	1
3	MPI_INTEGER2	2
4	MPI_INTEGER4	4
5	MPI_INTEGER8	8
6	MPI_INTEGER16	16
7	MPI_REAL2	2
8	MPI_REAL4	4
9	MPI_REAL8	8
10	MPI_REAL16	16
11	MPI_COMPLEX4	2*2
12	MPI_COMPLEX8	2*4
13	MPI_COMPLEX16	2*8
14	MPI_COMPLEX32	2*16
15		
16		
17	Table 14.3: "external $32$ " sizes of option	al datatypes
18		
19		
20		
21		
22		
23		
24		
25		
26		·
27		
28		
29		
30		
31		
32		
33		T (1
34	C++ Types	Length
35	MPI_CXX_BOOL	1
36	MPI_CXX_FLOAT_COMPLEX	$2^{*4}$
37	MPI_CXX_DOUBLE_COMPLEX	$2^{*8}$
38	MPI_CXX_LONG_DOUBLE_COMPLEX	X 2*16
39		
40	Table 14.4: "external32" sizes of $C++$	- datatypes
41		and of F and
42		
43		
44		
45		
46		
47		
48		

14.5.3	User-Defined Data Represer	itations	1
There a	re two situations that cannot	be handled by the required representations:	2 3
1. a	user wants to write a file in a	representation unknown to the implementation, and	4
		n in a representation unknown to the implementation.	5 6
			7
	er-defined data representation stream to do the data repres	s allow the user to insert a third party converter into	8
the I/O	stream to do the data repres		9 10
	CICTED DATADED	read_conversion_fn, write_conversion_fn,	11
	dtype_file_extent_fn,		12
IN	datarep	data representation identifier (string)	13
IN	read_conversion_fn	function invoked to convert from file representation	14 15
		to native representation (function)	16
IN	write_conversion_fn	function invoked to convert from native	17
		representation to file representation (function)	18 19
IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as	20
		represented in the file (function)	21
IN	extra_state	extra state	22 23
C bind	ling		24
	I_Register_datarep(const	char *datarep,	25
		sion_function *read_conversion_fn,	26
	MPI_Datarep_conver	rsion_function *write_conversion_fn,	27
	MPI_Datarep_extent	_function *dtype_file_extent_fn,	28
	void *extra_state)		29
int MP	I_Register_datarep_c(cons	t char *datarep.	30
		rsion_function_c *read_conversion_fn,	31 32
	_	sion_function_c *write_conversion_fn,	33
	-		34
	void *extra_state)	• -	35
The state	2002 1 1 1		36
	n 2008 binding		37
MPI_Re		ead_conversion_fn, write_conversion_fn,	38
CU		fn, extra_state, ierror)	39
	ARACTER(LEN=*), INTENT(IN	rsion_function), INTENT(IN) ::	40
PR	-	n, write_conversion_fn	41
PRI		t_function) :: dtype_file_extent_fn	42
	-	ND), INTENT(IN) :: extra_state	43
	TEGER, OPTIONAL, INTENT(O		44
		<pre>read_conversion_fn, write_conversion_fn,</pre>	45
IN T TIG		fn, extra_state, ierror)	46
CH	ARACTER(LEN=*), INTENT(IN		47 48
	-	-	-

```
1
          PROCEDURE(MPI_Datarep_conversion_function_c), INTENT(IN) ::
\mathbf{2}
                     read_conversion_fn, write_conversion_fn
3
          PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
4
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
5
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     Fortran binding
7
     MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
8
                    DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
9
          CHARACTER*(*) DATAREP
10
          EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
11
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
12
          INTEGER IERROR
13
14
         The call associates read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn
15
     with the data representation identifier datarep. datarep can then be used as an argument
16
     to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conver-
17
     sion functions to convert all data items accessed between file data representation and na-
18
     tive representation. MPI_REGISTER_DATAREP is a local operation and only registers the
19
     data representation for the calling MPI process. If datarep is already defined, an error
20
     in the error class MPI_ERR_DUP_DATAREP is raised using the default file error handler
21
     (see Section 14.7). The length of a data representation string is limited to the value of
22
     MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64.
23
     No routines are provided to delete data representations and free the associated resources;
^{24}
     it is not expected that an application will generate them in significant numbers.
25
26
     Extent Callback
27
     typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
28
                    MPI_Aint *extent, void *extra_state);
29
30
     ABSTRACT INTERFACE
31
       SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
32
                     ierror)
33
          TYPE(MPI_Datatype) :: datatype
34
          INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
35
          INTEGER :: ierror
36
     SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
37
          INTEGER DATATYPE, IERROR
38
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
39
40
          The function dtype_file_extent_fn must return, in file_extent, the number of bytes re-
41
     quired to store datatype in the file representation. The function is passed, in extra_state,
42
     the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call
43
     this routine with predefined datatypes employed by the user.
44
45
           Rationale. This callback does not have a large count variant because it is anticipated
46
           that large counts will not be required to represent the extent output value. (End of
47
           rationale.)
48
```

14.5. FILE INTEROPERABILITY 693	
\mpicallbackmain{MPI\_Datarep\_conversion\_function}MPI\_Datarep\_conversion	n\_function\_c}
MPI_DATAREP_CONVERSION_FUNCTION also supports large count types in separate	<sup>1</sup> #-error!
additional MPI procedures in C (suffixed with the "_c") and interface polymorphism in	<sup>2</sup> #-other
Fortran when using USE mpi_f08.	This is a callback
If the extent cannot be represented in extent, the callback function shall set extent to	prototype, use
MPI_UNDEFINED. The MPI implementation will then raise an error of class	correct macro!
MPI_ERR_VALUE_TOO_LARGE.	6 Done
	7 in
Datarep Conversion Functions	8 PR449
typedef int MPI_Datarep_conversion_function(void *userbuf,	9
MPI_Datatype datatype, int count, void *filebuf,	10
MPI_Offset position, void *extra_state);	11
In i_diiset position, void #extid_state),	12
<pre>typedef int MPI_Datarep_conversion_function_c(void *userbuf,</pre>	13
MPI_Datatype datatype, MPI_Count count, void *filebuf,	14
<pre>MPI_Offset position, void *extra_state);</pre>	15
ABSTRACT INTERFACE	16
SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,	17
filebuf, position, extra_state, ierror)	18 19
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	
TYPE(C_PTR), VALUE :: userbuf, filebuf	20
TYPE(MPI_Datatype) :: datatype	21 22
INTEGER :: count, ierror	22
INTEGER(KIND=MPI_OFFSET_KIND) :: position	23
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	24
	26
ABSTRACT INTERFACE	20
SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count,	28
filebuf, position, extra_state, ierror)	29
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	30
TYPE(C_PTR), VALUE :: userbuf, filebuf	31
TYPE(MPI_Datatype) :: datatype	32
INTEGER(KIND=MPI_COUNT_KIND) :: count	33
INTEGER(KIND=MPI_OFFSET_KIND) :: position	34
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	35
INTEGER :: ierror	36
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,	37
POSITION, EXTRA_STATE, IERROR)	38
<type> USERBUF(*), FILEBUF(*)</type>	39
INTEGER DATATYPE, COUNT, IERROR	40
INTEGER(KIND=MPI_OFFSET_KIND) POSITION	41
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	42
	43
The function read_conversion_fn must convert from file data representation to na-	

The function read\_conversion\_fn must convert from file data representation to native representation. Before calling this routine, MPI allocates and fills filebuf with 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 is passed, in extra\_state, the argument that was passed to the MPI\_REGISTER\_DATAREP call. The function must

44

45

46

47

copy all count data items from filebuf to userbuf in the distribution described by datatype,
 converting each data item from file representation to native representation. datatype will be
 equivalent to the datatype that the user passed to the read function. If the size of datatype
 is less than the size of the count data items, the conversion function must treat datatype
 as being contiguously tiled over the userbuf. The conversion function must begin storing
 converted data at the location in userbuf specified by position into the (tiled) datatype.

Advice to users. Although the conversion functions have similarities to MPI\_PACK and MPI\_UNPACK, one should note the differences in the use of the arguments count and position. In the conversion functions, count is a count of data items (i.e., count of typemap entries of datatype), and position is an index into this typemap. In MPI\_PACK, incount refers to the number of whole datatypes, and position is a number of bytes. (*End of advice to users.*)

- Advice to implementors. A converted read operation could be implemented as follows:
  - 1. Get file extent of all data items
  - 2. Allocate a filebuf large enough to hold all count data items
  - 3. Read data from file into filebuf
    - 4. Call read\_conversion\_fn to convert data and place it into userbuf
  - 5. Deallocate filebuf
  - (End of advice to implementors.)

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call, and the userbuf pointer will be unchanged.

Rationale. Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion function can use the **position** to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

The function write\_conversion\_fn must convert from native representation to file data
 representation. Before calling this routine, MPI allocates filebuf of a size large enough to
 hold count contiguous data items. The type of each data item matches the corresponding

### Unofficial Draft for Comment Only

 $^{31}$ 

entry for the predefined datatype in the type signature of datatype. The function must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the write function. The function is passed, in extra\_state, the argument that was passed to the MPI\_REGISTER\_DATAREP call.

The predefined constant MPI\_CONVERSION\_FN\_NULL may be used as either write\_conversion\_fn or read\_conversion\_fn. In that case, MPI will not attempt to invoke write\_conversion\_fn or read\_conversion\_fn, respectively, but will perform the requested data access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by using a filebuf large enough to hold all the requested data items or else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section (read\_conversion\_fn, write\_conversion\_fn, and dtype\_file\_extent\_fn) when one of the read or write routines in Section 14.4, 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.

The conversion functions must be reentrant. User defined data representations are restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion functions to call any collective routines or to free datatype.

The conversion functions should return an error code. If the returned error code has a value other than MPI\_SUCCESS, the implementation will raise an error in the class MPI\_ERR\_CONVERSION.

#### 14.5.4 Matching Data Representations

It is the user's responsibility to ensure that the data representation used to read data from a file is *compatible* with the data representation that was used to write that data to the file.

In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible. Similarly, using different representation names on two different implementations may yield compatible representations.

Compatibility can be obtained when "external32" representation is used, although precision may be lost and the performance may be less than when "native" representation is used. Compatibility is guaranteed using "external32" provided at least one of the following conditions is met.

- The data access routines directly use types enumerated in Section 14.5.2, 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 19.1.9).

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

 $45 \\ 46$ 

• 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 19.1.9 defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

# 14.6 Consistency and Semantics

14.6.1 File Consistency

14Consistency semantics define the outcome of multiple accesses to a single file. All file 15accesses in MPI are relative to a specific file handle created from a collective open. MPI 16provides three levels of consistency: sequential consistency among all accesses using a single 17file handle, sequential consistency among all accesses using file handles created from a single 18 collective open with atomic mode enabled, and user-imposed consistency among accesses 19 other than the above. Sequential consistency means the behavior of a set of operations will 20be as if the operations were performed in some serial order consistent with program order; 21each access appears atomic, although the exact ordering of accesses is unspecified. User-22 imposed consistency may be obtained using program order and calls to MPI\_FILE\_SYNC. 23

Let  $FH_1$  be the set of file handles created from one particular collective open of the  $^{24}$ file FOO, and  $FH_2$  be the set of file handles created from a different collective open of 25FOO. Note that nothing restrictive is said about  $FH_1$  and  $FH_2$ : the sizes of  $FH_1$  and 26 $FH_2$  may be different, the groups of processes used for each open may or may not intersect, 27the file handles in  $FH_1$  may be destroyed before those in  $FH_2$  are created, etc. Consider 28the following three cases: a single file handle (e.g.,  $fh_1 \in FH_1$ ), two file handles created 29from a single collective open (e.g.,  $fh_{1a} \in FH_1$  and  $fh_{1b} \in FH_1$ ), and two file handles from 30 different collective opens (e.g.,  $fh_1 \in FH_1$  and  $fh_2 \in FH_2$ ).  $^{31}$ 

For the purpose of consistency semantics, a matched pair (Section 14.4.5) 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 operations.

Advice to users. For an MPI\_FILE\_IREAD and MPI\_WAIT pair, the operation begins when MPI\_FILE\_IREAD is called and ends when MPI\_WAIT returns. (*End of advice to users.*)

<sup>43</sup> Assume that  $A_1$  and  $A_2$  are two data access operations. Let  $D_1$  ( $D_2$ ) be the set of <sup>44</sup> absolute byte displacements of every byte accessed in  $A_1$  ( $A_2$ ). The two data accesses <sup>45</sup> overlap if  $D_1 \cap D_2 \neq \emptyset$ . The two data accesses conflict if they overlap and at least one is a <sup>46</sup> write access.

<sup>47</sup> Let  $SEQ_{fh}$  be a sequence of file operations on a single file handle, bracketed by <sup>48</sup> MPI\_FILE\_SYNCs on that file handle. (Both opening and closing a file implicitly perform

### Unofficial Draft for Comment Only

1

2

3 4

5 6

7

8

9 10 11

12

13

38 39

40

41

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.

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.

Case 3:  $fh_1 \in FH_1$  and  $fh_2 \in FH_2$  Consider access to a single file using file handles from distinct collective opens. In order to guarantee sequential consistency, MPI\_FILE\_SYNC must be used (both opening and closing a file implicitly perform an MPI\_FILE\_SYNC).

Sequential consistency is guaranteed among accesses to a single file if for any write sequence  $SEQ_1$  to the file, there is no sequence  $SEQ_2$  to the file which is *concurrent* with  $SEQ_1$ . To guarantee sequential consistency when there are write sequences, MPI\_FILE\_SYNC must be used together with a mechanism that guarantees nonconcurrency of the sequences.

See the examples in Section 14.6.11 for further clarification of some of these consistency semantics.

MPI_FILE_SET_ATOMICITY(fh, flag)		34
INOUT fh	file handle (handle)	35
	ine numere (numere)	36
IN flag	true to set atomic mode, $false$ to set nonatomic mode	37
	(logical)	38
		39
C binding		
int MPI_File_set_atomicity(MPI_File fh, int flag)		
Eastern 2008 bin din m		
Fortran 2008 binding		
MPI_File_set_atomicity(fh, flag, ierror)		
TYPE(MPI_File), INTENT(IN) ::	fh	45
LOGICAL, INTENT(IN) :: flag		
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror	47
Fortran hinding		

#### Fortran binding

Unofficial Draft for Comment Only

 $\mathbf{2}$ 

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 2	MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR			
3	LOGICAL FLAG			
4	Let $FH$ be the set of file handles created by one collective open. The consistency			
5 6	semantics for data access operations using $FH$ is set by collectively calling			
7	$MPI\_FILE\_SET\_ATOMICITY \ \mathrm{on} \ FH. \ MPI\_FILE\_SET\_ATOMICITY \ \mathrm{is} \ \mathrm{collective}; \ \mathrm{all} \ \mathrm{pro-}$			
8	cesses in the group must pass identical values for fh and flag. If flag is true, atomic mode is			
9	set; if flag is false, nonatomic mode is set. Changing the consistency semantics for an open file only affects new data accesses.			
10	All completed data accesses are guaranteed to abide by the consistency semantics in effect			
11 12	during their execution. Nonblocking data accesses and split collective operations that have			
12	not completed (e.g., via MPI_WAIT) are only guaranteed to abide by nonatomic mode			
14	consistency semantics.			
15	Advice to implementors. Since the semantics guaranteed by atomic mode are stronger			
16	than those guaranteed by nonatomic mode, an implementation is free to adhere to			
17	the more stringent atomic mode semantics for outstanding requests. (End of advice			
18 19	to implementors.)			
20				
21				
22	MPI_FILE_GET_ATOMICITY(fh, flag)			
23 24	IN fh file handle (handle)			
25	OUTflagtrue if atomic mode, false if nonatomic mode (logical)			
26				
27	C binding			
28 29	<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>			
30	Fortran 2008 binding			
31				
	MPI_File_get_atomicity(fh, flag, ierror)			
32	TYPE(MPI_File), INTENT(IN) :: fh			
33				
	TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
33 34	TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag			
33 34 35	<pre>TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR</pre>			
33 34 35 36 37 38	TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)			
33 34 35 36 37 38 39	<pre>TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR</pre>			
33 34 35 36 37 38	<pre>TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic</pre>			
33 34 35 36 37 38 39 40	TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access			
<ol> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> </ol>	<pre>TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic</pre>			
<ol> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> </ol>	<pre>TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic</pre>			
<ol> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> </ol>	<pre>TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic mode is enabled; if flag is false, nonatomic mode is enabled.</pre>			
<ol> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> </ol>	<pre>TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic mode is enabled; if flag is false, nonatomic mode is enabled. MPI_FILE_SYNC(fh)</pre>			

int MPI\_File\_sync(MPI\_File fh)

#### Fortran 2008 binding

MPI\_File\_sync(fh, ierror)
 TYPE(MPI\_File), INTENT(IN) :: fh
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_FILE\_SYNC(FH, IERROR) INTEGER FH, IERROR

Calling MPI\_FILE\_SYNC with fh causes all previous writes to fh by the calling process to be transferred to the storage device. If other processes have made updates to the storage device, then all such updates become visible to subsequent reads of fh by the calling process. MPI\_FILE\_SYNC may be necessary to ensure sequential consistency in certain cases (see above).

MPI\_FILE\_SYNC is a collective operation.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI\_FILE\_SYNC—otherwise, the call to MPI\_FILE\_SYNC is erroneous.

#### 14.6.2 Random Access vs. Sequential Files

MPI distinguishes ordinary random access files from sequential stream files, such as pipes and tape files. Sequential stream files must be opened with the MPI\_MODE\_SEQUENTIAL flag set in the amode. For these files, the only permitted data access operations are shared file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the notion of file pointer is not meaningful; therefore, calls to MPI\_FILE\_SEEK\_SHARED and MPI\_FILE\_GET\_POSITION\_SHARED are erroneous, and the pointer update rules specified for the data access routines do not apply. The amount of data accessed by a data access operation will be the amount requested unless the end of file is reached or an error is raised.

*Rationale.* This implies that reading on a pipe will always wait until the requested amount of data is available or until the process writing to the pipe has issued an end of file. (*End of rationale.*)

Finally, for some sequential files, such as those corresponding to magnetic tapes or streaming network connections, writes to the file may be destructive. In other words, a write may act as a truncate (a MPI\_FILE\_SET\_SIZE with size set to the current position) followed by the write.

#### 14.6.3 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

#### Unofficial Draft for Comment Only

1

2

6

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

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.

14.6.4 Collective File Operations

<sup>9</sup> Collective file operations are subject to the same restrictions as collective communication
 <sup>11</sup> operations. For a complete discussion, please refer to the semantics set forth in Section 6.14.

Collective file operations are collective over a duplicate of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

16

7 8

- 17
- 14.6.5 Nonblocking Collective File Operations

<sup>18</sup> Nonblocking collective file operations are defined only for data access routines with explicit
 <sup>20</sup> offsets and individual file pointers but not with shared file pointers.

Nonblocking collective file operations are subject to the same restrictions as blocking collective I/O operations. All processes belonging to the group of the communicator that was used to open the file must call collective I/O operations (blocking and nonblocking) in the same order. This is consistent with the ordering rules for collective operations in threaded environments. For a complete discussion, please refer to the semantics set forth in Section 6.14.

Nonblocking collective I/O operations do not match with blocking collective I/O operations. 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 6.12.

37 38

39

14.6.6 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.

45 46

47

48

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.*)

#### 14.6.7 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.

#### 14.6.8 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 19.3).

#### 14.6.9 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information may be necessary to optimize file layout. This information can be provided as *hints* specified via info when a file is created (see Section 14.2.8).

#### 14.6.10 File Size

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI *size changing* routines, such as MPI\_FILE\_SET\_SIZE. A call to a size changing routine does not necessarily change the file size. For example, calling MPI\_FILE\_PREALLOCATE with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since MPI\_FILE\_OPEN if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

• One plus the displacement of the high byte.

#### Unofficial Draft for Comment Only

 $^{24}$ 

 $46 \\ 47$ 

	702 CHAPTER 14. 1/O
1 2	• The size immediately after the size changing routine, or MPI_FILE_OPEN, returned.
3 4	When applying consistency semantics, calls to MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and
5 6 7	MPI_FILE_GET_SIZE is considered a read of the file (which overlaps with all accesses to the file).
8 9 10 11 12	Advice to users. Any sequence of operations containing the collective routines MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 14.6.1 are satisfied. ( <i>End of advice to users.</i> )
13 14 15	File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.
16 17 18 19 20 21	Advice to users. Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an MPI_FILE_READ of 10 bytes and an MPI_FILE_SET_SIZE to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. ( <i>End of advice to</i>
22 23 24	users.) 14.6.11 Examples
25 26 27	The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address
28	$\bullet$ conflicting accesses on file handles obtained from a single collective open, and
29 30	• all accesses on file handles obtained from two separate collective opens.
31 32 33 34	The simplest way to achieve consistency for conflicting accesses is to obtain sequential consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 integers. If the latter, every element of <b>b</b> will be 5. If nonatomic mode is set, the results of the read are undefined.
35 36 27	/* Process 0 */ int i, a[10];
37 38	int TRUE = 1;
39 40 41	<pre>for (i=0;i&lt;10;i++)     a[i] = 5;</pre>
42 43 44 45 46 47	<pre>MPI_File_open(MPI_COMM_WORLD, "workfile",</pre>
47 48	/* MPI_Barrier(MPI_COMM_WORLD); */

```
1
/* Process 1 */
                                                                                         2
int b[10];
int TRUE = 1;
MPI_File_open(MPI_COMM_WORLD, "workfile",
                MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_set_atomicity(fh1, TRUE);
/* MPI_Barrier(MPI_COMM_WORLD); */
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
                                                                                         10
A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
                                                                                         11
temporal order with, for example, calls to MPI_BARRIER.
                                                                                         12
                                                                                         13
     Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
                                                                                         14
     order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
                                                                                         15
     received by process 1 using MPI_RECV. (End of advice to users.)
                                                                                         16
                                                                                         17
    Alternatively, a user can impose consistency with nonatomic mode set:
                                                                                         18
/* Process 0 */
                                                                                         19
int i, a[10];
                                                                                         20
for (i=0;i<10;i++)
                                                                                         21
   a[i] = 5;
                                                                                         22
                                                                                         23
MPI_File_open(MPI_COMM_WORLD, "workfile",
                                                                                         ^{24}
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
                                                                                         25
MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                         26
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
                                                                                         27
MPI_File_sync(fh0);
                                                                                         28
MPI_Barrier(MPI_COMM_WORLD);
                                                                                         29
MPI_File_sync(fh0);
                                                                                         30
                                                                                         31
/* Process 1 */
                                                                                         32
int b[10];
                                                                                         33
MPI_File_open(MPI_COMM_WORLD, "workfile",
                                                                                         34
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
                                                                                         35
MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                         36
MPI_File_sync(fh1);
                                                                                         37
MPI_Barrier(MPI_COMM_WORLD);
                                                                                         38
MPI_File_sync(fh1);
                                                                                         39
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
                                                                                         40
                                                                                         41
The "sync-barrier-sync" construct is required because:
                                                                                         42
   • The barrier ensures that the write on process 0 occurs before the read on process 1.
                                                                                         43
                                                                                         44
   • The first sync guarantees that the data written by all processes is transferred to the
                                                                                         45
     storage device.
                                                                                         46
                                                                                         47
   • The second sync guarantees that all data which has been transferred to the storage
                                                                                         48
     device is visible to all processes. (This does not affect process 0 in this example.)
```

```
1
         The following program represents an erroneous attempt to achieve consistency by elim-
\mathbf{2}
     inating the apparently superfluous second "sync" call for each process.
3
     /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
4
     /* Process 0 */
5
     int i, a[10];
6
     for (i=0;i<10;i++)</pre>
7
        a[i] = 5;
8
9
     MPI_File_open(MPI_COMM_WORLD, "workfile",
10
                    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fhO);
11
     MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
12
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
13
     MPI_File_sync(fh0);
14
     MPI_Barrier(MPI_COMM_WORLD);
15
16
     /* Process 1 */
17
     int b[10];
18
     MPI_File_open(MPI_COMM_WORLD, "workfile",
19
                    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
20
     MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
21
     MPI_Barrier(MPI_COMM_WORLD);
22
     MPI_File_sync(fh1);
23
     MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
24
25
     /* ----- THIS EXAMPLE IS ERRONEOUS ------ */
26
27
     The above program also violates the MPI rule against out-of-order collective operations and
28
     will deadlock for implementations in which MPI_FILE_SYNC blocks.
29
30
          Advice to users. Some implementations may choose to implement MPI_FILE_SYNC
31
          as a temporally synchronizing function. When using such an implementation, the
32
           "sync-barrier-sync" construct above can be replaced by a single "sync." The results of
33
          using such code with an implementation for which MPI_FILE_SYNC is not temporally
34
          synchronizing is undefined. (End of advice to users.)
35
36
     Asynchronous I/O
37
38
     The behavior of asynchronous I/O operations is determined by applying the rules specified
39
     above for synchronous I/O operations.
         The following examples all access a preexisting file "myfile." Word 10 in myfile initially
40
41
     contains the integer 2. Each example writes and reads word 10.
42
         First consider the following code fragment:
43
     int a = 4, b, TRUE=1;
44
     MPI_File_open(MPI_COMM_WORLD, "myfile",
45
                    MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
46
     MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
47
     /* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */
48
```

MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);	1 2
<pre>MPI_File_iread_at(fh, 10, &amp;b, 1, MPI_INT, &amp;reqs[1]); MPI_Waitall(2, reqs, statuses);</pre>	3
rri_waitaii(2, ieqs, statuses),	4
For asynchronous data access operations, $MPI$ specifies that the access occurs at any time	5
between the call to the asynchronous data access routine and the return from the corre-	6
sponding request complete routine. Thus, executing either the read before the write, or the	7
write before the read is consistent with program order. If atomic mode is set, then MPI	8
guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic	9
mode is not set, then sequential consistency is not guaranteed and the program may read	10
something other than 2 or 4 due to the conflicting data access.	11
Similarly, the following code fragment does not order file accesses:	12
int a = 4, b;	13
MPI_File_open(MPI_COMM_WORLD, "myfile",	14
<pre>MPI_MODE_RDWR, MPI_INFO_NULL, &amp;fh);</pre>	15 16
<pre>MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);</pre>	10
/* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */	18
<pre>MPI_File_iwrite_at(fh, 10, &amp;a, 1, MPI_INT, &amp;reqs[0]);</pre>	19
<pre>MPI_File_iread_at(fh, 10, &amp;b, 1, MPI_INT, &amp;reqs[1]);</pre>	20
<pre>MPI_Wait(&amp;reqs[0], &amp;status); MDL_Uait(&amp;maga[1], &amp;status);</pre>	21
<pre>MPI_Wait(&amp;reqs[1], &amp;status);</pre>	22
If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee	23
sequential consistency in nonatomic mode.	24
On the other hand, the following code fragment:	25
int a = 4, b;	26
MPI_File_open(MPI_COMM_WORLD, "myfile",	27
MPI_MODE_RDWR, MPI_INFO_NULL, &fh);	28 29
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);	29 30
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);	31
MPI_Wait(&reqs[0], &status);	32
<pre>MPI_File_iread_at(fh, 10, &amp;b, 1, MPI_INT, &amp;reqs[1]);</pre>	33
<pre>MPI_Wait(&amp;reqs[1], &amp;status);</pre>	34
defines the same ordering as:	35
dennes the same ordering as.	36
int a = 4, b;	37
<pre>MPI_File_open(MPI_COMM_WORLD, "myfile",</pre>	38
MPI_MODE_RDWR, MPI_INFO_NULL, &fh);	39
<pre>MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);</pre>	40
MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status );	41
<pre>MPI_File_read_at(fh, 10, &amp;b, 1, MPI_INT, &amp;status );</pre>	42 43
Since	43
	45
• nonconcurrent operations on a single file handle are sequentially consistent, and	46
• the program fragments specify an order for the operations,	47
	48

1

 $^{24}$ 

25 26

27 28

29

30

 $^{31}$ 

32 33

34

35

36

37

```
\mathbf{2}
     to set atomic mode for this example.
3
          Similar considerations apply to conflicting accesses of the form:
4
     MPI_File_iwrite_all(fh,...);
5
     MPI_File_iread_all(fh,...);
6
     MPI_Waitall(...);
7
8
          In addition, as mentioned in Section 14.6.5, nonblocking collective I/O operations have
9
     to be called in the same order on the file handle by all processes.
10
          Similar considerations apply to conflicting accesses of the form:
11
12
     MPI_File_write_all_begin(fh,...);
13
     MPI_File_iread(fh,...);
14
     MPI_Wait(fh,...);
15
     MPI_File_write_all_end(fh,...);
16
          Recall that constraints governing consistency and semantics are not relevant to the
17
     following:
18
19
     MPI_File_write_all_begin(fh,...);
20
     MPI_File_read_all_begin(fh,...);
21
```

MPI guarantees that both program fragments will read the value 4 into b. There is no need

```
MPI_File_read_all_end(fh,...);
```

```
22
23 MPI_File_write_all_end(fh,...);
```

since split collective operations on the same file handle may not overlap (see Section 14.4.5).

# 14.7 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.*)

<sup>38</sup> Like communicators, each file handle has an error handler associated with it. The MPI
 <sup>39</sup> I/O error handling routines are defined in Section 9.3.

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

file 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 when using the World Model. 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.*)

# 14.8 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 14.5.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI\_ERR\_TYPE.

		20
MPI_ERR_FILE	Invalid file handle	21
MPI_ERR_NOT_SAME	Collective argument not identical on all	22
	processes, or collective routines called in	23
	a different order by different processes	20 24
MPI_ERR_AMODE	Error related to the amode passed to	24
	MPI_FILE_OPEN	
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to	26
	MPI_FILE_SET_VIEW	27
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	28
	a file which supports sequential access only	29
MPI_ERR_NO_SUCH_FILE	File does not exist	30
MPI_ERR_FILE_EXISTS	File exists	31
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	32
MPI_ERR_ACCESS	Permission denied	33
MPI_ERR_NO_SPACE	Not enough space	34
MPI_ERR_QUOTA	Quota exceeded	35
MPI_ERR_READ_ONLY	Read-only file or file system	36
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	37
	the file is currently open by some process	38
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	39
	tered because a data representation identi-	40
	fier that was already defined was passed to	41
	MPI_REGISTER_DATAREP	42
MPI_ERR_CONVERSION	An error occurred in a user supplied data	43
	conversion function.	44
MPI_ERR_IO	Other I/O error	45
	Other 1/O error	46
Table 14.5	: I/O Error Classes	47

 $\mathbf{2}$ 

```
14.9 Examples
1
\mathbf{2}
    14.9.1 Double Buffering with Split Collective I/O
3
4
     This example shows how to overlap computation and output. The computation is performed
5
     by the function compute_buffer().
6
7
     8
      *
9
                            double_buffer
      * Function:
10
      *
^{11}
      * Synopsis:
12
            void double_buffer(
      *
13
                    MPI_File fh,
                                                             ** IN
      *
14
      *
                    MPI_Datatype buftype,
                                                             ** IN
15
                    int bufcount
      *
                                                             ** IN
16
            )
      *
17
      *
18
      * Description:
19
            Performs the steps to overlap computation with a collective write
      *
20
            by using a double-buffering technique.
      *
21
      *
22
      * Parameters:
23
                               previously opened MPI file handle
      *
            fh
^{24}
                               MPI datatype for memory layout
            buftype
      *
25
                                (Assumes a compatible view has been set on fh)
26
                                # buftype elements to transfer
            bufcount
27
                                              -----*/
28
29
     /* this macro switches which buffer "x" is pointing to */
    #define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
30
31
32
     void double_buffer(MPI_File fh, MPI_Datatype buftype, int bufcount)
33
     {
34
35
       MPI_Status status;
                                 /* status for MPI calls */
36
       float *buffer1, *buffer2; /* buffers to hold results */
37
       float *compute_buf_ptr;
                                  /* destination buffer */
38
                                  /* for computing */
39
       float *write_buf_ptr;
                                  /* source for writing */
40
                                  /* determines when to quit */
       int done;
41
42
       /* buffer initialization */
43
       buffer1 = (float *)
44
                          malloc(bufcount*sizeof(float));
45
       buffer2 = (float *)
46
                          malloc(bufcount*sizeof(float));
47
       compute_buf_ptr = buffer1; /* initially point to buffer1 */
48
                                     /* initially point to buffer1 */
       write_buf_ptr = buffer1;
```

}

```
2
  /* DOUBLE-BUFFER prolog:
        compute buffer1; then initiate writing buffer1 to disk
    */
   compute_buffer(compute_buf_ptr, bufcount, &done);
  MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
   /* DOUBLE-BUFFER steady state:
                                                                                    10
       Overlap writing old results from buffer pointed to by write_buf_ptr
    *
                                                                                    11
       with computing new results into buffer pointed to by compute_buf_ptr.
    *
                                                                                    12
       There is always one write-buffer and one compute-buffer in use
                                                                                    13
                                                                                    14
    *
       during steady state.
                                                                                    15
    */
                                                                                    16
  while (!done) {
                                                                                    17
      TOGGLE_PTR(compute_buf_ptr);
                                                                                    18
      compute_buffer(compute_buf_ptr, bufcount, &done);
                                                                                    19
      MPI_File_write_all_end(fh, write_buf_ptr, &status);
      TOGGLE_PTR(write_buf_ptr);
                                                                                    20
                                                                                    21
      MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
  }
                                                                                    22
                                                                                    23
                                                                                    24
   /* DOUBLE-BUFFER epilog:
                                                                                    25
    *
        wait for final write to complete.
                                                                                    26
    */
  MPI_File_write_all_end(fh, write_buf_ptr, &status);
                                                                                    27
                                                                                    28
                                                                                    29
  /* buffer cleanup */
                                                                                    30
                                                                                    31
  free(buffer1);
                                                                                    32
  free(buffer2);
                                                                                    33
                                                                                    34
                                                                                    35
14.9.2 Subarray Filetype Constructor
                                                                                    36
```

Assume we are writing out a  $100 \times 100$  2D array of double precision floating point numbers that is distributed among 4 processes such that each process has a block of 25 columns (e.g., process 0 has columns 0–24, process 1 has columns 25–49, etc.; see Figure 14.4). To create the filetypes for each process one could use the following C program (see Section 5.1.3):

```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0]=100; sizes[1]=100;
```

**Unofficial Draft for Comment Only** 

1

37

38

39

40 41

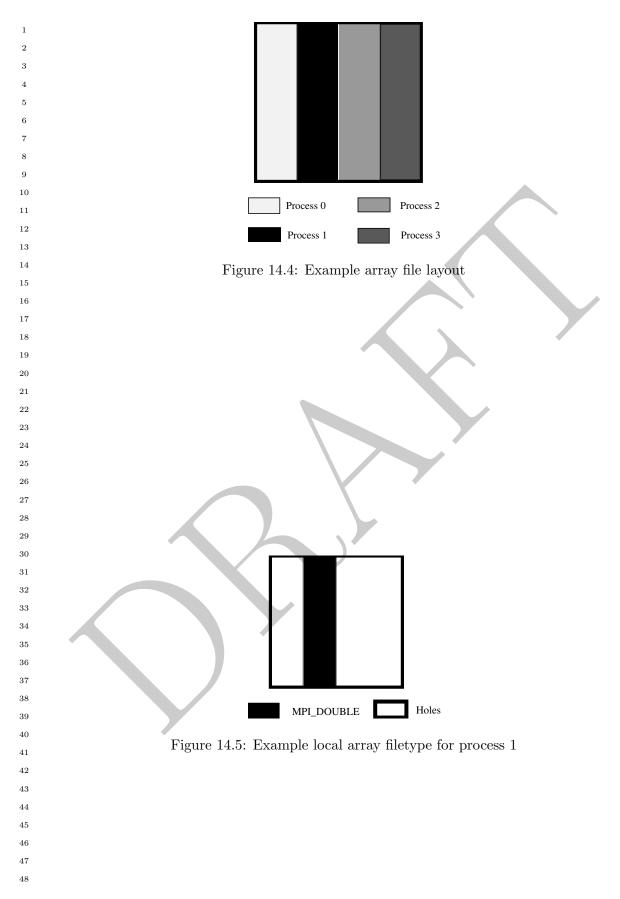
42

43

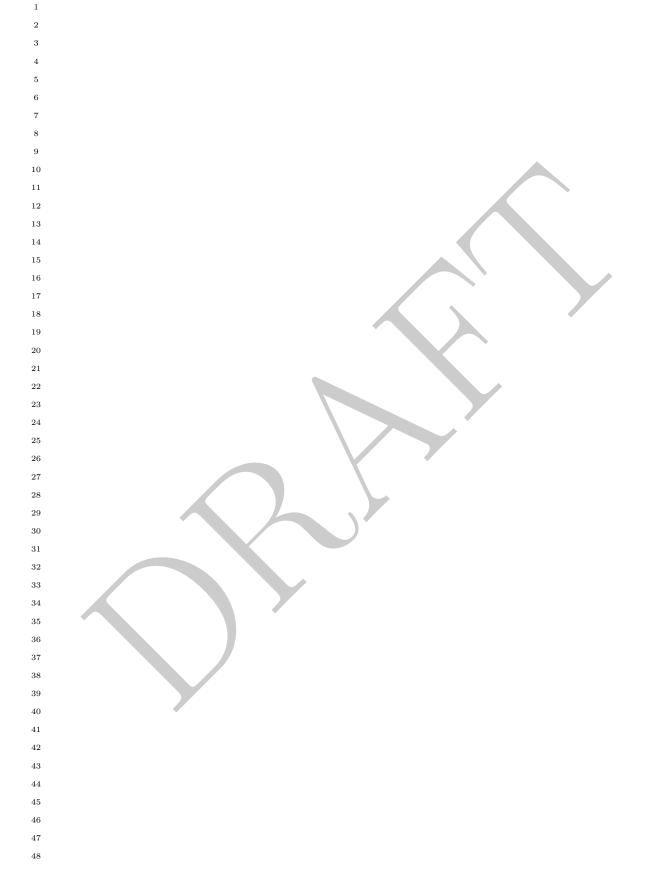
44

4546

47



```
subsizes[0]=100; subsizes[1]=25;
                                                                                             1
                                                                                             \mathbf{2}
   starts[0]=0; starts[1]=rank*subsizes[1];
                                                                                             3
   MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                                                                                             4
                                 MPI_DOUBLE, &filetype);
                                                                                             5
                                                                                             6
    Or, equivalently in Fortran:
                                                                                             7
                                                                                             8
double precision subarray(100,25)
                                                                                             9
integer filetype, rank, ierror
                                                                                             10
integer sizes(2), subsizes(2), starts(2)
                                                                                             11
                                                                                             12
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
                                                                                             13
sizes(1)
              = 100
                                                                                             14
sizes(2)
              = 100
                                                                                             15
subsizes(1) = 100
                                                                                             16
subsizes(2) = 25
                                                                                             17
starts(1)
              = 0
                                                                                             18
starts(2)
             = rank*subsizes(2)
                                                                                             19
                                                                                             20
call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
                                                                                             21
            MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
                                                                    &
                                                                                             22
            filetype, ierror)
                                                                                             23
                                                                                             ^{24}
    The generated filetype will then describe the portion of the file contained within the
                                                                                             25
process's subarray with holes for the space taken by the other processes. Figure 14.5 shows
                                                                                             26
the filetype created for process 1.
                                                                                             27
                                                                                             28
                                                                                             29
                                                                                             30
                                                                                             ^{31}
                                                                                             32
                                                                                             33
                                                                                             34
                                                                                             35
                                                                                             36
                                                                                             37
                                                                                             38
                                                                                             39
                                                                                             40
                                                                                             41
                                                                                             42
                                                                                             43
                                                                                             44
                                                                                             45
                                                                                             46
                                                                                             47
                                                                                             48
```



# Chapter 15

# Tool Support

### 15.1 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 15.2), which supports the transparent interception and inspection of MPI calls, and the MPI tool information interface (Section 15.3), which supports the inspection and manipulation of MPI control and performance variables, as well as the registration of callbacks for MPI library events. 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.

## 15.2 Profiling Interface

#### 15.2.1 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 19.1.5.

- 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 imple <sup>45</sup>
   <sup>46</sup>
   <sup>47</sup>
   <sup>47</sup>
   <sup>48</sup>

#### Unofficial Draft for Comment Only

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 13

14

1

2

3

4

5

6

7

8

9

10

5. provide a no-op routine MPI\_PCONTROL in the MPI library.

15.2.2 Discussion

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code that implements MPI on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

We believe that having such an interface is important if MPI is to be attractive to end users, since the availability of many different tools will be a significant factor in attracting users to the MPI standard.

The profiling interface is just that, an interface. It says *nothing* about the way in which it is used. There is therefore no attempt to lay down what information is collected through the interface, or how the collected information is saved, filtered, or displayed.

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as "internetworking" multiple MPI implementations. Since all that is defined is an interface, there is no objection to its being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which executable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the MPI profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this section is only present as justification and discussion of the logic for those requirements.

The examples below show one way in which an implementation could be constructed to meet the requirements on a Unix system (there are doubtless others that would be equally valid).

44

<sup>45</sup> 15.2.3 Logic of the Design

<sup>46</sup> Provided that an MPI implementation meets the requirements above, it is possible for <sup>47</sup> the implementor of the profiling system to intercept the MPI calls that are made by the user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

#### 15.2.4 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)

#### C binding

```
int MPI_Pcontrol(const int level, ...)
```

#### Fortran 2008 binding

```
MPI_Pcontrol(level)
    INTEGER, INTENT(IN) :: level
```

#### Fortran binding

MPI\_PCONTROL(LEVEL) INTEGER LEVEL

MPI libraries themselves make no use of this routine, and simply return immediately to the user code. However the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, we are unable to specify precisely the semantics that will be provided by calls to MPI\_PCONTROL. This vagueness extends to the number of arguments to the function, and their datatypes.

However to provide some level of portability of user codes to different profiling libraries, we request the following meanings for certain values of level.

- level==0 Profiling is disabled.
- level==1 Profiling is enabled at a normal default level of detail.
- level==2 Profile buffers are flushed, which may be a no-op in some profilers.
- All other values of level have profile library defined effects and additional arguments.

We also request that the default state after MPI has been initialized is for profiling to <sup>45</sup> be enabled at the normal default level. (i.e., as if MPI\_PCONTROL had just been called <sup>46</sup> with the argument 1). This allows users to link with a profiling library and to obtain profile <sup>47</sup> output without having to modify their source code at all. <sup>48</sup>

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

 $\frac{4}{5}$ 

6

7

8 9

10

11 12

13 14

15 16

17 18

19 20

21

22

23

24

25 26

27

28

29

30

31

32

33

34

35

36

37 38

39 40

41

42 43

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.

15.2.5 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:

```
<sup>10</sup> Example 15.1
```

```
11
     static int totalBytes = 0;
12
     static double totalTime = 0.0;
13
14
     int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
15
                   int dest, int tag, MPI_Comm comm)
16
     {
17
        double tstart = MPI_Wtime();
                                              /* Pass on all arguments */
18
        int size;
19
                       = PMPI_Send(buffer,count,datatype,dest,tag,comm);
        int result
20
21
        totalTime += MPI Wtime() - tstart;
                                                        /* and time
                                                                               */
22
23
        MPI_Type_size(datatype, &size);
                                            /* Compute size */
24
        totalBytes += count*size;
25
26
        return result;
27
     }
28
29
            MPI Library Implementation Example
     15.2.6
```

If the MPI library is implemented in C on a Unix system, then there are various options,
 including the two presented here, for supporting the name-shift requirement. The choice
 between these two options depends partly on whether the linker and compiler support weak
 symbols.

```
<sup>36</sup> Systems with Weak Symbols
```

<sup>37</sup>If the compiler and linker support weak external symbols (e.g., Solaris 2.x, other System V.4 machines), then only a single library is required as the following example shows:

4 5

6

7

8 9

30

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.

#### 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 15.3

```
#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.

### 15.2.7 Complications

### Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI functions (e.g., a portable implementation of the collective operations implemented using point

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4 5 6

7

8

9 10

11 12

13

14

15

16

17

18

19

20

21

22 23

24

25

26

27 28

29

30

 $^{31}$ 

32

33

34

35

36 37

38 39

40

41

42 43

44

 $45 \\ 46$ 

47

1 to point communications), there is potential for profiling functions to be called from within  $\mathbf{2}$ an MPI function that was called from a profiling function. This could lead to "double 3 counting" of the time spent in the inner routine. Since this effect could actually be useful 4 under some circumstances (e.g., it might allow one to answer the question "How much time  $\mathbf{5}$ is spent in the point to point routines when they are called from collective functions?"), we 6 have decided not to enforce any restrictions on the author of the MPI library that would  $\overline{7}$ overcome this. Therefore the author of the profiling library should be aware of this problem, 8 and guard against it. In a single-threaded world this is easily achieved through use of a 9 static variable in the profiling code that remembers if you are already inside a profiling 10 routine. It becomes more complex in a multithreaded environment (as does the meaning of 11the times recorded).

12

14

## <sup>13</sup> Linker Oddities

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 19 achieved by using wrapper functions on top of the C implementation. The author of the 20profile library then assumes that it is reasonable only to provide profile functions for the C 21binding, since Fortran will eventually call these, and the cost of the wrappers is assumed 22 to be small. However, if the wrapper functions are not in the profiling library, then none 23of the profiled entry points will be undefined when the profiling library is called. Therefore 24none of the profiling code will be included in the image. When the standard MPI library 25is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of 26the MPI functions. The overall effect is that the code will link successfully, but will not be 27profiled. 28

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**.

33

## 34 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 19.1.5.

 $40 \\ 41$ 

## 15.2.8 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

 $46 \\ 47$ 

48

- assuming a particular implementation language,
- imposing a run time cost even when no profiling was taking place.

## Unofficial Draft for Comment Only

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we decided to accept the scheme outlined above.

Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function. This capability has been demonstrated in the  $P^N$ MPI tool infrastructure [58].

## 15.3 The MPI Tool Information Interface

MPI implementations often use internal variables to control their operation and performance and rely on internal events for their implementation. Understanding and manipulating these variables and tracking these events 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, as well as expose events that can be tracked by tools. The interface is split into three 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. The third part enables tools to query available events within an MPI implementation and register callbacks for them.

To avoid restrictions on the MPI implementation, the MPI tool information interface allows the implementation to specify which control variables, performance variables, and events exist. Additionally, the user of the MPI tool information interface can obtain metadata about each available variable or event, such as its datatype, and a textual description. The MPI tool information interface provides the necessary routines to find all variables and events that exist in a particular MPI implementation; to query their properties; to retrieve descriptions about their meaning; to access and, if appropriate, to alter their values; and (in case of events) set callbacks triggered by them.

Variables, events, 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 15.3.3). 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

### 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

 $46 \\ 47$ 

<sup>1</sup> initialized and after MPI is finalized. In order to support this behavior cleanly, the MPI tool
 <sup>2</sup> information interface uses separate initialization and finalization routines. All identifiers
 <sup>3</sup> used in the MPI tool information interface have the prefix MPI\_T\_.

<sup>4</sup> On success, all MPI tool information interface routines return MPI\_SUCCESS, otherwise <sup>5</sup> they return an appropriate and unique return code indicating the reason why the call was not <sup>6</sup> successfully completed. Details on return codes can be found in Section 15.3.10. However, <sup>7</sup> unsuccessful calls to the MPI tool information interface are not fatal and do not impact the <sup>8</sup> execution of subsequent MPI routines.

<sup>9</sup> Since the MPI tool information interface primarily focuses on tools and support li-<sup>10</sup> braries, MPI implementations are only required to provide C bindings for functions and <sup>11</sup> constants introduced in this section. Except where otherwise noted, all conventions and <sup>12</sup> principles governing the C bindings of the MPI API also apply to the MPI tool information <sup>13</sup> interface, which is available by including the mpi.h header file. All routines in this interface <sup>14</sup> have local semantics.

Advice to users. The number and type of control variables, performance variables, and events 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 or of a particular event. (*End of advice to users.*)

15.3.1 Verbosity Levels

29The MPI tool information interface provides access to internal configuration and perfor-30 mance information through a set of control and performance variables defined by the MPI  $^{31}$ implementation. Since some implementations may export a large number of variables, 32 variables are classified by a verbosity level that categorizes both their intended audience 33 (end users, performance tuners or MPI implementors) and a relative measure of level of 34 detail (basic, detailed or all). These verbosity levels are described by a single integer. 35 Table 15.1 lists the constants for all possible verbosity levels. The values of the con-36 stants are monotonic in the order listed in the table; i.e., MPI\_T\_VERBOSITY\_USER\_BASIC 37 < MPI\_T\_VERBOSITY\_USER\_DETAIL < ... < MPI\_T\_VERBOSITY\_MPIDEV\_ALL. 38

39 40

15

16

17

18

19

20

21

22

23

24

25

26 27

28

## 15.3.2 Binding MPI Tool Information Interface Variables to MPI Objects

 $^{41}$ Each MPI tool information interface variable provides access to a particular control setting 42or performance property of the MPI implementation. A variable may refer to a specific 43 MPI object such as a communicator, datatype, or one-sided communication window, or the 44variable may refer more generally to the MPI environment of the process. Except for the 45last case, the variable must be bound to exactly one MPI object before it can be used. 46Table 15.2 lists all MPI object types to which an MPI tool information interface variable 47can be bound, together with the matching constant that MPI tool information interface 48routines return to identify the object type.

### Unofficial Draft for Comment Only

MPI_T_VERBOSITY_USER_BASIC	Basic information of interest to users
MPI_T_VERBOSITY_USER_DETAIL	Detailed information of interest to users
MPI_T_VERBOSITY_USER_ALL	All remaining information of interest to users
MPI_T_VERBOSITY_TUNER_BASIC	Basic information required for tuning
MPI_T_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning
MPI_T_VERBOSITY_TUNER_ALL	All remaining information required for tuning
MPI_T_VERBOSITY_MPIDEV_BASIC	Basic information for MPI implementors
MPI_T_VERBOSITY_MPIDEV_DETAIL	Detailed information for MPI implementors
MPI_T_VERBOSITY_MPIDEV_ALL	All remaining information for MPI implementors

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
MPI_T_BIND_MPI_SESSION	MPI session object

Table 15.1: MPI tool information interface verbosity levels

# Table 15.2: 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.*)

# 15.3.3 Convention for Returning Strings

Several MPI tool information interface functions return one or more strings. These functions <sup>43</sup> have two arguments for each string to be returned: an OUT parameter that identifies a <sup>44</sup> pointer to the buffer in which the string will be returned, and an INOUT parameter to pass <sup>45</sup> the length of the buffer. The user is responsible for the memory allocation of the buffer <sup>46</sup> and must pass the size of the buffer (n) as the length argument. Let n be the length <sup>47</sup> value specified to the function. On return, the function writes at most n-1 of the string's <sup>48</sup>

## Unofficial Draft for Comment Only

 $^{31}$ 

1 characters into the buffer, followed by a null terminator. If the returned string's length is  $\mathbf{2}$ greater than or equal to n, the string will be truncated to n-1 characters. In this case, the 3 length of the string plus one (for the terminating null character) is returned in the length 4 argument. If the user passes the null pointer as the buffer argument or passes 0 as the 5length argument, the function does not return the string and only returns the length of the 6 string plus one in the length argument. If the user passes the null pointer as the length  $\overline{7}$ argument, the buffer argument is ignored and nothing is returned.

8 MPI implementations behave as if they have an internal character array that is copied 9 to the output character array supplied by the user. Such output strings are only defined 10 to be equivalent if their notional source-internal character arrays are identical (up to and 11including the null terminator), even if the output string is truncated due to a small input 12length parameter n.

13 14

15

#### Initialization and Finalization 15.3.4

The MPI tool information interface requires a separate set of initialization and finalization 16routines. 17

18 19

25

26

## MPI\_T\_INIT\_THREAD(required, provided)

1	IN	required	desired level of thread support (integer)
2	OUT	provided	provided level of thread support (integer)
3			

### C binding

int MPI\_T\_init\_thread(int required, int \*provided)

All programs or tools that use the MPI tool information interface must initialize the 27MPI tool information interface in the processes that will use the interface before calling 28any other of its routines. A user can initialize the MPI tool information interface by calling 29MPI\_T\_INIT\_THREAD, which can be called multiple times. In addition, this routine initial-30 izes the thread environment for all routines in the MPI tool information interface. Calling  $^{31}$ this routine when the MPI tool information interface is already initialized has no effect 32 beyond increasing the reference count of how often the interface has been initialized. The 33 argument required is used to specify the desired level of thread support. The possible values 34and their semantics are identical to the ones that can be used with MPI\_INIT\_THREAD 35 listed in Section 11.6. The call returns in provided information about the actual level of 36 thread support that will be provided by the MPI implementation for calls to MPI tool 37 information interface routines. It can be one of the four values listed in Section 11.6. 38

The MPI specification does not require all MPI processes to exist before MPI is initial-39 ized. If the MPI tool information interface is used before initialization of MPI, the user is 40responsible for ensuring that the MPI tool information interface is initialized on all processes 41 it is used in. Processes created by the MPI implementation during initialization inherit the 42status of the MPI tool information interface (whether it is initialized or not as well as all 43 active sessions and handles) from the process from which they are created. 44

Processes created at runtime as a result of calls to MPI's dynamic process management 45require their own initialization before they can use the MPI tool information interface. 46

47 48

Advice to users. If MPI\_T\_INIT\_THREAD is called before MPI\_INIT\_THREAD, the

## **Unofficial Draft for Comment Only**

requested and provided 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. Likewise, when using the Sessions Model (Section 11.3), the requested and provided thread level for MPI\_T\_INIT\_THREAD may influence the behavior and return values of MPI\_SESSION\_INIT (see Section 11.3), with the same being 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()

### C binding

### 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.

### 15.3.5 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.

*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.

### Unofficial Draft for Comment Only

1 2

3

4

5 6

7

8

9

10

11

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

	124 CHAPTER 15. TOOL SUPPORT
1	MPI_INT
2	MPI_INT32_T
3	MPI_INT64_T
4	MPI_UNSIGNED
5	MPI_UNSIGNED_LONG
6	MPI_UNSIGNED_LONG_LONG
7	MPI_UINT32_T
8	MPI_UINT64_T
9	MPI_COUNT
10	MPI_CHAR
11	MPI_DOUBLE
12	
13	
14	Table 15.3: MPI datatypes that can be used by the MPI tool information interface
15	
16	This would cause unnecessary complexity in the implementation of tools based on the
17	MPI tool information interface. ( <i>End of rationale.</i> )
18	
19	The MPI tool information interface only relies on a subset of the basic MPI datatypes
20	and does not use any derived MPI datatypes. Table 15.3 lists all MPI datatypes that can
21	be returned by the MPI tool information interface to represent its variables.
22	The use of the datatype MPI_CHAR in the MPI tool information interface implies a null-
23	terminated character array, i.e., a string in the C language. If a variable has type MPI_CHAR,
23	the value of the count parameter returned by MPI_T_CVAR_HANDLE_ALLOC and
24 25	MPI_T_PVAR_HANDLE_ALLOC must be large enough to include any valid value, including
26	its terminating null character. The contents of returned MPI_CHAR arrays are only defined
	from index 0 through the location of the first null character.
27	from index o through the location of the first full character.
28	Rationale. The MPI tool information interface requires a significantly simpler type
29	system than MPI itself. Therefore, only its required subset must be present before
30	MPI initialization and MPI implementations do not need to initialize the complete
31	MPI datatype system. ( <i>End of rationale.</i> )
32	with tradatype system. (Dita of fattonate.)
33	For variables of type MPI_INT, an MPI implementation can provide additional informa-
34	tion by associating names with a fixed number of values. We refer to this information in
35	the following as an enumeration. In this case, the respective calls that provide additional
36	metadata for each control or performance variable, i.e., MPI_T_CVAR_GET_INFO (Sec-
37	tion 15.3.6), MPI_T_PVAR_GET_INFO (Section 15.3.7), and MPI_T_EVENT_GET_INFO
38	(Section 15.3.8), return a handle of type MPI_T_enum that can be passed to the follow-
39	ing functions to extract additional information. Thus, the MPI implementation can de-
40	scribe variables with a fixed set of values that each represents a particular state. Each
41	schoe variables with a fixed set of values that each represents a particular state. Each enumeration type can have $N$ different values, with a fixed $N$ that can be queried using
42	MPI_T_ENUM_GET_INFO.
43	
44	
45	
46	
47	
48	

CHAPTER 15. TOOL SUPPORT

	OM_GET_INTO(enumype, nu	ini, name, name_ien)	
IN	enumtype	enumeration to be queried (handle)	2
114	enuntype	chumeration to be queried (nandie)	3
OUT	num	number of discrete values represented by this	4
		enumeration (integer)	5
OUT	name	buffer to return the string containing the name of the	6
		enumeration item (string)	7
INOUT	nama lan	length of the string and (on huffor for nome (interes)	8
INCOT	name_len	length of the string and/or buffer for name (integer)	9
			10

MPI\_T\_ENUM\_GET\_INFO(enumtype, num, name, name\_len)

#### C binding

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 15.3.3.

The routine is required to return a name of at least length one. This name must be unique with respect to all other names for enumerations that the MPI implementation uses.

Names associated with individual values in each enumeration enumtype can be queried using MPI\_T\_ENUM\_GET\_ITEM.

MPI_T_ENUM	_GET_ITEM	(enumtype,	index,	value,	name,	name_len	ı)

IN	enumtype	enumeration to be queried (handle)	26
IN	index	number of the value to be queried in this	27
IIN	index	enumeration (integer)	28
			29
OUT	value	variable value (integer)	30
OUT	name	buffer to return the string containing the name of the	31 32
		enumeration item (string)	33
INOUT	name_len	length of the string and/or buffer for name (integer) $% \left( {{\left[ {{{\rm{name}}} \right]}_{\rm{and}}} \right)$	34
			35

#### C binding

The arguments name and name\_len are used to return the name of the enumeration item as described in Section 15.3.3.

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.

## 15.3.6 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.

10 11 12

1

2

3

4

5

6

7

8

9

Control Variable Query Functions

<sup>13</sup> An MPI implementation exports a set of N control variables through the MPI tool infor-<sup>14</sup> mation interface. If N is zero, then the MPI implementation does not export any control <sup>15</sup> variables, otherwise the provided control variables are indexed from 0 to N-1. This index <sup>16</sup> number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of control variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a control variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

Advice to users. While the MPI tool information interface guarantees that indices or variable properties do not change during a particular run of an MPI program, it does not provide a similar guarantee between runs. (*End of advice to users.*)

The following function can be used to query the number of control variables, num\_cvar:

28 29 30

31 32

33 34

35

36

37

 $^{24}$ 

25

26

27

MPI\_T\_CVAR\_GET\_NUM(num\_cvar)

OUT num\_cvar

returns number of control variables (integer)

C binding

int MPI\_T\_cvar\_get\_num(int \*num\_cvar)

The function  $\mathsf{MPI\_T\_CVAR\_GET\_INFO}$  provides access to additional information for each variable.

	desc_len, bind, scope	x, name, name_len, verbosity, datatype, enumtype, desc, e)
IN	cvar_index	index of the control variable to be queried, value
		between 0 and $num\_cvar - 1$ (integer)
OUT	name	buffer to return the string containing the name of the
		control variable (string)
INOUT	name_len	length of the string and/or buffer for name (integer)
OUT	verbosity	verbosity level of this variable (integer)
OUT	datatype	MPI datatype of the information stored in the
001	uatatype	control variable (handle)
OUT	enumtype	optional descriptor for enumeration information (handle)
OUT	desc	buffer to return the string containing a description of
		the control variable (string)
INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)
OUT	bind	type of MPI object to which this variable must be
		bound (integer)
OUT	scope	scope of when changes to this variable are possible
		(integer)
	S	

## 

After a successful call to MPI\_T\_CVAR\_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\_CVAR\_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 control variable as described in Section 15.3.3.

If completed successfully, the routine is required to return a name of at least length one. The name must be unique with respect to all other names for control variables used by the MPI implementation.

The argument verbosity returns the verbosity level of the variable (see Section 15.3.1).

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 15.3.5. 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 47 as described in Section 15.3.3.

#### Unofficial Draft for Comment Only

26

27

28

29 30

31

32 33

34

35

36 37

38 39

40 41

42

43 44

 $45 \\ 46$ 

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 of this call.

<sup>4</sup> The parameter bind returns the type of the MPI object to which the variable must be <sup>5</sup> bound or the value MPI\_T\_BIND\_NO\_OBJECT (see Section 15.3.2).

6 The scope of a variable determines whether changing a variable's value is either local 7to the MPI process or must be done by the user across multiple connected MPI processes. 8 The latter is further split into variables that require changes in a group of MPI processes 9 and those that require collective changes among all connected MPI processes. Both cases 10 can require variables on all participating MPI processes either to be set to consistent (but 11potentially different) values or to equal values. The description provided with the variable 12must contain an explanation about the requirements and/or restrictions for setting the 13particular variable.

On successful return from MPI\_T\_CVAR\_GET\_INFO, the argument scope will be set to
 one of the constants listed in Table 15.4.

<sup>16</sup> If the name of a control variable is equivalent across connected MPI processes, the
 <sup>17</sup> following OUT parameters must be identical: verbosity, datatype, enumtype, bind, and scope.
 <sup>18</sup> The returned description must be equivalent.

19		
20	Scope Constant	Description
21	MPI_T_SCOPE_CONSTANT	read-only, value is constant
22	MPI_T_SCOPE_READONLY	read-only, cannot be written, but can change
23	MPI_T_SCOPE_LOCAL	may be writeable, writing is a local operation
24	MPI_T_SCOPE_GROUP	may be writeable, must be set to consistent values
25		across a group of connected MPI processes
26	MPI_T_SCOPE_GROUP_EQ	may be writeable, must be set to the same value
27		across a group of connected MPI processes
28	MPI_T_SCOPE_ALL	may be writeable, must be set to consistent values
29		across all connected MPI processes
30	MPI_T_SCOPE_ALL_EQ	may be writeable, must be set to the same value
31		across all connected MPI processes
32		
33	Table	15 4. Commenter la serie blas
34	Table	15.4: Scopes for control variables
35		*
36	Advice to users. The	scope of a variable only indicates if a variable might
37		arantee that it can be changed at any time. (End of adv
38	to users.)	
39		
40		
41		
42	MPI_T_CVAR_GET_INDEX(nan	ne, cvar_index)
43	IN name	name of the control variable (string)
44	OUT cvar_index	index of the control variable (integer)
45		
46	C binding	
47	6	and than thoma int towar index)
48	int mPi_1_cvar_get_index(Co	onst char *name, int *cvar_index)

MPI\_T\_CVAR\_GET\_INDEX is a function for retrieving the index of a control variable given a known variable name. The name parameter is provided by the caller, and cvar\_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 control variable provided by the implementation at the time of the call.

*Rationale.* This routine is provided to enable fast retrieval of control 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.*)

Example: Printing All Control Variables

**Example 15.4** The following example shows how the MPI tool information interface can be used to query and to print the names of all available control variables.

```
#include <stdio.h>
                                                                                       23
                                                                                       ^{24}
#include <stdlib.h>
                                                                                       25
#include <mpi.h>
                                                                                       26
                                                                                       27
int main(int argc, char *argv[]) {
                                                                                       28
  int i, err, num, namelen, bind, verbose, scope;
                                                                                       29
  int threadsupport;
  char name[100];
                                                                                       30
                                                                                       31
  MPI_Datatype datatype;
                                                                                       32
                                                                                       33
  err=MPI_T_init_thread(MPI_THREAD_SINGLE,&threadsupport);
                                                                                       34
  if (err!=MPI_SUCCESS)
                                                                                       35
    return err;
                                                                                       36
                                                                                       37
  err=MPI_T_cvar_get_num(&num);
                                                                                       38
  if (err!=MPI_SUCCESS)
                                                                                       39
    return err;
                                                                                       40
                                                                                       41
  for (i=0; i<num; i++) {</pre>
                                                                                       42
    namelen=100;
    err=MPI_T_cvar_get_info(i, name, &namelen,
                                                                                       43
                                                                                       44
             &verbose, &datatype, NULL,
             NULL, NULL, /*no description */
                                                                                       45
                                                                                       46
             &bind, &scope);
                                                                                       47
    if (err!=MPI_SUCCESS && err!=MPI_T_ERR_INVALID_INDEX) return err;
                                                                                       48
    printf("Var %i: %s\n", i, name);
```

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

14

15

16 17

18 19 20

21

```
1
        }
2
3
        err=MPI_T_finalize();
4
        if (err!=MPI_SUCCESS)
5
          return 1;
6
        else
7
          return 0;
8
      }
9
10
      Handle Allocation and Deallocation
11
      Before reading or writing the value of a variable, a user must first allocate a handle of type
12
      MPI_T_cvar_handle for the variable by binding it to an MPI object (see also Section 15.3.2).
13
14
            Rationale.
                         Handles used in the MPI tool information interface are distinct from
15
           handles used in the remaining parts of the MPI standard because they must be usable
16
           before MPI is initialized and after MPI is finalized. Further, accessing handles, in
17
           particular for performance variables, can be time critical and having a separate handle
18
           space enables optimizations. (End of rationale.)
19
20
21
      MPI_T_CVAR_HANDLE_ALLOC(cvar_index, obj_handle, handle, count)
22
23
        IN
                  cvar_index
                                                index of control variable for which handle is to be
24
                                                allocated (index)
25
        IN
                  obj_handle
                                                reference to a handle of the MPI object to which this
26
                                                variable is supposed to be bound (pointer)
27
        OUT
                  handle
                                                allocated handle (handle)
28
29
        OUT
                  count
                                                number of elements used to represent this variable
30
                                                (integer)
^{31}
32
      C binding
33
      int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,
34
                      MPI_T_cvar_handle *handle, int *count)
35
36
          This routine binds the control variable specified by the argument index to an MPI object.
37
      The object is passed in the argument obj_handle as an address to a local variable that stores
38
      the object's handle. The argument obj_handle is ignored if the MPI_T_CVAR_GET_INFO
      call for this control variable returned MPI_T_BIND_NO_OBJECT in the argument bind. The
39
      handle allocated to reference the variable is returned in the argument handle. Upon success-
40
41
      ful return, count contains the number of elements (of the datatype returned by a previous
42
      MPI_T_CVAR_GET_INFO call) used to represent this variable.
43
           Advice to users. The count can be different based on the MPI object to which the
44
           control variable was bound. For example, variables bound to communicators could
45
           have a count that matches the size of the communicator.
46
47
           It is not portable to pass references to predefined MPI object handles, such as
48
           MPI_COMM_WORLD to this routine, since their implementation depends on the MPI
```

library. Instead, such object handles should be stored in a local variable and the address of this local variable should be passed into MPI\_T\_CVAR\_HANDLE\_ALLOC. (*End of advice to users.*)

The value of cvar\_index should be in the range 0 to num\_cvar -1, where num\_cvar is the number of available control variables as determined from a prior call to MPI\_T\_CVAR\_GET\_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI\_T\_CVAR\_GET\_INFO.

			0
	<i>i</i>		9 10
MPI_T_C	CVAR_HANDLE_FREE(har	ndle)	11
INOUT	handle	handle to be freed (handle)	12
			13
C bindi	ng		14
		I_T_cvar_handle *handle)	15
<b>XX71</b>			16
		eded, a user of the MPI tool information interface should to free the handle and the associated resources in the	17
		essful return, MPI sets the handle to	18
-	VAR_HANDLE_NULL.	Ssiul letuin, with sets the handle to	19
			20
Control V	ariable Access Functions		21 22
control v			22
			24
MPLT (	CVAR_READ(handle, buf)		25
	· · · · · ·		26
IN	handle	handle to the control variable to be read (handle)	27
OUT	buf	initial address of storage location for variable value	28
		(choice)	29
			30
C bindi	ng		31
int MPI.	_T_cvar_read(MPI_T_cva	r_handle handle, void *buf)	32
This	routine queries the value of	f a control variable identified by the argument handle and	33
	-	fied by the parameter <b>buf</b> . The user must ensure that the	34
		nold the entire value of the control variable (based on the	35
		prior corresponding calls to MPI_T_CVAR_GET_INFO	36
and MPI.	_T_CVAR_HANDLE_ALLC	DC, respectively).	37 38
			39
			40
	CVAR_WRITE(handle, buf)		41
IN	handle	handle to the control variable to be written (handle)	42
IN	buf	initial address of storage location for variable value	43
		(choice)	44
			45
C bindi	ng		46
int MPI	_T_cvar_write(MPI_T_cv	ar_handle handle, const void *buf)	47
			48

 $\overline{7}$ 

1	This routine sets the value of the control variable identified by the argument handle to
2	the data stored in the buffer identified by the parameter <b>buf</b> . The user must ensure that the
3	buffer is of the appropriate size to hold the entire value of the control variable (based on the
4	returned datatype and count from prior corresponding calls to MPI_T_CVAR_GET_INFO
5	and MPI_T_CVAR_HANDLE_ALLOC, respectively).
6	If the variable has a global scope (as returned by a prior corresponding
7	MPI_T_CVAR_GET_INFO call), any write call to this variable must be issued by the user
8	in all connected (as defined in Section 11.10.4) MPI processes. If the variable has group
9	scope, any write call to this variable must be issued by the user in all MPI processes in
10	
11	the group, which must be described by the MPI implementation in the description by the MPI_T_CVAR_GET_INFO.
12	In both cases, the user must ensure that the writes in all participating MPI processes
13	are consistent. If the scope is either MPI_T_SCOPE_ALL_EQ or MPI_T_SCOPE_GROUP_EQ
14	
15	this means that the variable in all connected MPI processes or MPI processes of the group,
16	respectively, must be set to the same value.
17	If it is not possible to change the variable at the time the call is made, the function
18	returns either MPI_T_ERR_CVAR_SET_NOT_NOW, if there may be a later time at which the
19	variable could be set, or MPI_T_ERR_CVAR_SET_NEVER, if the variable cannot be set for the
20	remainder of the application's execution.
20	
21	Example: Reading the Value of a Control Variable
23	
24	<b>Example 15.5</b> The following example shows a routine that can be used to query the
25	value with a control variable with a given index. The example assumes that the variable is
26	intended to be bound to an MPI communicator.
20	
28	int getValue_int_comm(int index, MPI_Comm comm, int *val) {
29	int err, count;
30	MPI_T_cvar_handle handle;
31	
32	<pre>/* This example assumes that the variable index */</pre>
33	<pre>/* can be bound to a communicator */</pre>
33 34	
	err=MPI_T_cvar_handle_alloc(index, &comm, &handle, &count);
35	if (err!=MPI_SUCCESS) return err;
36 27	
37	/* The following assumes that the variable is $*/$
38	<pre>/* represented by a single integer */</pre>
39	
40	<pre>err=MPI_T_cvar_read(handle,val);</pre>
41	if (err!=MPI_SUCCESS) return err;
42	
43	err=MPI_T_cvar_handle_free(&handle);
44	return err;
45	}
46	
47	
48	

### 15.3.7 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.*)

Some performance variables and classes refer to *events*. In general, such events describe state transitions within software or hardware related to the performance of an MPI application. The events offered through the callback-driven event-notification interface described in Section 15.3.8 also refer to such state transitions; however, the set of state transitions referred to by performance variables and events as described in Section 15.3.8 may not be identical.

#### Performance Variable Classes

Each performance variable is associated with a class that describes its basic semantics, possible datatypes, basic behavior, its starting value, whether it can overflow, and when and how an MPI implementation can change the variable's value. The starting value is the value that is assigned to the variable the first time that it is used or whenever it is reset.

Advice to users. If a performance variable belongs to a class that can overflow, it is up to the user to protect against this overflow, e.g., by frequently reading and resetting the variable value. (*End of advice to users.*)

Advice to implementors. MPI implementations should use large enough datatypes for each performance variable to avoid overflows under normal circumstances. (End of advice to implementors.)

The classes are defined by the following constants:

#### • MPI\_T\_PVAR\_CLASS\_STATE

A performance variable in this class represents a set of discrete states. Variables of this class are represented by MPI\_INT and can be set by the MPI implementation at any time. Variables of this type should be described further using an enumeration, as discussed in Section 15.3.5. 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 <sup>45</sup> level of a resource. The value of a variable of this class can change at any time to match <sup>46</sup> the current utilization level of the resource. Values returned from variables in this class <sup>47</sup> are non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, <sup>48</sup>

#### Unofficial Draft for Comment Only

 $^{24}$ 

MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

• MPI\_T\_PVAR\_CLASS\_SIZE

A performance variable in this class represents a value that is the size of a resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG,

MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current size of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

12 13

## • 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

1

2

3

4

5

6

7

8

9

10

11

14

15

16

17

18

19

20 21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37 38

39

40

41

42

43

44

45

46 47

represents a sum of arguments processed during a specific event (e.g., the amount of memory allocated by all memory allocations). This class is similar to the counter class, but instead of counting individual events, the value can be incremented by arbitrary amounts. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value for variables of this class is 0. Variables of this class can overflow.

• MPI\_T\_PVAR\_CLASS\_TIMER

The value of a performance variable in this class represents the aggregated time that the MPI implementation spends executing a particular event, type of event, or section of the MPI library. This class has the same basic semantics as MPI\_T\_PVAR\_CLASS\_AGGREGATE, but explicitly records a timing value. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value for variables of this class is 0. If the type MPI\_DOUBLE is used, the units that represent time in this datatype must match the units used by MPI\_WTIME. Otherwise, the time units should be documented, e.g., in the description returned by MPI\_T\_PVAR\_GET\_INFO. Variables of this class can overflow.

MPI\_T\_PVAR\_CLASS\_GENERIC

This class can be used to describe a variable that does not fit into any of the other classes. For variables in this class, the starting value is variable-specific and implementation-defined.

#### Performance Variable Query Functions

An MPI implementation exports a set of N performance variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any performance variables; otherwise the provided performance variables are indexed from 0 to N-1. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of performance variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a performance variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

The following function can be used to query the number of performance variables, num\_pvar:

MPI\_T\_PVAR\_GET\_NUM(num\_pvar)

OUT num\_pvar returns number of performance variables (integer)

C binding

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6 7

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22 23

 $^{24}$ 

25

26

27 28

29

30 31

32 33

34

35

36

37

38

39

40 41

42 43

44

45

46 47

100		
int MPI_T	_pvar_get_num(int	*num_pvar)
The fu each variab		$R_GET\_INFO$ provides access to additional information for
MPI_T_PV		_index, name, name_len, verbosity, var_class, datatype, desc_len, bind, readonly, continuous, atomic)
IN	pvar_index	index of the performance variable to be queried between 0 and $num\_pvar - 1$ (integer)
OUT	name	buffer to return the string containing the name of the performance variable (string)
INOUT	name_len	length of the string and/or buffer for name (integer)
OUT	verbosity	verbosity level of this variable (integer)
OUT	var_class	class of performance variable (integer)
OUT	datatype	MPI datatype of the information stored in the performance variable (handle)
OUT	enumtype	optional descriptor for enumeration information (handle)
OUT	desc	buffer to return the string containing a description of the performance variable (string)
INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)
OUT	bind	type of MPI object to which this variable must be bound (integer)
OUT	readonly	flag indicating whether the variable can be written/reset (integer)
OUT	continuous	flag indicating whether the variable can be started and stopped or is continuously active (integer)
OUT	atomic	flag indicating whether the variable can be atomically read and reset (integer)
C binding		t pvar_index, char *name, int *name_len,
	int *verbosit MPI_T_enum *e	y, int *var_class, MPI_Datatype *datatype, enumtype, char *desc, int *desc_len, int *bind, , int *continuous, int *atomic)
calls to this information If any tion will ig The any variable as	s routine that query n. An MPI implemen OUT parameter to M nore the parameter a rguments name and	PI_T_PVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same tation is not allowed to alter any of the returned values. IPI_T_PVAR_GET_INFO is a NULL pointer, the implementa- and not return a value for the parameter. name_len are used to return the name of the performance in 15.3.3. If completed successfully, the routine is required gth one.

2

 $12 \\ 13$ 

 $\frac{43}{44}$ 

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

The argument verbosity returns the verbosity level of the variable (see Section 15.3.1). The class of the performance variable is returned in the parameter var\_class. The class must be one of the constants defined in Section 15.3.7.

The combination of the name and the class of the performance variable must be unique with respect to all other names for performance variables used by the MPI implementation.

Advice to implementors. Groups of variables that belong closely together, but have different classes, can have the same name. This choice is useful, e.g., to refer to multiple variables that describe a single resource (like the level, the total size, as well as high and low watermarks). (End of advice to implementors.)

The argument datatype returns the MPI datatype that is used to represent the performance variable.

If the variable is of type MPI\_INT, MPI can optionally specify an enumeration for the values represented by this variable and return it in enumtype. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 15.3.5. 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.

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 from this function.

The parameter bind returns the type of the MPI object to which the variable must be bound or the value MPI\_T\_BIND\_NO\_OBJECT (see Section 15.3.2).

Upon return, the argument **readonly** is set to zero if the variable can be written or reset by the user. It is set to one if the variable can only be read.

Upon return, the argument **continuous** is set to zero if the variable can be started and stopped by the user, i.e., it is possible for the user to control if and when the value of a variable is updated. It is set to one if the variable is always active and cannot be controlled by the user.

Upon return, the argument **atomic** is set to zero if the variable cannot be read and reset atomically. Only variables for which the call sets **atomic** to one can be used in a call to MPI\_T\_PVAR\_READRESET.

If a performance variable has an equivalent name and has the same class across connected MPI processes, the following OUT parameters must be identical: verbosity, varclass, datatype, enumtype, bind, readonly, continuous, and atomic. The returned description must be equivalent.

MPI_T_PVAR_	GET_INDEX(name, `	var_class, pva	r_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)

#### C binding

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

### Unofficial Draft for Comment Only

43 44 45

> 46 47

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.

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.)

## <sup>16</sup> 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. Starting, stopping, reading, writing, or resetting a variable in one performance experiment session shall not influence whether a variable is started, stopped, read, written, or reset in another performance experiment session.

25 26

27

33

34 35

36 37

38 39

40

42

43

44

### MPI\_T\_PVAR\_SESSION\_CREATE(session)

 28
 OUT
 session
 identifier of performance session (handle)

 29
 30
 C binding

 31
 int MPI\_T\_pvar\_session\_create(MPI\_T\_pvar\_session \*session)

 32
 This is a session in the session in the session in the session in the session is a session in the session is a session in the session in the session is a session in the set of the set of

This call creates a new session for accessing performance variables and returns a handle for this session in the argument session of type MPI\_T\_pvar\_session.

MPI\_T\_PVAR\_SESSION\_FREE(session)

session

identifier of performance experiment session (handle)

## C binding

INOUT

int MPI\_T\_pvar\_session\_free(MPI\_T\_pvar\_session \*session)

This call frees an existing session. Calls to the MPI tool information interface can no longer be made within the context of a session after it is freed. On a successful return, MPI sets the session identifier to MPI\_T\_PVAR\_SESSION\_NULL.

- 45 46
- 47 48

6 7

8

9

10

11

12

13

Handle Allocation and Deallocation			1
	, , , , , , , , , , , , , , , , , , ,	user must first allocate a handle of type ding it to an MPI object (see also Section $15.3.2$ ).	2 3 4
MPLT PV	AR HANDLE ALLOC(session	pvar_index, obj_handle, handle, count)	5 6
IN IN	session	identifier of performance experiment session (handle)	7
IN	pvar_index	index of performance experiment session (mandle) be allocated (integer)	8 9 10
IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)	11 12
OUT	handle	allocated handle (handle)	13 14
OUT	count	number of elements used to represent this variable (integer)	15 16 17
C binding			18
int MPI_T	_pvar_handle_alloc(MPI_T_ void *obj_handle, MPI	<pre>pvar_session session, int pvar_index, _T_pvar_handle *handle, int *count) e variable specified by the argument index to an</pre>	19 20 21 22
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. Advice to users. The count can be different based on the MPI object to which the			
	have a count that matches th	For example, variables bound to communicators e size of the communicator.	32 33
It is not portable to pass references to predefined MPI object handles, such as MPI_COMM_WORLD, to this routine, since their implementation depends on the MPI library. Instead, such an object handle should be stored in a local variable and the address of this local variable should be passed into MPI_T_PVAR_HANDLE_ALLOC. ( <i>End of advice to users.</i> )			
The 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.			<ul> <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>

1 MPI\_T\_PVAR\_HANDLE\_FREE(session, handle) 2 IN session identifier of performance experiment session (handle) 3 INOUT handle handle to be freed (handle) 4 5C binding 6 int MPI\_T\_pvar\_handle\_free(MPI\_T\_pvar\_session session, 7MPI\_T\_pvar\_handle \*handle) 8 9 When a handle is no longer needed, a user of the MPI tool information interface should 10 call MPI\_T\_PVAR\_HANDLE\_FREE to free the handle in the session identified by the pa-11 rameter session and the associated resources in the MPI implementation. On a successful 12return, MPI sets the handle to MPI\_T\_PVAR\_HANDLE\_NULL. 13 14Starting and Stopping of Performance Variables 1516Performance variables that have the continuous flag set during the query operation are 17continuously operating once a handle has been allocated. Such variables may be queried at 18any time, but they cannot be started or stopped by the user. All other variables are in a 19stopped state after their handle has been allocated; their values are not updated until they have been started by the user. 202122 MPI\_T\_PVAR\_START(session, handle) 23 $^{24}$ IN session identifier of performance experiment session (handle) 25handle of a performance variable (handle) IN handle 2627C binding 28int MPI\_T\_pvar\_start(MPI\_T\_pvar\_session session, MPI\_T\_pvar\_handle handle) 29 30 This functions starts the performance variable with the handle identified by the pa- $^{31}$ rameter handle in the session identified by the parameter session. 32 If the constant MPI\_T\_PVAR\_ALL\_HANDLES is passed in handle, the MPI implementa-33 tion attempts to start all variables within the session identified by the parameter session for 34which handles have been allocated. In this case, the routine returns MPI\_SUCCESS if all vari-35 ables are started successfully (even if there are no non-continuous variables to be started), 36 otherwise MPI\_T\_ERR\_PVAR\_NO\_STARTSTOP is returned. Continuous variables and vari-37 ables that are already started are ignored when MPI\_T\_PVAR\_ALL\_HANDLES is specified. 38 39 MPI\_T\_PVAR\_STOP(session, handle) 4041 IN session identifier of performance experiment session (handle) 42IN handle handle of a performance variable (handle) 43 44C binding 45int MPI\_T\_pvar\_stop(MPI\_T\_pvar\_session session, MPI\_T\_pvar\_handle handle) 4647This functions stops the performance variable with the handle identified by the param-48eter handle in the session identified by the parameter session.

## Unofficial Draft for Comment Only

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.

#### Performance Variable Access Functions

MPI_T_PVAR_READ(session, handle, buf)				
IN session	identifier of performance experiment session (handle)	13 14		
IN handle	handle of a performance variable (handle)	15		
OUT buf	initial address of storage location for variable value	16		
OOT Bui	(choice)	17		
	(choice)	18		
C binding		19		
0	ssion session, MPI_T_pvar_handle handle,	20		
void *buf)	····· ·····, ·······, ·········,	21 22		
		22		
-	ies the value of the performance variable with the	20		
	y the parameter <b>session</b> and stores the result in the The user is responsible to ensure that the buffer	25		
v *	entire value of the performance variable (based on	26		
the datatype and count returned by the	-	27		
MPI_T_PVAR_GET_INFO and MPI_T_F		28		
The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the func-				
tion MPI_T_PVAR_READ.		30		
		31		
MDL T DVAD WDITE (associan handle h		32		
MPI_T_PVAR_WRITE(session, handle, b	,	33		
IN session	identifier of performance experiment session (handle)	$\frac{34}{35}$		
IN handle	handle of a performance variable (handle)	36		
IN buf	initial address of storage location for variable value	37		
	(choice)	38		
		39		
C binding				
	ession session, MPI_T_pvar_handle handle,	41		
const void *buf)				
The MPL T PVAR WRITE call atte	empts to write the value of the performance variable	43		
		44		
· · ·	with the handle identified by the parameter handle in the session identified by the parameter session. The value to be written is passed in the buffer identified by the parameter buf. The			

# **Unofficial Draft for Comment Only**

user must ensure that the buffer is of the appropriate size to hold the entire value of the per-

 $\mathbf{2}$ 

 $\overline{7}$ 

1 2 3 4 5 6 7 8	calls to MF If it is MPI_T_ERF The co	ance variable (based on the datatype and count returned by the corresponding previous to MPI_T_PVAR_GET_INFO and MPI_T_PVAR_HANDLE_ALLOC, respectively). If it is not possible to change the variable, the function returns _T_ERR_PVAR_NO_WRITE. The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the func- MPI_T_PVAR_WRITE.		
9	MPI_T_PV	/AR_RESET(session, handle)		
10	IN	session	identifier of performance experiment session (handle)	
11 12	IN	handle	handle of a performance variable (handle)	
13 14 15	C binding int MPI_T		ession session, MPI_T_pvar_handle handle)	
16 17 18 19 20 21 22 23 24 25 26	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 15.3.7. 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.			
27	MPI_T_PV	/AR_READRESET(session, hai	ndle, buf)	
28 29	IN	session	identifier of performance experiment session (handle)	
30	IN	handle	handle of a performance variable (handle)	
31 32 33	Ουτ	buf	initial address of storage location for variable value (choice)	
33 34 35 36	C binding int MPI_T	g '_pvar_readreset(MPI_T_pv MPI_T_pvar_handle ha		
<ol> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> <li>46</li> <li>47</li> </ol>	MPI_T_PV If atomic of MPI_T_ERF The co tion MPI_T Advia inform	AR_RESET with the same sepperations on this variable a R_PVAR_NO_ATOMIC. Instant MPI_T_PVAR_ALL_HA T_PVAR_READRESET. <i>ce to implementors.</i> Samplin mation interface, in particula	functionality of MPI_T_PVAR_READ and emantics as if these two calls were called separately. The not supported, this routine returns NDLES cannot be used as an argument for the func- ag-based tools rely on the ability to call the MPI tool or routines to start, stop, read, write, and reset per- gram context, including asynchronous contexts such	
47 48	as sig	snal handlers. MPI implement	tations should strive, if possible in their particular	

#### 15.3. THE MPI TOOL INFORMATION INTERFACE

environment, to enable these usage scenarios for all or a subset of the routines mentioned above. If implementing only a subset, the read, write, and reset routines are typically the most critical for sampling based tools. An MPI implementation should clearly document any restrictions on the program contexts in which the MPI tool information interface can be used. Restrictions might include guaranteeing usage outside of all signals or outside a specific set of signals. Any restrictions could be documented, for example, through the description returned by MPI\_T\_PVAR\_GET\_INFO. (*End of advice to implementors.*)

*Rationale.* All routines to read, to write or to reset performance variables require the session argument. This requirement keeps the interface consistent and allows the use of MPI\_T\_PVAR\_ALL\_HANDLES where appropriate. Further, this opens up additional performance optimizations for the implementation of handles. (*End of rationale.*)

#### Example: Tool to Detect Receives with Long Unexpected Message Queues

**Example 15.6** 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 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.

```
#include <stdio.h>
#include <stdib.h>
#include <string.h>
#include <string.h>
#include <assert.h>
#include <mpi.h>
/* Global variables for the tool */
static MPI_T_pvar_session session;
static MPI_T_pvar_handle handle;
int MPI_Init(int *argc, char ***argv ) {
    int err, num, i, index, namelen, verbosity;
    int var_class, bind, threadsup;
    int readonly, continuous, atomic, count;
    char name[18];
    MPI_Comm comm;
    MPI_Datatype datatype;
```

 $\mathbf{2}$ 

 $^{24}$ 

```
1
           MPI_T_enum enumtype;
2
3
           err=PMPI_Init(argc, argv);
4
           if (err!=MPI_SUCCESS) return err;
5
6
           err=PMPI_T_init_thread(MPI_THREAD_SINGLE, &threadsup);
7
           if (err!=MPI_SUCCESS) return err;
8
9
           err=PMPI_T_pvar_get_num(&num);
10
           if (err!=MPI_SUCCESS) return err;
11
           index=-1;
12
           i=0;
13
           while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {</pre>
14
                 /* Pass a buffer that is at least one character longer than */
15
                 /* the name of the variable being searched for to avoid */
16
                 /* finding variables that have a name that has a prefix */
17
                 /* equal to the name of the variable being searched. */
18
                 namelen=18;
19
                  err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
20
                          &var_class, &datatype, &enumtype, NULL, NULL, &bind,
21
                          &readonly, &continuous, &atomic);
22
                  if (strcmp(name,"MPI_T_UMQ_LENGTH")==0) index=i;
23
                  i++; }
24
           if (err!=MPI_SUCCESS) return err;
25
26
           /* this could be handled in a more flexible way for a generic tool */
27
           assert(index>=0);
           assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
28
29
           assert(datatype==MPI_INT);
30
           assert(bind==MPI_T_BIND_MPI_COMM);
31
32
           /* Create a session */
33
           err=PMPI_T_pvar_session_create(&session);
34
           if (err!=MPI_SUCCESS) return err;
35
36
           /* Get a handle and bind to MPI_COMM_WORLD */
37
           comm=MPI_COMM_WORLD;
38
           err=PMPI_T_pvar_handle_alloc(session, index, &comm, &handle, &count);
39
           if (err!=MPI_SUCCESS) return err;
40
41
           /* this could be handled in a more flexible way for a generic tool */
42
           assert(count==1);
43
44
           /* Start variable */
45
           err=PMPI_T_pvar_start(session, handle);
46
           if (err!=MPI_SUCCESS) return err;
47
48
           return MPI_SUCCESS;
```

}

Part 2—Testing the Queue Lengths During Receives: During every receive operation, the tool reads the unexpected queue length through the matching performance variable and compares it against a predefined threshold.

```
#define THRESHOLD 5
```

int value, err;

```
{
```

}

```
if (comm==MPI_COMM_WORLD) {
    err=PMPI_T_pvar_read(session, handle, &value);
    if ((err==MPI_SUCCESS) && (value>THRESHOLD))
    {
        /* tool identified receive called with long UMQ */
        /* execute tool functionality, */
        /* e.g., gather and print call stack */
    }
}
```

```
return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
```

Part 3—Termination: In the wrapper for MPI\_FINALIZE, the MPI tool information interface is finalized.

```
int MPI_Finalize(void)
{
    int err;
    err=PMPI_T_pvar_handle_free(session, &handle);
    err=PMPI_T_pvar_session_free(&session);
    err=PMPI_T_finalize();
    return PMPI_Finalize();
}
```

## 15.3.8 Events

During the execution of an MPI application, the MPI implementation can raise events of a specific type to inform the user of a state change in the implementation. Event types describe specific state changes within the MPI implementation. In comparison to aggregate performance variables, events provide per-instance information on such state changes. The MPI implementation is said to *raise an event* when it invokes a callback function previously registered for the corresponding event type by the user. Each callback invocation for a specific event instance has a timestamp associated with it, which can be queried by the user, describing the time when the event was observed by the implementation. This decouples 

### Unofficial Draft for Comment Only

the observation of the state change from the communication of this information to the user.  $\mathbf{2}$ A timestamp in this context is a count of clock ticks elapsed since some time in the past and represented as a variable of type MPI\_Count.

## **Event Sources**

As a means to manage multiple state changes to be observed concurrently by different parts of the software and hardware system, the event interface of the MPI Tool Information Interface uses the concept of *sources*. A source in this context is a concept describing the logical entity raising the event. A source may or may not directly represent a concrete part of the software or hardware system. This concept is used primarily to describe partial ordering of events across different components where total ordering cannot necessarily be determined or is too costly to enforce. 

The following function can be used to query the number of event sources, *num\_sources*:

MPI\_T\_SOURCE\_GET\_NUM(num\_sources)

num\_sources

OUT

returns number of event sources (integer)

#### C binding

int MPI\_T\_source\_get\_num(int \*num\_sources)

The number of available event sources can be queried with a call to MPI\_T\_SOURCE\_GET\_NUM. An MPI implementation is allowed to increase the number of  $^{24}$ sources during the execution of an MPI process. However, MPI implementations are not allowed to change the index of an event source or to delete an event source once it has been made visible to the user (e.g., if new event sources become available via dynamic loading of additional components in the MPI implementation). 

MPI_T_SOURCE_GET_INFO(source_index, name, name_len, desc, desc_len, ordering,
ticks_per_second, max_ticks, info)

IN	source_index	index of the source to be queried between 0 and	3
		num_sources $-1$ (integer)	4
OUT	name	buffer to return the string containing the name of the	5
001	lianc	source (string)	6 7
MOUT	and the		8
INOUT	name_len	length of the string and/or buffer for name (integer)	
OUT	desc	buffer to return the string containing the description	9
		of the source (string)	10
	line lin		11
INOUT	desc_len	length of the string and/or buffer for desc (integer)	12
OUT	ordering	flag indicating chronological ordering guarantees	13
		given by the source (integer)	14
OUT	ticks_per_second	the number of ticks per second for the timer of this	15
001		source (integer)	16
			17
OUT	max_ticks	the maximum count of ticks reported by this source	18
		before overflow occurs (integer)	19
OUT	info	optional info object (handle)	20
			21

### C binding

int MPI\_T\_source\_get\_info(int source\_index, char \*name, int \*name\_len, char \*desc, int \*desc\_len, MPI\_T\_source\_order \*ordering, MPI\_Count \*ticks\_per\_second, MPI\_Count \*max\_ticks, MPI\_Info \*info)

A call to MPI\_T\_SOURCE\_GET\_INFO returns additional information on the source identified by the source\_index argument.

The arguments name and name\_len are used to return the name of the source as described in Section 15.3.3.

The arguments desc and desc\_len are used to return the description of the source as described in Section 15.3.3.

The ordering argument returns whether event callbacks of this source will be invoked in chronological order, i.e., the timestamps reported by MPI\_T\_EVENT\_GET\_TIMESTAMP of subsequent events of the same source are monotonically increasing. The value of ordering can be MPI\_T\_SOURCE\_ORDERED or MPI\_T\_SOURCE\_UNORDERED.

The ticks\_per\_seconds argument returns the number of ticks elapsed in one second for the timer used for the specific source.

The max\_ticks argument returns the largest number of ticks reported by this source as a timestamp before the value overflows.

Advice to users. As the size of MPI\_Count is defined in relation to the types MPI\_Aint and MPI\_Offset, the effective size of MPI\_Count may lead to overflows of the timestamp values reported. Users can use the argument max\_ticks to mitigate resulting problems. (End of advice to users.)

MPI can optionally return an info object containing the default hints set for this source. If the argument to info provided by the user is the NULL pointer, this argument is ignored,

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

1 2

22 23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41 42

43

44

45

4647

otherwise an MPI implementation is required to return all hints that are supported by
 the implementation for this source 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 via
 MPI\_INFO\_FREE.

 9
 MPI\_T\_SOURCE\_GET\_TIMESTAMP(source\_index, timestamp)

 10
 IN
 source\_index
 index of the source (integer)

 11
 OUT
 timestamp
 current timestamp from specified source (integer)

### C binding

```
14
```

17

30

31

32

33

34

35

36 37

38 39

40

41

13

8

int MPI\_T\_source\_get\_timestamp(int source\_index, MPI\_Count \*timestamp)
To enable proper query of a reference timestamp for a specific source, a user can

To enable proper query of a reference timestamp for a specific source, a user can obtain a current timestamp using MPI\_T\_SOURCE\_GET\_TIMESTAMP. The argument

source\_index identifies the index of the source to query. The call returns MPI\_SUCCESS and
 a current timestamp in the argument timestamp if the source supports ad-hoc generation of
 timestamps. The call returns MPI\_T\_ERR\_INVALID\_INDEX if the index does not identify a
 valid source. The call returns MPI\_T\_ERR\_NOT\_SUPPORTED if the source does not support
 the ad-hoc generation of timestamps.

#### <sup>24</sup> <sub>25</sub> Callback Safety Requirements

The actions a user is allowed to perform inside a callback function may vary with its execution context. As the user has no control over the execution context of specific callback function invocations, MPI provides a way to communicate this information using callback safety levels.

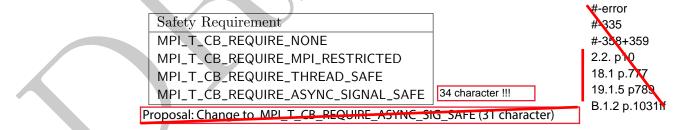


Table 15.5: Hierarchy of safety requirement levels for event callback routines.

Table 15.5 provides the hierarchy of callback safety requirements levels within userdefined callback functions. The MPI implementation provides the safety requirement as an argument to the callback when it is invoked.

<sup>42</sup> The level of MPI\_T\_CB\_REQUIRE\_NONE is the lowest level and does not impose any <sup>43</sup> restrictions on the callback function.

The level of MPI\_T\_CB\_REQUIRE\_MPI\_RESTRICTED restricts the set of MPI functions
 that can be called from inside the callback to all functions with the prefix MPI\_T as well as
 MPI\_WTICK and MPI\_WTIME.

- 47
- 48

Advice to users. While some MPI functions are safe to be called inside a callback

function used in the MPI tool information interface—which may in some implementations be issued from asynchronous contexts such as signal handlers—this does not imply that those MPI functions are generally safe to be called in asynchronous contexts such as signal handlers. (End of advice to users.)

The level of MPI\_T\_CB\_REQUIRE\_THREAD\_SAFE includes all the limitations of MPI\_T\_CB\_REQUIRE\_MPI\_RESTRICTED and additionally requires the callback to be reentrant and thread-safe. This means the callback must allow its execution to be interrupted by or happen concurrently with any other callback including itself.

The level of MPI\_T\_CB\_REQUIRE\_ASYNC\_SIGNAL\_SAFE includes all the limitations of MPI\_T\_CB\_REQUIRE\_THREAD\_SAFE and additionally requires the callback to meet the safety requirements needed to support invocations from asynchronous contexts, such as signal handlers.

Advice to users. It is always safe to assume the highest restrictions for a callback invocation (i.e., MPI\_T\_CB\_REQUIRE\_ASYNC\_SIGNAL\_SAFE). By evaluating the specific requirements at runtime, a tool may obtain more freedom of action within the callback. (End of advice to users.)

A high-quality implementation will strive to set callback Advice to implementors. safety requirements to the most permissive level for a given callback invocation. (End of advice to implementors.)

All functions with the prefix MPI\_T, except those listed in Table 15.6, may return the error code MPI\_T\_ERR\_NOT\_ACCESSIBLE to indicate that the user may not access this function at this time.

MPI_T_EVENT_COPY	PMPI_T_EVENT_COPY	
MPI_T_EVENT_GET_SOURCE	PMPI_T_EVENT_GET_SOURCE	
MPI_T_EVENT_GET_TIMESTAMP	PMPI_T_EVENT_GET_TIMESTAMP	
MPI_T_EVENT_READ	PMPI_T_EVENT_READ	
MPI_T_PVAR_READ	PMPI_T_PVAR_READ	
MPI_T_PVAR_READRESET	PMPI_T_PVAR_READRESET	
MPI_T_PVAR_RESET	PMPI_T_PVAR_RESET	
MPI_T_PVAR_START	PMPI_T_PVAR_START	
MPI_T_PVAR_STOP	PMPI_T_PVAR_STOP	
MPI_T_PVAR_WRITE	PMPI_T_PVAR_WRITE	
MPI_T_SOURCE_GET_TIMESTAMP	PMPI_T_SOURCE_GET_TIMESTAMP	

Table 15.6: List of MPI functions that when called from within a callback function may not return MPI\_T\_ERR\_NOT\_ACCESSIBLE.

A call may be implemented in a way that is not safe for all execution Rationale. contexts of a callback function, e.g., inside a signal handler. An MPI implementation therefore needs a way to communicate its inability to perform a certain action due to the execution context of a callback invocation. (End of rationale.)

2

#-error

-335 358 + 3

1 p.

9 B.1.2 p.1031ff

.5 p789

8 19.

34 charactel !!!

11

12

13

17

18 19

20

21

22 23

 $^{24}$ 

25

39

44

45

46

Advice to implementors. A high-quality implementation shall not return MPI\_T\_ERR\_NOT\_ACCESSIBLE except where absolutely necessary. (End of advice to *implementors.*) Advice to users. Users intercepting calls into the MPI tool information interface using the PMPI interface must ensure that the safety requirements for the calling context are met. This means that users may have to implement the wrapper with the highest safety level used by the MPI implementation. (End of advice to users.) Event Type Query Functions An MPI implementation exports a set of N event types through the MPI tool information interface. If N is zero, then the MPI implementation does not export any event types; otherwise, the provided event types are indexed from 0 to N-1. This index number is used in subsequent calls to identify a specific event type. An MPI implementation is allowed to increase the number of event types during the execution of an MPI process. However, MPI implementations are not allowed to change the index of an event type or to delete an event type once it has been made visible to the user (e.g., if new event types become available via dynamic loading of additional components in the MPI implementation). The following function can be used to query the number of event types, *num\_events*: MPI\_T\_EVENT\_GET\_NUM(num\_events)  $^{24}$ OUT returns number of event types (integer) num\_events C binding int MPI\_T\_event\_get\_num(int \*num\_events) The function MPI\_T\_EVENT\_GET\_INFO provides access to additional information about a specific event type.  $^{31}$ 

MPI\_T\_EVENT\_GET\_INFO(event\_index, name, name\_len, verbosity, array\_of\_datatypes, array\_of\_displacements, num\_elements, enumtype, info, desc, desc\_len, bind)

IN	event_index	index of the event type to be queried between 0 and $num\_events - 1$ (integer)	4 5 6
OUT	name	buffer to return the string containing the name of the event type (string)	7 8
INOUT	name_len	length of the string and/or buffer for name (integer) $% \left( {{\left[ {{{\rm{A}}} \right]}_{{\rm{A}}}}} \right)$	9
OUT	verbosity	verbosity level of this event type (integer)	10 11
OUT	array_of_datatypes	array of MPI basic datatypes used to encode the event data (array of handles)	11 12 13
OUT	array_of_displacements	array of byte displacements of the elements in the event buffer (array of non-negative integers)	14 15
INOUT	num_elements	length of array_of_datatypes and array_of_displacements arrays (non-negative integer)	16 17 18
OUT	enumtype	optional descriptor for enumeration information (handle)	19 20
OUT	info	optional info object (handle)	21
OUT	desc	buffer to return the string containing a description of the event type (string)	22 23
INOUT	desc_len	length of the string and/or buffer for desc (integer)	24 25
OUT	bind	type of MPI object to which an event of this type	26
-		must be bound (integer)	27
			28

#### C binding

int MPI_T_eve	<pre>ent_get_info(int event_index, char *name, int *name_len,</pre>
	<pre>int *verbosity, MPI_Datatype array_of_datatypes[],</pre>
	MPI_Aint array_of_displacements[], int *num_elements,
	<pre>MPI_T_enum *enumtype, MPI_Info *info, char *desc,</pre>
	<pre>int *desc_len, int *bind)</pre>

After a successful call to MPI\_T\_EVENT\_GET\_INFO for a particular event type, subsequent calls to this routine that query information about the same event type must return the same information. If any INOUT or OUT argument to MPI\_T\_EVENT\_GET\_INFO is a NULL pointer, the implementation will ignore the argument and not return a value for the specific argument.

The arguments name and name\_len are used to return the name of the event type as described in Section 15.3.3. If completed successfully, the routine is required to return a name of at least length one. The name of the event type must be unique with respect to all other names for event types used by the MPI implementation.

The argument verbosity returns the verbosity level of the event type (see Section 15.3.1).

The argument array\_of\_datatypes returns an array of MPI datatype handles that de-46scribe the elements returned for an instance of the event type with index event\_index. The 47event data can either be queried element by element with MPI\_T\_EVENT\_READ or copied 48

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

1  $\mathbf{2}$ 

3

29

36

37

38

39

40

41

42

43

44

1 into a contiguous event buffer with MPI\_T\_EVENT\_COPY. For the latter case, the argu- $\mathbf{2}$ ment array\_of\_displacements returns an array of byte displacements in the event buffer in 3 ascending order starting with zero.

The user is responsible for the memory allocation for the array\_of\_datatypes and

 $\mathbf{5}$ array\_of\_displacements arrays. The number of elements in each array is supplied by the user 6 in num\_elements. If the number of elements used by the event type is larger than the value 7of num\_elements provided by the user, the number of datatype handles and displacements 8 returned in the corresponding arrays is truncated to the value of num\_elements passed in 9 by the user. If the user passes the NULL pointer for array\_of\_datatypes or

10 array\_of\_displacements, the respective arguments are ignored. Unless the user passes the 11NULL pointer for num\_elements, the function returns the number of elements required for 12this event type. If the user passes the NULL pointer for num\_elements, the arguments 13 num\_elements, array\_of\_datatypes, and array\_of\_displacements are ignored.

14MPI can optionally return an enumeration identifier in the enumtype argument, de-15scribing the individual elements in the array\_of\_datatypes argument. Otherwise, enumtype 16is set to MPI\_T\_ENUM\_NULL. If the argument to enumtype provided by the user is the 17MPI\_T\_ENUM\_NULL pointer, no enumeration type is returned.

18 MPI can optionally return an info object containing the default hints set for a regis-19tration handle for this event type. If the argument to info provided by the user is the NULL 20pointer, this argument is ignored, otherwise an MPI implementation is required to return 21all hints that are supported by the implementation for a registration handle for this event 22type and have default values specified; any user-supplied hints that were not ignored by the 23implementation; and any additional hints that were set by the implementation. If no such  $^{24}$ hints exist, a handle to a newly created info object is returned that contains no key/value 25pair. The user is responsible for freeing info via MPI\_INFO\_FREE.

26The arguments desc and desc\_len are used to return the description of the event type as 27described in Section 15.3.3. Returning a description is optional. If an MPI implementation 28does not return a description, the first character for desc must be set to the null character 29and desc\_len must be set to one at the return from this function.

30 The parameter bind returns the type of the MPI object to which the event type must  $^{31}$ be bound or the value MPI\_T\_BIND\_NO\_OBJECT (see Section 15.3.2).

32 If an event type has an equivalent name across connected MPI processes, the following 33 OUT parameters must be identical: verbosity, array\_of\_datatypes, num\_elements, enumtype, 34and bind. The returned description must be equivalent. As the argument

35 array\_of\_displacements is process dependent, it may differ across connected MPI processes. 36 This routine returns MPI\_SUCCESS on success and returns MPI\_T\_ERR\_INVALID\_INDEX 37 if event\_index does not match a valid event type index provided by the implementation at 38the time of the call.

39 40

41

MPI\_T\_EVENT\_GET\_INDEX(name, event\_index)

42	IN	name	name of the event type (string)
43 44	OUT	event_index	index of the event type (integer)
45			

C binding 46

```
int MPI_T_event_get_index(const char *name, int *event_index)
47
48
```

MPI\_T\_EVENT\_GET\_INDEX returns the index of an event type identified by a known event type name. The name parameter is provided by the caller, and event\_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 event type provided by the implementation at the time of the call.

*Rationale.* This routine is provided to enable fast retrieval of an event index by a tool, assuming it knows the name of the event type for which it is looking. The number of event types exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of event types once at initialization. Although using MPI implementation specific event type 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 event types to find a specific one. (*End of rationale.*)

## Handle Allocation and Deallocation

Before the MPI implementation calls a callback function on the occurrence of a specific event, the user needs to register a callback function to be called for that event type and obtain a handle of type MPI\_T\_event\_registration.

MPI_T_I	EVENT_HANDLE_ALLOC(event	index, obj_handle, info, event_registration)	24
IN	event_index	index of event type for which the registration handle	25 26
		is to be allocated (integer)	27
IN	obj_handle	reference to a handle of the MPI object to which this	28
		event is supposed to be bound (pointer)	29
IN	info	info object (handle)	30
OUT	event_registration	event registration (handle)	31
001	event_registration	event registration (nandie)	32
			33

## C binding

MPI\_T\_EVENT\_HANDLE\_ALLOC creates a *registration handle* for the event type identified by event\_index. Furthermore, if required by the event type, the registration handle is bound to the object referred to by the argument obj\_handle. The argument obj\_handle is ignored if the MPI\_T\_EVENT\_GET\_INFO call for this event type returned MPI\_T\_BIND\_NO\_OBJECT in the argument bind. The user can pass hints for the handle allocation to the MPI implementation via the info argument. The allocated event-registration handle is returned in the argument event\_registration.

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

14

15

16 17

18 19

20

21

22 23

34

35

36 37

38

39

40

41

42

43

<sup>1</sup> N	1PI_T_EV	ENT_HANDLE_SET_	NFO(event_registra	ation, info)	
2 3	IN	event_registration	event regis	stration (handle)	
1	IN	info	info object	t (handle)	
	C binding nt MPI_T	_event_handle_set_		t_registration, MPI_Info i	<b>~f</b> ~)
$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{array}$	le associa <sup>.</sup> o effect or ffect on pr	EVENT_HANDLE_S ted with event_registrant previously set or de reviously set or defaul	ET_INFO updates ation using the hin faulted hints that a ted hints that are s	the hints of the event-registrat ats provided in info. This opera are not specified by info. It als specified by info, but are ignore "_HANDLE_SET_INFO.	ion han- ition has o has no
5 6 7 8 9 0 1 2	an ev is cre would unabl MPI_	vent-registration hand eated. Thus, an imp l have accepted in an le to update certain in	le cannot easily be lementation may handle allocation fo hints in a call to GET_INFO can be	implementation can use when it e changed once the registration ignore hints issued in this call a call. An implementation may o MPI_T_EVENT_HANDLE_SE used to determine whether info f advice to users.)	h handle l that it r also be T_INFO.
3					
4 5 <b>№</b>	1PI_T_EV	ENT_HANDLE_GET_	INFO(event_registr	ration, info_used)	
6 7	IN	event_registration	event regis	stration (handle)	
8	OUT	info_used	info object	t (handle)	
	C binding nt MPI_T	_event_handle_get_	gistration even	t_registration,	
$egin{array}{ccc} & & & & & & & & & & & & & & & & & &$	ne event-r ints relate required alues spec ny additic o a newly	egistration handle ass ed to this registration to return all hints th ified; any user-supplie onal hints that were so	ociated with event handle is returned at are supported b ed hints that were n et by the implement s returned that co	new info object containing the t_registration. The current settid in info_used. An MPI implement of the implementation and have not ignored by the implementation. If no such hints exist, a contains no key/value pairs. The EE.	ng of all entation e default ion; and a handle
8					

	EVENT_REGISTER_CALLE	ACK (event_registration, cb_safety, into, user_data,	-
	$event_cb_function)$		2
IN	event_registration	event registration (handle)	3
			4
IN	cb_safety	maximum callback safety level (integer)	5
IN	info	info object (handle)	6
			7
IN	user_data	pointer to a user-controlled buffer	8
IN	event_cb_function	pointer to user-defined callback function (function)	9
			10

# MPLT EVENT REGISTER CALLBACK (event registration ch safety info user data

## C binding

int MPI\_T\_event\_register\_callback(

MPI\_T\_event\_registration event\_registration, MPI\_T\_cb\_safety cb\_safety, MPI\_Info info, void \*user\_data, MPI\_T\_event\_cb\_function event\_cb\_function)

MPI\_T\_EVENT\_REGISTER\_CALLBACK associates a user-defined function pointed to by event\_cb\_function with an allocated event-registration handle. The maximum callback safety level supported by the callback function is passed in the argument cb\_safety. The safety levels are defined in Table 15.5. A user can register multiple callback functions for a given event-registration handle, potentially specifying one for each callback safety level. Registering a callback function for a specific callback safety level overwrites any previously registered callback function pointer and info object associated with the event registration for the specific callback safety level. If event\_cb\_function is the NULL pointer, the association of a callback function for that callback safety level is removed.

When an event is triggered, the implementation will select from all registered callbacks the callback with the lowest safety level valid in the context in which the callback is invoked. In situations where the required callback safety level exceeds the highest level for which a callback function is registered for a given registration handle, the event instance is dropped.

At invocation time, the implementation passes the pointer to a user-defined memory region specified during callback registration with the argument user\_data.

The user can pass hints for the registration of the specified callback function to the MPI implementation via the info argument.

Advice to users. As event instances can be raised as soon as the registration handle is associated with the first callback function, the callback function with the highest callback safety guarantees should be registered before any further registrations for lower callback safety guarantees, to avoid dropped events due to insufficient callback safety guarantees. (End of advice to users.)

The callback function passed to MPI\_T\_EVENT\_REGISTER\_CALLBACK in the argument event\_cb\_function needs to have the following type:

```
typedef void (*MPI_T_event_cb_function)(
                        MPI_T_event_instance event_instance,
                        MPI_T_event_registration event_registration,
                        MPI_T_cb_safety cb_safety,
                                                             #-TODO
                        void *user_data);
```

After the merge of PR409, the remaining \typedef.. must be checked and may be removed.

**Unofficial Draft for Comment Only** 

11

12

13

14

1516

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

4243

44

1 2 3 4 5 6 7 8 9 10 11 12	object of invocation to the even function to identify th even to de describes to The argun	type MPI_T_event_instance. T of the function to which it is p at-registration handle returned to the same event type and bou e specific event registration in allocate the handle from within the safety requirements the call	sponds to a handle for the opaque event-instance this handle is only valid inside the corresponding assed. The argument event_registration corresponds d by MPI_T_EVENT_HANDLE_ALLOC for the user and object combination. The handle can be used to formation, such as event type and bound object, or in the callback invocation. The argument cb_safety lback function must fulfill in the current invocation. o user-allocated memory that was passed to the MPI tion.		
13	MPI_T_E\	/ENT_CALLBACK_SET_INFO	(event_registration, cb_safety, info)		
14	IN	event_registration	event registration (handle)		
15	IN	cb_safety	callback safety level (integer)		
16 17	IN	info	info object (handle)		
18					
20 21 22 23 24 25 26 27 28 29	C binding int MPI_T_event_callback_set_info(				
30 21					
31 32			(event_registration, cb_safety, info_used)		
33	IN	event_registration	event registration (handle)		
34	IN	cb_safety	callback safety level (integer)		
35 36	OUT	info_used	info object (handle)		
37 38 39 40	C binding int MPI_T_event_callback_get_info(				
41 42 43 44 45 46 47 48	of the call cb_safety of setting of returned i supported	EVENT_CALLBACK_GET_INF black function registered for to of the event-registration hand all hints related to this callban n info_used. An MPI implem by the implementation and ha	••••••••••••••••••••••••••••••••••••••		

1 the implementation. If no such hints exist, a handle to a newly created info object is  $\mathbf{2}$ returned that contains no key/value pairs. The user is responsible for freeing info\_used via 3 MPI\_INFO\_FREE. 4 To stop the MPI implementation from raising events for a specific registration, a user needs to free the corresponding event-registration handle. 56 7 MPI\_T\_EVENT\_HANDLE\_FREE(event\_registration, user\_data, free\_cb\_function) 9 IN event\_registration event registration (handle) 10 IN user\_data pointer to a user-controlled buffer 11 IN free\_cb\_function pointer to user-defined callback function (function) 1213 C binding 14int MPI\_T\_event\_handle\_free(MPI\_T\_event\_registration event\_registration, 15void \*user\_data, 16MPI\_T\_event\_free\_cb\_function free\_cb\_function) 1718 MPI\_T\_EVENT\_HANDLE\_FREE returns MPI\_SUCCESS when deallocation of the handle 19was initiated successfully and returns MPI\_T\_ERR\_INVALID\_HANDLE if 20event\_registration does not match a valid allocated event-registration handle at the time 21of the call. The callback function free\_cb\_function is called by the MPI implementation, 22when it is able to guarantee that no further event instances for the corresponding event-23registration handle will be raised. If the pointer to free\_cb\_function is the NULL pointer, no  $^{24}$ user function is invoked after successful deallocation of the event registration handle. The 25pointer to user-controlled memory provided in the user\_data argument will be passed to the 26function provided in the free cb function on invocation. 27Advice to users. A free-callback function associated with a registration handle should 2829always be prepared to postpone any pending actions, should the provided callback safety requirements exceed those required by the pending actions. (End of advice to 30 31users.) 32 The callback function passed to MPI\_T\_EVENT\_HANDLE\_FREE in the argument 33 free\_cb\_function needs to have the following type: 34 35typedef void (\*MPI\_T\_event\_free\_cb\_function)( 36 MPI\_T\_event\_registration event\_registration, 37 MPI\_T\_cb\_safety cb\_safety, #-TODO void \*user\_data); After the merge of PR409,

## the remaining \typedef.. must be checked and may be removed.

## Handling Dropped Events

Events may occur at times when the MPI implementation cannot invoke the user function <sup>42</sup> corresponding to a matching event handle. An implementation is allowed to buffer such <sup>43</sup> events and delay the callback invocation. If an event occurs at times when the corresponding <sup>44</sup> callback function cannot be called and the corresponding data cannot be buffered, or no <sup>45</sup> callback function meeting the required callback safety level is registered, the event data may <sup>46</sup> be dropped. To discover such data loss, the user can set a handler function for a specific <sup>47</sup> event-registration handle. <sup>48</sup>

## Unofficial Draft for Comment Only

	100			
1	ΜΡΙ Τ Ε΄	VENT SET DROPPED HAN	DLER(event_registration, dropped	l cb function)
2	IN	event_registration	valid event registration (handle)	
3	IN	dropped_cb_function	pointer to user-defined callback	function (function)
4 5	IIN	dropped_cb_function	pointer to user-defined canback	function (function)
6	C bindin	lg		
7		T_event_set_dropped_hand	ler(	
8		0	ation event_registration,	
9		MPI_T_event_dropped	_cb_function dropped_cb_fun	ction)
10 11	MPI_	T_EVENT_SET_DROPPED_	HANDLER registers the function	n
12			he MPI implementation when e	
13			ecified in event_registration. Sub-	-
14			DLER with the same registration s for that registration handle. If	-
15	* 0	0	er, no data loss is recorded or re	-
16 17		back function is registered.		
18	1 .1	···· ···· ···· ··· ··· ··· ····		
19			n of the dropped handler callbac was actually lost. ( <i>End of advic</i>	
20	0000	ir close to the third the event	was actually lost. (Live of device	
21		-	MPI_T_EVENT_SET_DROPPED	-HANDLER in the
22 23	argument	dropped_cb_function needs to	b have the following type:	
23	typedef	void (*MPI_T_event_dropp	ed_cb_function)(int count,	
25		MPI_T_ev	ent_registration event_regis	stration,
26			ce_index,	#-TODO
27			_safety cb_safety, er_data);	After the merge of PR409, the remaining \typedef must be
28 29		VOIU #US	er_data),	checked and may be removed.
30			prresponds to the event registration	
31			rgument count provides a best	
32			ed event callback corresponding t stration of the dropped-callback	
33			back handler. The source_index	
34 35			bonding event information. The	-
36			allback function must fulfill in the	
37	-		lescribed in Table 15.5. The arg	
38	-	egistration.	that was passed to the MPI imp	during
39	canoack i	CSISTINIOII.		
40 41			ction for dropped events associa	0
42			epared to postpone any pending	
43	-	d of advice to users.)	nents exceed those required by the	ne pending actions.
44	× ×	·		
45		_	igh-quality implementation will	
46 47			ents associated with a registrati	
48		lementors.)	action to the function as possible	Enta of advice to
	unip			

CHAPTER 15. TOOL SUPPORT

## Unofficial Draft for Comment Only

If events are dropped for a specific source, the corresponding handler callback function must be called before other events are raised for this source. This means in a sequence of five events E1 to E5 from the same source, where E3 and E4 were dropped, any handler function set through MPI\_T\_EVENT\_SET\_DROPPED\_HANDLER for event-registration handles associated with E3 or E4 must be called before E5 is raised.

### Reading Event Data

In event callbacks, the parameter event\_instance provides access to the per-instance event data, i.e., the data encoded by the specific event type for this instance. The user can obtain event data as well as event meta data, such as a time stamp and the source, by providing this handle to the respective query functions. The event-instance handle is invalid beyond the scope of the current invocation of the callback function to which it is provided.

The callback function argument event\_registration identifies the registration handle that was used to register the callback function.

The callback function argument cb\_safety indicates the requirements for the specific callback invocation. The value is one of the safety requirements levels described in Table 15.5. The argument user\_data passes the pointer provided by the user during callback registration back to the function call.

Advice to users. Depending on the registered event and usage of MPI by the application, a callback function may be invoked with high frequency. Users should therefore strive to minimize the amount of work done inside callback functions. Furthermore, the time spent in a callback function may influence the capability of an implementation to buffer events and long execution times may lead to an increased number of dropped events. (*End of advice to users.*)

MPI provides the following function calls to access data of a specific event instance and its corresponding meta data (such as its time and source).

MPI_T_EVENT	_READ(event	_instance,	element_	_index,	buffer)
-------------	-------------	------------	----------	---------	---------

	·	,	32
IN	event_instance	event-instance handle provided to the callback function (handle)	33
		function (nancie)	34
1N	element_index	index into the array of datatypes of the item to be	35
		queried (integer)	36
OUT	buffer	pointer to a memory location to store the item data	37
		(choice)	38
			39
~			40

## C binding

MPI\_T\_EVENT\_READ allows users to copy one element of the event data to a user-specified buffer at a time.

The event\_instance argument identifies the event instance to query. It is erroneous 46 to provide any other event-instance handle to the call than the one passed by the MPI 47 implementation to the callback function in which the data is read. The buffer argument 48

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 

1 must point to a memory location the MPI implementation can copy the element of the event  $\mathbf{2}$ data to identified by element\_index. 3 4 MPI\_T\_EVENT\_COPY(event\_instance, buffer) 56 IN event instance provided to the callback function event\_instance 7 (handle) 8 OUT buffer user-allocated buffer for event data (choice) 9 10 C binding 11 int MPI\_T\_event\_copy(MPI\_T\_event\_instance event\_instance, void \*buffer) 1213 MPI\_T\_EVENT\_COPY copies the event data as a whole into the user-provided buffer. 14The user must assure that the buffer is of at least the size of the extent of the event 15type, which can be computed from the type and displacement information returned by 16the corresponding call to MPI\_T\_EVENT\_GET\_INFO. The data may include padding bytes 17between individual elements of the event data in the buffer. A user can reconstruct the 18location and size of the data contained in the buffer through the information returned by 19MPI\_T\_EVENT\_GET\_INFO. 20Advice to implementation. An implementation should strive to use an appropriately 2122compact representation when copying event instance data to a user buffer via 23MPI\_T\_EVENT\_COPY to reduce the amount of memory required for the user buffer.  $^{24}$ (End of advice to implementors.) 2526Reading Event Meta Data 27Additional to the specific event data encoded by each event type, supplemental information 28available across all event types can be queried. 29 30 31MPI\_T\_EVENT\_GET\_TIMESTAMP(event\_instance, event\_timestamp) 32 IN event\_instance event instance provided to the callback function 33 (handle) 3435OUT event\_timestamp timestamp the event was observed (integer) 36 37 C binding 38 int MPI\_T\_event\_get\_timestamp(MPI\_T\_event\_instance event\_instance, 39 MPI\_Count \*event\_timestamp) 40 MPI\_T\_EVENT\_GET\_TIMESTAMP returns the timestamp of when the event was ini-41 tially observed by the implementation. The event\_instance argument identifies the event 42instance to query. It is erroneous to provide any other handle to the call than the one 43 passed by the MPI implementation to the callback function in which the timestamp is read. 4445Advice to users. An MPI implementation may postpone the call to the user's callback 46function. In this case, the call to MPI\_T\_EVENT\_GET\_TIMESTAMP may yield a 47 timestamp in the past that is closer to the time the event was initially observed, as 48

opposed to a timestamp captured during callback function invocation. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will return a timestamp as close as possible to the earliest time the event was observed by the MPI implementation. (*End of advice to implementors.*)

An event may be raised from different components acting as event sources in the MPI implementation. A source in this context is an abstract concept that helps to define partial ordering of raised events, as each source provides its own ordering guarantees. A source describes the entity that raises the event, rather than the origin of the data.

To identify the source of an event instance, the user can query the index of the source within the corresponding event callback function invocation.

Advice to implementors. An excessive number of event sources may negatively impact performance of a tool due to per-source overhead in event handling. (End of advice to implementors.)

MPI_T_E	VENT_GET_SOURCE(	event_instance, source_index)
IN	event_instance	event instance provided to the callback function
		(handle)
OUT	source_index	index identifying the source (integer)

## C binding

The event\_instance argument identifies the event instance to query. It is erroneous to provide any other event-instance handle to the call than the one passed by the MPI implementation to the callback function in which the source is queried.

The source\_index argument returns the index of the source of the event instance. It can be used to query more information on the source using MPI\_T\_SOURCE\_GET\_INFO.

*Rationale.* Event callback function invocations are associated with a source to enable chronological processing of events on the tool side, when required, while retaining low overhead on the side of the MPI implementation. (*End of rationale.*)

## 15.3.9 Variable Categorization

MPI implementations can optionally group performance and control variables into categories to express logical relationships between various variables. For example, an MPI implementation could group all control and performance variables that refer to message transfers in the MPI implementation and thereby distinguish them from variables that refer to local resources such as memory allocations or other interactions with the operating system.

Categories can also contain other categories to form a hierarchical grouping. Categories <sup>46</sup> can never include themselves, either directly or transitively within other included categories. <sup>47</sup> Expanding on the example above, this allows MPI to refine the grouping of variables referring <sup>48</sup>

### Unofficial Draft for Comment Only

 $^{31}$ 

to message transfers into variables to control and to monitor message queues, message
 matching activities and communication protocols. Each of these groups of variables would
 be represented by a separate category and these categories would then be listed in a single
 category representing variables for message transfers.

<sup>5</sup> The category information may be queried in a fashion similar to the mechanism for <sup>6</sup> querying variable information. The MPI implementation exports a set of N categories via <sup>7</sup> the MPI tool information interface. If N = 0, then the MPI implementation does not export <sup>8</sup> any categories, otherwise the provided categories are indexed from 0 to N - 1. This index <sup>9</sup> number is used in subsequent calls to functions of the MPI tool information interface to <sup>10</sup> identify the individual categories.

<sup>11</sup> An MPI implementation is permitted to increase the number of categories during the <sup>12</sup> execution of an MPI program when new categories become available through dynamic load-<sup>13</sup> ing. However, MPI implementations are not allowed to change the index of a category or <sup>14</sup> delete it once it has been added to the set.

<sup>15</sup> Similarly, MPI implementations are allowed to add variables to categories, but they
 <sup>16</sup> are not allowed to remove variables from categories or change the order in which they are
 <sup>17</sup> returned.

<sup>19</sup> Category Query Functions

The following function can be used to query the number of categories, num\_cat.

## MPI\_T\_CATEGORY\_GET\_NUM(num\_cat)

OUT num\_cat current number of categories (integer)

## C binding

- int MPI\_T\_category\_get\_num(int \*num\_cat)
- 29 30 31

18

20

21 22 23

24

25 26

27

Individual category information can then be queried by calling the following function:

MPI_T_CA	ATEGORY_GET_INFO(c num_pvars, num_c	cat_index, name, name_len, desc, desc_len, num_cvars, categories)	$\frac{1}{2}$		
IN	cat_index	index of the category to be queried (integer)	$\frac{3}{4}$		
OUT	name	buffer to return the string containing the name of the category (string)	4 5 6		
INOUT	name_len	length of the string and/or buffer for name (integer)	7		
OUT	desc	buffer to return the string containing the description of the category (string)	8 9 10		
INOUT	desc_len	length of the string and/or buffer for desc (integer)	10		
OUT	num_cvars	number of control variables in the category (integer)	12		
OUT	num_pvars	number of performance variables in the category (integer)	13 14 15		
OUT	num_categories	number of categories contained in the category (integer)	15 16 17		
			18		
C binding		int art index about the int them. The	19		
int MP1_1		<pre>int cat_index, char *name, int *name_len,  *desc_len, int *num_cvars, int *num_pvars,</pre>	20 21		
	int *num_catego		22		
The a	rguments <b>name</b> and <b>n</b> a	mme_len are used to return the name of the category as	23		
	in Section $15.3.3$ .		24 25		
	The routine is required to return a name of at least length one. This name must be $_{26}$				
-	unique with respect to all other names for categories used by the MPI implementation. If any OUT parameter to MPI_T_CATEGORY_GET_INFO is the NULL pointer, the im-				
ě	-	meter and not return a value for the parameter.	28		
•		_len are used to return the description of the category as	29		
described	in Section 15.3.3.		30 31		
		tional. If an MPI implementation decides not to return a	32		
-	n, the first character for one at the return of this	desc must be set to the null character and desc_len must	33		
		ber of control variables, performance variables and other	34		
		ed category in the arguments num_cvars, num_pvars, and	35		
	ories, respectively.		$\frac{36}{37}$		
		equivalent across connected MPI processes, then the re-	38		
turned des	cription must be equive	llent.	39		
			40		
MPI_T_CA	ATEGORY_GET_NUM_I	EVENTS(cat_index, num_events)	41		
IN	cat_index	index of the category to be queried (integer)	42 43		
OUT	num_events	number of event types in the category (integer)	43		
			45		
C binding			46		
int MPI_7	C_category_get_num_e	vents(int cat_index, int *num_events)	47		
			48		

## **Unofficial Draft for Comment Only**

MPI_T_C	ATEGORY_GET_INI	DEX(name, cat_index)
IN	name	the name of the category (string)
OUT	cat_index	the index of the category (integer)
C bindir int MPI_	-	dex(const char *name, int *cat_index)
given a kı	nown category name. d by the MPI implen	<b>INDEX</b> is a function for retrieving the index of a cate The name parameter is provided by the caller, and cat_in mentation. The name parameter is a string terminated with
This	routine returns MPI_ oes not match the n	_SUCCESS on success and returns MPI_T_ERR_INVALID_N ame of any category provided by the implementation at
by a num poss AltI MP at r	a tool, assuming it k aber of categories exp sible for the tool to s nough using MPI im I implementations, to	ine is provided to enable fast retrieval of a category in mows the name of the category for which it is looking. bosed by the implementation can change over time, so it is simply iterate over the list of categories once at initializat plementation specific category names is not portable ac bol developers may choose to take this route for lower overh tool will not have to iterate over the entire set of categor End of rationale.)
	Member Query Funct	
MPI_T_C	ATEGORY_GET_CV	ARS(cat_index, len, indices)
IN	cat_index	index of the category to be queried, in the range and $num_cat - 1$ (integer)
IN	len	the length of the indices array (integer)
OUT	indices	an integer array of size len, indicating control variable indices (array of integers)
C bindir	•	ars(int cat_index, int len, int indices[])
int MPI_		

MPI_T_CA	ATEGORY_GET_PVARS(cat_in	dex, len, indices)	1		
IN	cat_index	index of the category to be queried, in the range 0 and num_cat $-1$ (integer)	2 3 4		
IN	len	the length of the indices array (integer)	4 5		
OUT	indices	an integer array of size len, indicating performance	6		
		variable indices (array of integers)	7		
			8 9		
C binding	-	cat_index, int len, int indices[])	10 11		
are contai		an be used to query which performance variables A category contains zero or more performance	11 12 13		
variables.			14 15		
MPI_T_CA	ATEGORY_GET_EVENTS(cat_	index, len, indices)	16 17		
IN	cat_index	index of the category to be queried, in the range 0 and $num\_cat - 1$ (integer)	18 19		
IN	len	the length of the indices array (integer)	20		
OUT	indices	an integer array of size len, indicating event type indices (array of integers)	21 22 23		
C bindin	-	<pre>cat_index, int len, int indices[])</pre>	24 25 26		
MPI_T_CATEGORY_GET_EVENTS can be used to query which event types are con- tained in a particular category. A category contains zero or more event types.					
MPI_T_CA	ATEGORY_GET_CATEGORIES	(cat_index, len, indices)	30 31		
IN	cat_index	index of the category to be queried, in the range 0 and num_cat - 1 (integer)	32 33		
IN	len	the length of the indices array (integer)	34		
Ουτ	indices	an integer array of size len, indicating category	35 36		
		indices (array of integers)	37		
			38		
C binding	-	(int and index int law int indiana[])	39 40		
		<pre>int cat_index, int len, int indices[])</pre>	41		
		<b>RIES</b> can be used to query which other categories category contains zero or more other categories.	42		
	* 0 0	tations can grow the number of categories as well	43 44		
	· –	egories within a category. In order to allow users	44 45		
		check quickly whether new categories have been	46		
added or new variables or categories have been added to a category, MPI maintains a $_{47}$					

## Unofficial Draft for Comment Only

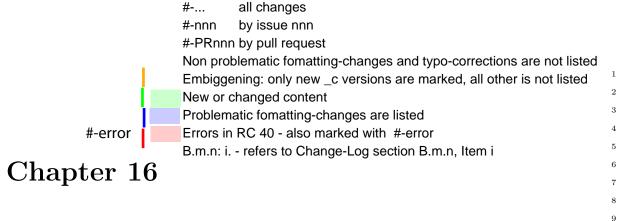
1 virtual timestamp. This timestamp is monotonically increasing during the execution and is  $\mathbf{2}$ returned by the following function: 3 4 MPI\_T\_CATEGORY\_CHANGED(stamp) 56 OUT a virtual time stamp to indicate the last change to stamp 7 the categories (integer) 8 9 C binding 10 int MPI\_T\_category\_changed(int \*stamp) 11 If two subsequent calls to this routine return the same timestamp, it is guaranteed that 12the category information has not changed between the two calls. If the timestamp retrieved 13 from the second call is higher, then some categories have been added or expanded. 1415Advice to users. The timestamp value is purely virtual and only intended to check 16for changes in the category information. It should not be used for any other purpose. 17 (End of advice to users.) 18 The index values returned in indices by MPI\_T\_CATEGORY\_GET\_CVARS, 19MPI\_T\_CATEGORY\_GET\_PVARS and MPI\_T\_CATEGORY\_GET\_CATEGORIES can be used 20as input to MPI\_T\_CVAR\_GET\_INFO, MPI\_T\_PVAR\_GET\_INFO and 21MPI\_T\_CATEGORY\_GET\_INFO, respectively. 22The user is responsible for allocating the arrays passed into the functions 23MPI\_T\_CATEGORY\_GET\_CVARS, MPI\_T\_CATEGORY\_GET\_PVARS and  $^{24}$ MPI\_T\_CATEGORY\_GET\_CATEGORIES. Starting from array index 0, each function writes 2526up to len elements into the array. If the category contains more than len elements, the function returns an arbitrary subset of size len. Otherwise, the entire set of elements is 27returned in the beginning entries of the array, and any remaining array entries are not 28modified. 2930 Return Codes for the MPI Tool Information Interface  $^{31}$ 15.3.10 32 All functions defined as part of the MPI tool information interface return an integer error 33 code (see Table 15.7) to indicate whether the function was completed successfully or was 34 aborted. In the latter case the error code indicates the reason for not completing the routine. 35 Such errors neither impact the execution of the MPI process nor invoke MPI error handlers. 36 The MPI process continues executing regardless of the return code from the call. The MPI 37 implementation is not required to check all user-provided parameters; if a user passes invalid 38 parameter values to any routine the behavior of the implementation is undefined. 39 All error codes with the prefix MPI\_T\_ must be unique values and cannot overlap with 40 any other error codes or error classes returned by the MPI implementation. Further, they 41 shall be treated as MPI error classes as defined in Section 9.4 and follow the same rules and 42restrictions. In particular, they must satisfy: 43 44 $0 = MPI_SUCCESS < MPI_T_ERR_XXX \le MPI_ERR_LASTCODE.$ 4546Rationale. All MPI tool information interface functions must return error classes, be-47 cause applications cannot portably call MPI\_ERROR\_CLASS before MPI initialization 48 to map an arbitrary error code to an error class. (*End of rationale.*)

## Unofficial Draft for Comment Only

## 15.3.11 Profiling Interface

All requirements for the profiling interfaces, as described in Section 15.2, also apply to the MPI tool information interface. All rules, guidelines, and recommendations from Section 15.2 apply equally to calls defined as part of the MPI tool information interface.

Return Code	Description
Return Codes for All Functions in the	he MPI Tool Information Interface
MPI_SUCCESS	Call completed successfully
MPI_T_ERR_INVALID	Invalid or bad parameter value(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
MPI_T_ERR_NOT_ACCESSIBLE	Requested functionality not accessible
Return Codes for Datatype Function	ns: MPLT FNUM *
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)
Boturn Codes for Variable Category	y, and Event Query Functions: MPI_T_*_GET_*
MPI_T_ERR_INVALID_INDEX	The variable or category index is invalid
MPI_T_ERR_INVALID_INDEX	The variable of 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 A	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	
	MPI_T_PVAR_START and MPI_T_PVAR_STOP)
MPI_T_ERR_PVAR_NO_WRITE	Variable cannot be written or reset (for
	MPI_T_PVAR_WRITE and MPI_T_PVAR_RESET)
MPI_T_ERR_PVAR_NO_ATOMIC	Variable cannot be read and written atomically (for
	MPI_T_PVAR_READRESET)
Return Codes for Source Functions:	, ,
MPI_T_ERR_INVALID_INDEX	The source index is invalid
MPI_T_ERR_NOT_SUPPORTED	Requested functionality not supported
Return Codes for Category Function	
MPI_T_ERR_INVALID_INDEX	The category index is invalid
Table 15.7: Return codes used in	n functions of the MPI tool information interface



# **Deprecated Interfaces**

#### Deprecated since MPI-2.0 16.1

Done in #-PR442

And finally done in PR449 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 19.3.7. The language bindings are modified.

MPI_KEYV	/AL_CREATE(copy_fn, delete_	fn, keyval, extra_state)	22	
IN	copy_fn	Copy callback function for keyval	23	
IN	delete_fn	Delete callback function for keyval	24 25	
OUT	keyval	key value for future access (integer)	25 26	
			27	
IN	extra_state	Extra state for callback functions	28	
			29	
C binding			30	
int MPI_K	eyval_create(MPI_Copy_fur		31	
		<pre>*delete_fn, int *keyval,</pre>	32	
	void *extra_state)		33	
For this rou	utine, an interface within the	<pre>mpi_f08 module was never defined.</pre>	34	
Fortran b	inding		35	
	<u> </u>	FN, KEYVAL, EXTRA_STATE, IERROR)	36	
	NAL COPY_FN, DELETE_FN		37	
	ER KEYVAL, EXTRA_STATE, 1	IERROR	38 39	
			40	
		hen a communicator is duplicated by type MPI_Copy_function, which is defined as follows:		#-error
In the function in		this space before the typdef is lost since RC-40		
		Comm oldcomm, int keyval,	43	#-PR442
oypodor 1		oid *attribute_val_in,	44	
	void *attribute_val_		45	(#-TODO) Don't
			46	worry!!!
	tran declaration for such a fur		47	
FOr this rol	utilie, an interface within the	mpi_f08 module was never defined.	48	

For this routine, an interface within the mpi\_f08 module was never defined.

1516

17

18

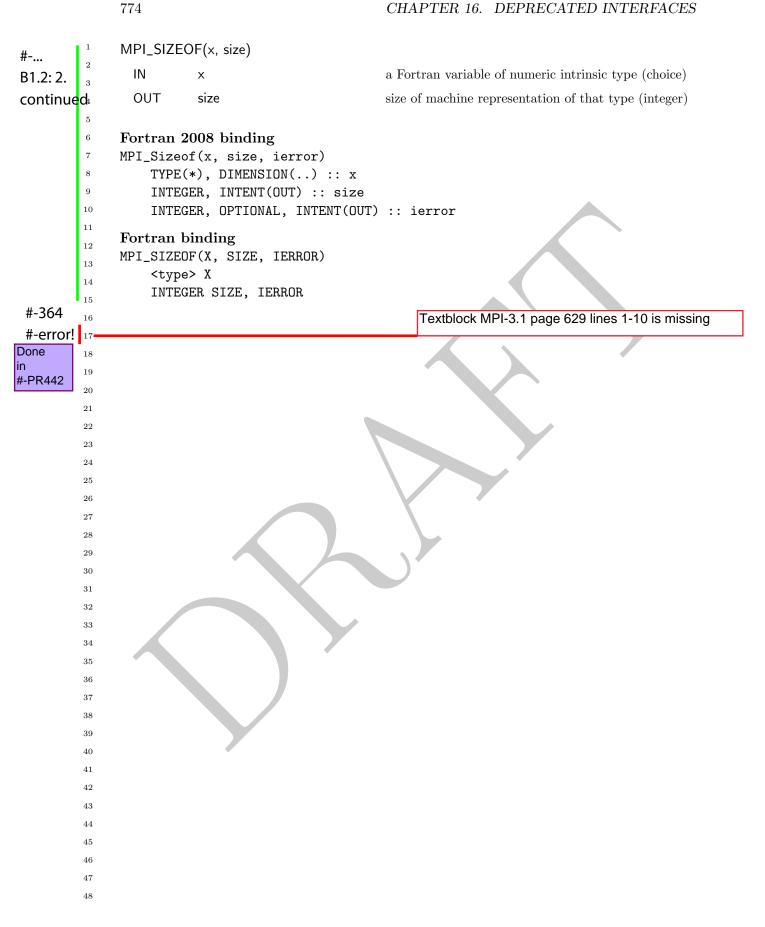
19

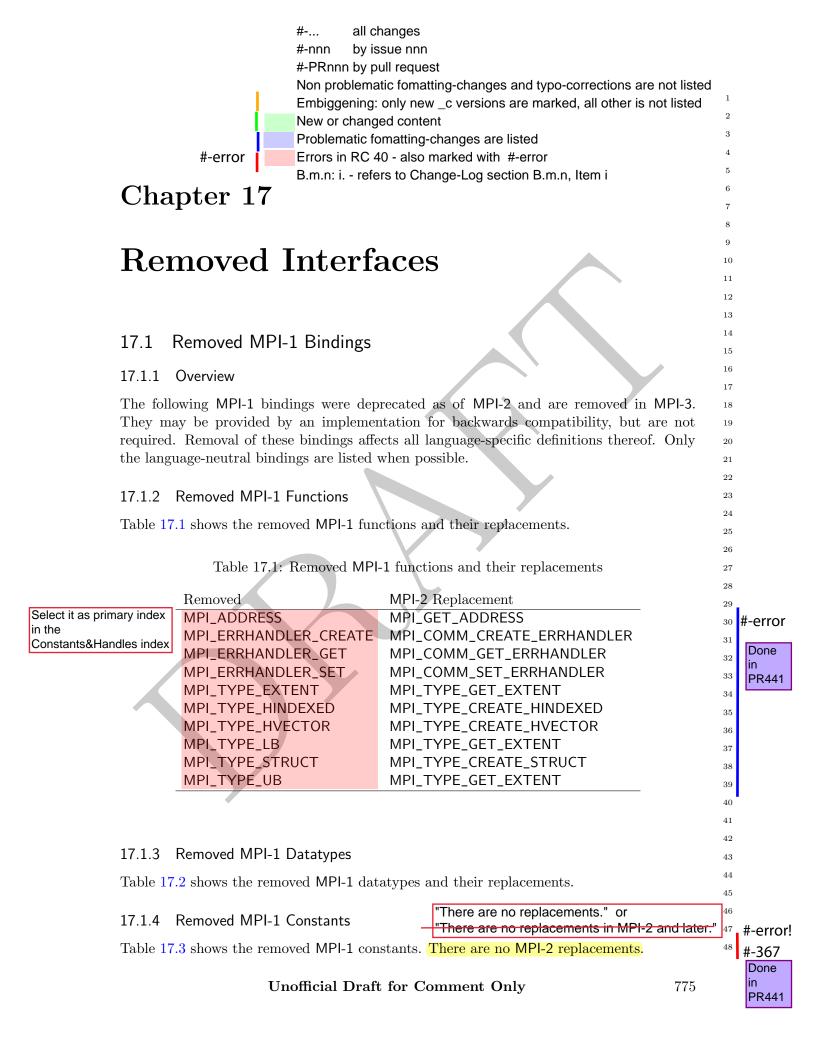
#-PR442       In the function index as primary         1       SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)         3       INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR         4       ATTRIBUTE_VAL_OUT, IERR         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 flag 0         9       0         10       Fortran; MPI_NULL_COPY_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note that MPI_NULL_COPY_FN are also deprecated.         10       Test \mbox()         13       function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: In the function index as primary tunedef int MPI_Delete function	Done in
1       SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)       Rolf: In MPI-3.1, the first location was completely on line 9 and the second location was line-broken. Bill: I will leave this as is.         4       ATTRIBUTE_VAL_OUT, IERR       Bill: I will leave this as is.         5       LOGICAL FLAG       Expression of this as is.         6       copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or Fortran; MPI_NULL_COPY_FN is a function that does nothing other than returning flag 0       no         7       returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated.       Modernet flag       no         9       10       function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: In the function index as primary tweedef int MPI_DElete function (MPI_COMM_FREE)       with is defined as follows: In the function index as primary	
<ul> <li>ATTRIBUTE_VAL_OUT, FLAG, IERR)</li> <li>INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR</li> <li>LOGICAL FLAG</li> <li>copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or</li> <li>Fortran; MPI_NULL_COPY_FN is a function that does nothing other than returning flag</li> <li>and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note</li> <li>Test </li> <li>Test </li> <li>Done</li> <li>in</li> <li>#-PR442</li> <li>the defined as follows: In the function index as primary</li> <li>twich is defined as follows: In the function index as primary</li> </ul>	1
<ul> <li>INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR</li> <li>LOGICAL FLAG</li> <li>copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or</li> <li>Fortran; MPI_NULL_COPY_FN is a function that does nothing other than returning flag</li> <li>o and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note</li> <li>that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated. Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn</li> <li>function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: In the function index as primary</li> </ul>	2
4       ATTRIBUTE_VAL_OUT, IERR       as cond location was line-broken.         5       LOGICAL FLAG       Bill: I will leave this as is.         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 flag       no         9       10       Fortran; MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag = 1,       returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note       Moderation that sets flag = 1,         10       Test        Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn       What must         13       function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call       which is defined as follows: In the function index as primary         14       15       MPI_Delete_function       function index as primary	3
<ul> <li>LOGICAL FLAG</li> <li>Bill: 1 will leave this as is.</li> <li>=&gt; don't worry</li> <li>copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or</li> <li>Fortran; MPI_NULL_COPY_FN is a function that does nothing other than returning flag</li> <li>0 and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag = 1,</li> <li>returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note</li> <li>that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated.</li> <li>Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn</li> <li>function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call</li> <li>is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function,</li> <li>which is defined as follows: In the function index as primary</li> <li>twpedef_int_MPI_Delete_function, MPI_Comm_commint_knyval</li> </ul>	4
<pre>6 7 #-error # 354 9 10 Test  Done in #-PR442 13 6 7 #-PR442 14 7 **-PR442 15 6 7 8 7 8 7 8 7 8 7 8 7 8 7 8 7 8 7 8 7</pre>	5
<ul> <li>#-error # 354</li> <li>Fortran; MPI_NULL_COPY_FN is a function that does nothing other than returning flag</li> <li>0 and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated. Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: In the function index as primary</li> </ul>	6
<ul> <li>and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated. Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: In the function index as primary two defint_MPI_Delete_function</li> </ul>	
<ul> <li><sup>9</sup></li> <li><sup>10</sup></li> <li>Test </li> <li>Done</li> <li><sup>13</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>15</sup></li> <li><sup>16</sup></li> <li><sup>17</sup></li> <li><sup>18</sup></li> <li><sup>19</sup></li> <li><sup>19</sup></li> <li><sup>19</sup></li> <li><sup>10</sup></li> <li><sup>11</sup></li> <li><sup>11</sup></li> <li><sup>11</sup></li> <li><sup>11</sup></li> <li><sup>12</sup></li> <li><sup>13</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>15</sup></li> <li><sup>14</sup></li> <li><sup>15</sup></li> <li><sup>14</sup></li> <li><sup>14</sup></li> <li><sup>15</sup></li> <li><sup>15</sup></li> <li><sup>16</sup></li> <li><sup>17</sup></li> <li><sup>18</sup></li> <li><sup>19</sup></li> <li><sup>19</sup></li> <li><sup>19</sup></li> <li><sup>19</sup></li> <li><sup>19</sup></li> <li><sup>19</sup></li> <li><sup>11</sup></li> <li><sup>11</sup></li> <li><sup>11</sup></li> <li><sup>11</sup></li> <li><sup>11</sup></li> <li><sup>11</sup></li> <li><sup>14</sup></li> <li><sup>15</sup></li> <li><sup>14</sup></li> <li><sup>15</sup></li> <li><sup>15</sup></li> <li><sup>16</sup></li> <li><sup>17</sup></li> <li><sup>18</sup></li> <li><sup>19</sup></li> <li><sup>11</sup></li> <li><sup>11</sup><th>#-error</th></li></ul>	#-error
<ul> <li>Test </li> <li>Test </li> <li>that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated. Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: In the function index as primary</li> <li>twpedef_int_MPI_Delete_function(MPI_Comm_commint_keywal)</li> </ul>	# <b>3</b> 54 •
Test Done in #-PR442 13 14 15 Test Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 13 14 15 Test 15 Test 15 Test 16 Test 16 Test 17 Test 16 Test 16 Test 16 Test 16 Test 16 Test 16 Test 16 Test 16 Test 16 Test 16 Test 16 Test 16 Test 17 Test 16 Test 17 Test	10
Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: In the function index as primary twpedef_int_MPI_Delete_function(MPI_Comm_commint_keywal)	
in #-PR442 is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows: In the function index as primary	Test
#-PR442 which is defined as follows: In the function index as primary	Done 13
typedef int MPT Delete function (MPT Comm comm int keyval	14
16 typedef int MPI_Delete_function(MPI_Comm comm, int keyval,	
	16
void *attribute_val, void *extra_state);	17
Done <sup>18</sup> A Fortran declaration for such a function is as follows:	Done 18
in 19 For this routing on interface within the main f08 module was never defined	13
#-PR442 In the function index as primary	#-PR442
21 SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)	21
<sup>22</sup> INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR	22
<sup>23</sup> delete_fn may be specified as MPI_NULL_DELETE_FN from either C or Fortran;	23
<sup>24</sup> MPI_NULL_DELETE_FN is a function that does nothing, other than returning	24
<sup>25</sup> MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated.	25
<sup>26</sup> The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL	26
<sup>27</sup> in MPI-2.0. The language independent definition of the deprecated function is the same as	27
<sup>28</sup> of the new function, except of the function name. The language bindings are modified.	28
29	29
30	30
31 MPI_KEYVAL_FREE(keyval)	31
<sup>32</sup> INOUT keyval Frees the integer key value (integer)	32
33	33
<sup>34</sup> C binding	34
int MPI_Keyval_free(int *keyval)	35
	36
<sup>37</sup> For this routine, an interface within the mpi_f08 module was never defined.	
Fortran binding	
MPT KEYVAL FREE (KEYVAL, TERROR)	
INTEGER KEYVAL. TERROR	
41	
<sup>42</sup> The following function is deprecated and is superseded by MPI_COMM_SET_ATTR in	
<sup>43</sup> MPI-2.0. The language independent definition of the deprecated function is the same as of	
<sup>44</sup> the new function, except of the function name. The language bindings are modified.	
45	
46	
47 48	
	6±°

MPI_ATT	R_PUT(comm, keyval, attribut	e_val)	1
INOUT	comm	communicator to which attribute will be attached (handle)	2 3
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE (integer)	4 5
IN	attribute_val	attribute value	6 7
C bindin	a		8 9
	0	nt keyval, void *attribute_val)	10 11
For this re	outine, an interface within the	mpi_f08 module was never defined.	12
Fortran l	6		13 14
	_PUT(COMM, KEYVAL, ATTRIB GER COMM, KEYVAL, ATTRIBU		14
			16
	· ·	d and is superseded by MPI_COMM_GET_ATTR in inition of the deprecated function is the same as of	17
	0 0 1	name. The language bindings are modified.	18 19
	) · · · · · · · · · · · · · · · · · · ·		20
	CET (communication attribute		21
	R_GET(comm, keyval, attribute		22
IN	comm	communicator to which attribute is attached (handle)	23
IN	keyval	key value (integer)	24 25
OUT	attribute_val	attribute value, unless $flag = false$	25 26
OUT	flag	true if an attribute value was extracted; false if no attribute is associated with the key	27 28
			29
C bindin	0		30
int MPI_A	Attr_get(MPI_Comm comm, i	nt keyval, void *attribute_val, int *flag)	31
For this re	outine, an interface within the	mpi_f08 module was never defined.	32 33
Fortran l	binding		34
	_GET(COMM, KEYVAL, ATTRIB	UTE_VAL, FLAG, IERROR)	35
	GER COMM, KEYVAL, ATTRIBU	TE_VAL, IERROR	36
LOGI	CAL FLAG		37
The fe	ollowing function is deprecated	and is superseded by MPI_COMM_DELETE_ATTR	38 39
in MPI-2.0	. The language independent of	definition of the deprecated function is the same as	40
of the new	function, except of the funct	ion name. The language bindings are modified.	41
			42
MPI_ATT	R_DELETE(comm, keyval)		43
INOUT	comm	communicator to which attribute is attached (handle)	44
IN	keyval	The key value of the deleted attribute (integer)	45 46
		The may reade of the deleted dutilitude (model)	47
C bindin	g		48

	( (	2			CHAPIER 10.	DEPREC	JAIED INIEI	<i>RFACES</i>
		nt MPI_A	ttr_delete(MP	I_Comm comm,	int keyval)			
	2 3 Fo	or this rou	ıtine, an interfa	ce within the m	pi_f08 module	was never	defined.	
6			inding DELETE(COMM, 1 ER COMM, KEYV		JR)			
٤	8							
#-error 1 #-366 1 Done 1 in 4-PR442 1	10 T 11 Th 12 Int 13 14 na	he entire he entire terfaces f The fo ames. Ot	set of $C++$ lan or more informa illowing functio her than the ty	age bindings wa aguage bindings ation. n typedefs hav pedef names, t	s deprecated as o s have been remove we been deprecat he function signa s of other functio	oved. See ted and a atures are	Chapter 17, F re superseded exactly the sa	Removed by new
#-366 #-error Done in #-PR442 2 #-error		6.3 De	MPI_Comm_er		MPI_Comm_errh MPI_File_errh MPI_Win_errha Select them as p index in the callb	aandler_f aandler_f andler_fu primary pack	function	
B2.2: 2. 2	е 24 Са	ancelling	•	4.0 y calling MPI_C	function prototyp CANCEL has been n.		ed and may be	removed
#-error <sup>2</sup> #-354 Done	e 16	6.4 De	precated sind	ce MPI-4.0				
# <sup>2</sup> B2.2: 17. 3	$^{29}$ The second sec	he followi .ll in MPI	0	eprecated and is	s superseded by th	he new MI	PI_INFO_GET_	STRING
		PI_INFO_	GET(info, key,	valuelen, value,	flag)			
	34	IN	info		info object (handle	e)		
#_DB377	35 36	IN	key		key (string)	associa	ated with \\mp	iarg{kev}
		IN	valuelen		length of value arg	<u> </u>		5( -))
	38 39	OUT	value		value (string)			
in <sup>4</sup>		OUT	flag		true if key defined	, false if no	ot (boolean)	
4	13 in 14	binding nt MPI_In			nst char *key,	int valu	lelen, char *	∗value,
4	16	I_Info_	008 binding get(info, key MPI_Info), IN		value, flag, ie info	error)		

CHARACTER(LEN=*), INTENT(IN) :: key 1 INTEGER, INTENT(IN) :: valuelen 2 CHARACTER(LEN=valuelen), INTENT(OUT) :: value 3 LOGICAL, INTENT(OUT) :: flag 4 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 5 6	# B2.2: 17. continued
Fortran binding       7         MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)       8         INTEGER INFO, VALUELEN, IERROR       9         CHARACTER*(*) KEY, VALUE       10         LOGICAL FLAG       Textblock page 474 lines 11-16 is missing       11         The following function is deprecated and is superseded by the new       12	#-364 #-error! Done in
MPI_INFO_GET_STRING call in MPI-4.0.	#-PR442
INinfoinfo object (handle)17INkeykey (string)associated with \\mpiarg{key}19OUTvaluelenlength of value arg (integer)20OUTflagtrue if key defined, false if not (boolean)21	#-PR377 #-error! #-update4 Done
C binding int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen, int *flag)	in #-PR442
26Fortran 2008 bindingMPI_Info_get_valuelen(info, key, valuelen, flag, ierror)TYPE(MPI_Info), INTENT(IN) :: infoCHARACTER(LEN=*), INTENT(IN) :: keyOINTEGER, INTENT(OUT) :: valuelenLOGICAL, INTENT(OUT) :: flagINTEGER, OPTIONAL, INTENT(OUT) :: ierror33	
Fortran binding       34         MPI_INF0_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)       35         INTEGER INFO, VALUELEN, IERROR       36         CHARACTER*(*) KEY       37         LOGICAL FLAG       Textblock page 474 lines 44-48 is missing	Done in #-PR442 #-364 #-error!
This term seems to be still in regular use on page 768 will be fixed The following Fortran subroutines are deprecated because the Fortran language MPI_SIZEOF and c_sizeof() return the size in bytes, storage_size() provides the size in bits. ***********************************	# B2.2: 9. #-error! Done in #-PR442 # B1.2: 2.





776	CHAPTER 17. REMOVED INTERFACES
$\pi^{-1}$	Removed MPI-1 datatypes. The indicated routine may be used for changing upper bound respectively.
<ul> <li>#-error</li> <li>Done</li> <li>in</li> <li>PR441</li> <li>Select it as primary</li> <li>in the</li> <li>Constants&amp;Handles</li> </ul>	MPLUB MPL TYPE CREATE RESIZED
9 10	Table 17.3: Removed MPI-1 constants
11	Removed MPI-1 Constants
12	C type: const int (or unnamed enum)
13	Fortran type: INTEGER
#-error Is Select it as primary	
Done in the	MPI_COMBINER_HVECTOR_INTEGER
in PR441	s index MPI_COMBINER_STRUCT_INTEGER
19	aved MPL 1 Callback Prototypes "and their replacements." or
<sup>20</sup> 17.1.5 Rem #-error! <sup>21</sup>	oved MPI-1 Callback Prototypes and their replacements. or "and their replacements since MPI-2."
#-367 <sup>21</sup> Table 17.4 sh	ows the removed MPI-1 callback prototypes and their MPI-2 replacements.
Done 23	
	le 17.4: Removed MPI-1 callback prototypes and their replacements
PR441 25 Select it as primary i	
#-error <sup>26</sup> / <sub>27</sub> in the callback functi	MPI_Handler_function MPI_Comm_errhandler_function
Done 28	
in 29 PR441 29	
30	
<sup>31</sup> 17.2 C+-	⊢ Bindings
32	
	ndings were deprecated as of MPI-2.2. The C++ bindings are removed in namespace is still reserved, however, and bindings may only be provided by
	ation as described in the MPI-2.2 standard.
36	
37	
38	
39	
40	
41 42	
43	
44	
45	
46	
47	
48	

## Chapter 18

# **Backward Incompatibilities**

Backward Incompatible since MPI-3.2

....bilities Starting in 4.0

## #-error! 18.1

MPI\_COMM\_DUP and MPI\_COMM\_IDUP no longer propagate info hints from the input communicator to the output communicator. This behavior can be achieved using MPI\_COMM\_DUP\_WITH\_INFO and MPI\_COMM\_IDUP\_WITH\_INFO.

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.

		2
Proposal:	The limit for length of MPI identifiers was removed. Prior to MPI-4.0, MPI identifiers were limited to 30 characters (31	-
#-error	with the profiling interface). This was done to avoid exceeding the limit on some compilation systems.	Done
#-335	Rationale. For Fortran, this limit was already relaxed for the Fortran specific function names, see Section 19.1.5,	in PR434
#-358 + 359 2.2. p10	and the Fortran language specification 2003 requires support for a minimum of 63 characters for internal and external identifiers.	<b></b>
18.1 p.777	Starting with the ISO/IEC 9899:1999 C programming language standard, support for a minimum of 63 characters	7
19.1.5 p789	is required for internal identifiers, but only 31 characters are required to be significant for external identifiers. At the time of the release of MPI-4.0, most or nearly all compilers allow external identifiers longer than 31 characters.	)
B.1.2 p.1031f	f Therefore, the restriction is removed. (End of rationale.)	)

The maximumm is 72 - 6 (space) - 12 ("SUBROUTINE P") = 54	Tables to verify the	30 = MPI_TYPE_CREATE_HINDEXED_BLOCK_C 31 = MPI T EVENT SET DROPPED HANDLER
	new statements on	31 = MPI_TYPE_CREATE_INDEXED_BLOCK_C
Note that in mpif.h, the versions with _c are not defined.	page 789.	31 = MPI_REDUCE_SCATTER_BLOCK_INIT_C
Interfaces that are longer than 54:	They will be not part	30 = MPI_TYPE_CREATE_HINDEXED_BLOCK 30 = MPI DIST GRAPH NEIGHBORS COUNT
<pre>56 = MPI_Neighbor_alltoallw_init_fts(a,b,c,d,e,f,g,h,i,j,k,l)</pre>	of the standard.	30 = MPI_DISI_GRAPH_NEIGHBORS_COUNT 30 = MPI DIST GRAPH_CREATE ADJACENT
56 = MPI_Neighbor_alltoallv_init_fts(a,b,c,d,e,f,g,h,i,j,k,l)		30 = MPI NEIGHBOR ALLGATHERV INIT C
55 = MPI_Neighbor_allgatherv_init_fts(a,b,c,d,e,f,g,h,i,j,k)		29 = MPI TYPE CREATE INDEXED BLOCK
For these 3 exceptions, _fts is shortened to _f		29 = MPI_SESSION_CREATE_ERRHANDLER
54 = MPI_Neighbor_alltoallw_init_f(a,b,c,d,e,f,g,h,i,j,j,k)		29 = MPI_REDUCE_SCATTER_BLOCK_INIT
<pre>54 = MPI_Neighbor_alltoallv_init_f(a,b,c,d,e,f,g,h,i,j,k,l) 53 = MPI Neighbor allgatherv init f(a,b,c,d,e,f,g,h,i,j,k)</pre>		40
Long routines with fts, but without exceptions	Same and shorter one	
52 = MPI Rget accumulate fts(a,b,c,d,e,f,g,h,i,j,k,l,m,n)		ltoall init(a,b,c,d,e,f,g,h,i,j)
52 = MPI Reduce scatter block init fts(a,b,c,d,e,f,g,h,i)		lltoallw(a,b,c,d,e,f,g,h,i,j,k)
52 = MPI Neighbor allgather init fts(a,b,c,d,e,f,g,h,i,j)		lltoallv(a,b,c,d,e,f,g,h,i,j,k)
Same, but without fts		darray(a,b,c,d,e,f,g,h,i,j,k)
53 = MPI_Dist_graph_create_adjacent(a,b,c,d,e,f,g,h,i,j,k)		llgatherv(a,b,c,d,e,f,g,h,i,j)
52 = MPI_Neighbor_alltoallw_init(a,b,c,d,e,f,g,h,i,j,k,l)	45 = MPI_Get_accumul	ate(a,b,c,d,e,f,g,h,i,j,k,l,m)
52 = MPI_Neighbor_alltoallv_init(a,b,c,d,e,f,g,h,i,j,k,l)	44 = MPI_Comm_spawn_	multiple(a,b,c,d,e,f,g,h,i,j)
51 = MPI_Neighbor_allgatherv_init(a,b,c,d,e,f,g,h,i,j,k)	43 = MPI_Type_create	_hindexed_block(a,b,c,d,e,f)
<pre>51 = MPI_Intercomm_create_from_groups(a,b,c,d,e,f,g,h,i)</pre>		ltoallw(a,b,c,d,e,f,g,h,i,j)
<pre>48 = MPI_Rget_accumulate(a,b,c,d,e,f,g,h,i,j,k,l,m,n)</pre>		ltoallv(a,b,c,d,e,f,g,h,i,j)
48 = MPI_Reduce_scatter_block_init(a,b,c,d,e,f,g,h,i)		nit(a,b,c,d,e,f,g,h,i,j,k,l)
<pre>48 = MPI_Neighbor_allgather_init(a,b,c,d,e,f,g,h,i,j)</pre>	43 = MPI_Alltoallv_i	nit(a,b,c,d,e,f,g,h,i,j,k,l)

Unofficial Draft for Comment Only

32 = MPI INTERCOMM CREATE FROM GROUPS

1 2 3

6

10 11 12

 $13 \\ 14$ 

15 16

17

18

19

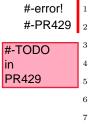
20

21

Done

## 

In the definition of \mpifunc{MPI\\_T\\_CATEGORY\\_CHANGED}, the name of the \OUT/



## Chapter 19

# Language Bindings

## 19.1 Support for Fortran

## 19.1.1 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 [45] + TS 29113 [46].

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 19.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

- 1. USE mpi\_f08: This method is described in Section 19.1.2. 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 19.1.3 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 19.1.4. The use of the include file mpif.h is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

1 Compliant MPI-3 implementations providing a Fortran interface must provide one or  $\mathbf{2}$ both of the following: 3 • The USE mpi\_f08 Fortran support method. 4 5• The USE mpi and INCLUDE 'mpif.h' Fortran support methods. 6  $\overline{7}$ Section 19.1.6 describes restrictions if the compiler does not support all the needed features. 8 Application subroutines and functions may use either one of the modules or the mpif.h 9 include file. An implementation may require the use of one of the modules to prevent type 10mismatch errors. 11Advice to users. Users are advised to utilize one of the MPI modules even if mpif.h 12enforces type checking on a particular system. Using a module provides several poten-13 tial advantages over using an include file; the mpi\_f08 module offers the most robust 14and complete Fortran support. (End of advice to users.) 1516In a single application, it must be possible to link together routines which USE mpi\_f08. 17 USE mpi, and INCLUDE 'mpif.h'. 18 The LOGICAL compile-time constant MPI\_SUBARRAYS\_SUPPORTED is set to .TRUE. if 19all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-20rank [46]; otherwise it is set to .FALSE.. The LOGICAL compile-time constant 21MPI\_ASYNC\_PROTECTS\_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was 22added 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. 28Section 19.1.2 through 19.1.4 define the Fortran support methods. The Fortran in-29terfaces of each MPI routine are shorthands. Section 19.1.5 defines the corresponding 30 full interface specification together with the specific procedure names and implications for  $^{31}$ the profiling interface. Section 19.1.6 the implementation of the MPI routines for differ-32 ent versions of the Fortran standard. Section 19.1.7 summarizes major requirements for 33 valid MPI-3.0 implementations with Fortran support. Section 19.1.8 and Section 19.1.9 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 functions 36 provides additional support for Fortran intrinsic numeric types, including parameterized 37 types: 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 19.1.10 through 19.1.19 give an overview and details on known prob-41 lems when using Fortran together with MPI; Section 19.1.20 compares the Fortran problems 42with those in C. 43

44

## 19.1.2 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 19.1.6 describes restrictions if the compiler
 does not support all the needed features. Within all MPI function specifications, the first

## Unofficial Draft for Comment Only

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 19.1.3.

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 19.1.6 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 noncontiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the MPI\_SUBARRAYS\_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TS 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 19.1.6 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 [45], 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 19.1.3. (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).

(e.g., of type MPI\_Comm\_copy\_attr\_function or COMM\_COPY\_ATTR\_FUNCTION) 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 [45] together with the Technical Specification "TS 29113 Further Interoperability with C" [46] 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 [46], "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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19 20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34 35

36

37

38

39

40

41

42

43

44

45

46

47

48

#-error! #-other Done in PR449 #-TODO Resulting line is 8pt to long. Should a flushline be added? 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.*)

## 19.1.3 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 19.1.6 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

 $\mathbf{2}$ 

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 19.1.12 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 [32], if the compiler supports this TS 29113 language feature. See Section 19.1.6 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 19.1.6 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 INTENT(OUT) is used. In particular, output array arguments are expected to keep their content as long as the MPI routine does not modify them. To keep this behavior, it is recommended that implementations not use INTENT(OUT) in the mpi module and the mpif.h include file, even though INTENT(OUT) is specified in an interface description of the mpi\_f08 module. (End of advice to implementors.)

## 19.1.4 Fortran Support Through the mpif.h Include File

The use of the mpif.h include file is strongly discouraged and may be deprecated in a future version of MPI.

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

 $\overline{7}$ 

 $46 \\ 47$ 

	786 CHAPTER 19. LANGUAGE BINDINGS
1 2	<ul> <li>Declaration of a status as an INTEGER variable instead of an INTEGER array with size MPI_STATUS_SIZE.</li> </ul>
$\frac{3}{4}$	- Incorrect argument positions; e.g., interchanging the count and
5	datatype arguments. — Passing incorrect MPI handles; e.g., passing a datatype instead of a commu-
6 7	nicator.
8	• The migration from mpif.h to the mpi module should be relatively straightfor-
9 10	ward (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.
11	• Migrating portable and correctly written applications to the mpi module is not
12 13	expected to be difficult. No compile or runtime problems should occur because an mpif.h include file was always allowed to provide explicit Fortran interfaces.
14 15	(End of advice to users.)
16 17	Rationale. With MPI-3.0, the mpif.h include file was not deprecated in order to
18	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
19 20	can be used. ( <i>End of rationale.</i> )
21	
22	19.1.5 Interface Specifications, Procedure Names, and the Profiling Interface
23 24	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
25	together with additional attributes. The Fortran standard allows a given Fortran interface
26 27	to be implemented with several methods, e.g., within or outside of a module, with or without
28	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
29	several implementation schemes together with the rules for the specific procedure names
30 31	and its implications for the profiling interface are specified within this section, but not the
32	implementation details.
33 34	<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:
35 36	• Portable implementation of the wrappers from the MPI Fortran interfaces to the
37 38	MPI routines in C.
39	• Binary backward compatible implementation path when switching MPI_SUBARRAYS_SUPPORTED from .FALSE. to .TRUE
40 41	• The Fortran PMPI interface need not be backward compatible, but a method
42 43	must be included that a tools layer can use to examine the MPI library about the specific procedure names and interfaces used.
43 44	• No performance drawbacks.
45	• Consistency between all three Fortran support methods.
46 47	• Consistent with Fortran $2008 + TS 29113$ .
48	

No.	Specific pro- cedure name	Calling convention	1
1A	MPI_Isend_f08	Fortran interface and arguments, as in Annex A.4, except	3
		that in routines with a choice buffer dummy argument, this	4
		dummy argument is implemented with non-standard ex-	5
		tensions like !\$PRAGMA IGNORE_TKR, which provides a call-	6
		by-reference argument without type, kind, and dimension	7
		checking.	8
1B	MPI_Isend_f08ts	Fortran interface and arguments, as in Annex A.4, but	9
		only for routines with one or more choice buffer dummy	10
		arguments; these dummy arguments are implemented with	1
		TYPE(*), DIMENSION().	15
2A	MPI_ISEND	Fortran interface and arguments, as in Annex A.5, except	13
		that in routines with a choice buffer dummy argument, this	14
		dummy argument is implemented with non-standard ex-	1
		tensions like <b>!</b> \$PRAGMA IGNORE_TKR, which provides a call-	10
		by-reference argument without type, kind, and dimension	17
		checking.	18
2B	MPI_ISEND_FTS	Fortran interface and arguments, as in Annex A.5, but	19
In mpif.h only, the p	_	only for routines with one or more choice buffer dummy	20
MPI_NEIGHBOR_ALL		arguments; these dummy arguments are implemented with	2
MPI_NEIGHBOR_ALLTOALLV_INIT, and		TYPE(*), DIMENSION().	22
MPI_NEIGHBOR_ALL shortened to "_F".	.GATHERV_INIT IS		23
shortened to _F.			24

Table 19.1: 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 19.1. Case is not significant in the names.

Note that for the deprecated routines in Section 16.1, 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, <sup>46</sup> it is required that all nonblocking and split-collective routines with buffer arguments are <sup>47</sup> implemented according to 1B and 2B, i.e., with MPI\_Xxxx\_f08ts in the mpi\_f08 module, <sup>48</sup>

## Unofficial Draft for Comment Only

Done

**PR434** 

-error

update2

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 19.1. However, the MPI library may contain multiple implementation schemes from Table 19.1.

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.*)

9 Rationale. After a compiler provides the facilities from TS 29113, i.e., TYPE(\*), 10 DIMENSION(...), it is possible to change the bindings within a Fortran support method 11 to support subarrays without recompiling the complete application provided that the 12previous interfaces with their specific procedure names are still included in the li-13 brary. Of course, only recompiled routines can benefit from the added facilities. 14There is no binary compatibility conflict because each interface uses its own spe-15cific procedure names and all interfaces use the same constants (except the value of 16MPI\_SUBARRAYS\_SUPPORTED and MPI\_ASYNC\_PROTECTS\_NONBLOCKING) and type 17 definitions. After a compiler also ensures that buffer arguments of nonblocking MPI 18 operations can be protected through the ASYNCHRONOUS attribute, and the proce-19 dure declarations in the mpi\_f08 and mpi module and the mpif.h include file declare 20choice buffers with the ASYNCHRONOUS attribute, then the value of

<sup>21</sup> MPI\_ASYNC\_PROTECTS\_NONBLOCKING can be switched to .TRUE. in the module def-<sup>22</sup> inition 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.)

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.

A user-written or middleware profiling routine should provide the same specific Fortran procedure names and calling conventions, and therefore can interpose itself as the MPI library routine. The profiling routine can internally call the matching

PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments, choice buffer arguments, or that are attribute caching routines (

<sup>38</sup> MPI\_{COMM|WIN|TYPE}\_{SET|GET}\_ATTR). In this case, the profiling software should <sup>40</sup> invoke the corresponding PMPI routine using the same Fortran support method as used in <sup>41</sup> the calling application program, because the C, mpi\_f08 and mpi callback prototypes are <sup>42</sup> different or the meaning of the choice buffer or attribute\_val arguments are different.

Advice to users. Although for each support method and MPI routine (e.g.,

<sup>44</sup> MPI\_ISEND in mpi\_f08), multiple routines may need to be provided to intercept <sup>45</sup> the specific procedures in the MPI library (e.g., MPI\_Isend\_f08 and MPI\_Isend\_f08ts), <sup>46</sup> each profiling routine itself uses only one support method (e.g., mpi\_f08) and calls <sup>47</sup> the real MPI routine through the one PMPI routine defined in this support method <sup>48</sup> (i.e., PMPI\_Isend in this example). (*End of advice to users.*)

## Unofficial Draft for Comment Only

1

5

6

7

8

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

		$^{27}$ #-orror
These examples are	SUBROUTINE PMPI_DIST_GRAPH_CREATE_ADJACENT(a,b,c,d,e,f,g,h,i,j,k)	<sup>27</sup> #-error <sup>28</sup> #-update2
beased on the tables	SUBROUTINE PMPI_NEIGHBOR_ALLTOALLW_INIT(a,b,c,d,e,f,g,h,i,j,k,l) SUBROUTINE PMPI NEIGHBOR ALLTOALLV INIT(a,b,c,d,e,f,g,h,i,j,k,l)	29
on page 777	SUBROUTINE PMPI_NEIGHBOR_ALLTUALLV_INIT(a,b,c,d,e,i,g,n,i,j,k,i)	
on page 777	71 and 70 each Some of the	30
	with 71 and 66 characters. With buffers implemented with TS 29113, the specific	31
	procedure names have an additional postfix. The longest of such interface definitions	32
are		33
arc	SUBROUTINE PMPI NEIGHBOR ALLTOALLW INIT F(a,b,c,d,e,f,g,h,i,j,j,k)	34
	INTERFACE PMPI_NEIGHBOR_ALLGATHERV_INIT	35
	SUBROUTINE PMPI_NEIGHBOR_ALLGATHERV_INIT_F(a,b,c,d,e,f,g,h,i,j,k)	36
	INTERFACE PMPI_RGET_ACCUMULATE	37
72, 71, a		38
	with 70 characters. In principle, continuation lines would be possible in mpif.h (spaces	39
	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	40
	characters. Column 133 is also not available for the continuation character because	41
	lines longer than 132 characters are invalid with some compilers by default.	42
<i>"</i> , , , ,		43
#-error!	The longest specific procedure names are PMPI_Dist_graph_create_adjacent_f08 and	44
#-335	PMPI_File_write_ordered_begin_f08ts both with 35 characters in the mpi_f08 module.	45
#-358 + 359	For example, the interface specifications together with the specific procedure names	46
2.2. p10	can be implemented with The longest specific procedure name is	47
18.1 p.777	The longest specific procedure name is	48
19.1.5 p789	PMPI_Reduce_scatter_block_init_c_f08ts	
B.1.2 p.1031ff	with 38 characters in the mpi_f08 module.	· ]
1	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

Done

PR434

26 In

```
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)
                                                  ! The INTENT may be added although
17
                 INTEGER, INTENT(IN) :: comm
                 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
           !23456789012345678901234567890123456789012345678901234567890123456789012
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
           Advice to users.
                             The following is an example of how a user-written or middleware
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
```

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 19.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

# 19.1.6 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	• For Fortran 50. The major additional features that are needed from Fortran 90 are:
3	
4	- The MODULE and INTERFACE concept.
5	- The KIND= and SELECTED_XXX_KIND concept.
6 7	- Fortran derived TYPEs and the SEQUENCE attribute.
8	- The OPTIONAL attribute for dummy arguments.
9	- Cray pointers, which are a non-standard compiler extension, are needed for the
10	use of MPI_ALLOC_MEM.
11	With these features MPL11 MPL22 can be implemented without restrictions
12	With these features, MPI-1.1 – MPI-2.2 can be implemented without restrictions. MPI-3.0 can be implemented with some restrictions. The Fortran support methods
13	are abbreviated with $S1 = \text{the mpi_f08}$ module, $S2 = \text{the mpi}$ module, and $S3 = \text{the}$
14 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	
18	- MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non-
19	contiguous subarrays cannot be used as buffers in nonblocking routines, RMA, or split-collective I/O.
20	
21	<ul> <li>S1, S2, and S3 can be implemented, but for S1, only a preliminary implementa- tion is possible.</li> </ul>
22 23	
23 24	- In this preliminary interface of S1, the following changes are necessary:
25	* TYPE(*), DIMENSION() is substituted by non-standardized extensions
26	like !\$PRAGMA IGNORE_TKR.
27	* The ASYNCHRONOUS attribute is omitted.
28	* PROCEDURE() callback declarations are substituted by EXTERNAL.
29	- The specific procedure names are specified in Section 19.1.5.
30 31	- Due to the rules specified in Section 19.1.5, choice buffer declarations should be
32	implemented only with non-standardized extensions like <b>!</b> \$PRAGMA IGNORE_TKR
33	(as long as $F2008+TS$ 29113 is not available).
34	In S2 and S3: Without such extensions, routines with choice buffers should be provided with an implicit interface, instead of overloading with a different MPI
35	function for each possible buffer type (as mentioned in Section 19.1.11). Such
36	overloading would also imply restrictions for passing Fortran derived types as
37	choice buffer, see also Section 19.1.15.
38	Only in S1: The implicit interfaces for routines with choice buffer arguments
39 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	
40	

- In S1 and S2, the definition of the handle types (e.g., TYPE(MPI\_Comm) and  $\mathbf{2}$ the status type TYPE(MPI\_Status) must be modified: The SEQUENCE attribute 3 must be used instead of BIND(C) (which is not available in Fortran 90/95). This 4 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 5handles may have changed. For this reason, an implementor may choose not to 6  $\overline{7}$ 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 19.3.5) are also not available in the mpi module and mpif.h. 9
- For Fortran 95:

The quality of the MPI interface and the restrictions are the same as with Fortran 90.

• For Fortran 2003:

The major features that are needed from Fortran 2003 are:

- Interoperability with C, i.e.,
  - \* BIND(C) derived types.
  - \* The ISO\_C\_BINDING intrinsic type C\_PTR and routine C\_F\_POINTER.
- The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy arguments.
- The ability to overload the operators .EQ. and .NE. to allow the comparison of derived types (used in MPI-3.0 for MPI handles).
- The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O. This feature is not vet used by MPI, but it is the basis for the enhancement for MPI communication in the TS 29113.

With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2 can be implemented without restrictions, but with one enhancement:

- 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.

MPI-3.0 can be implemented with the following restrictions:

- MPI\_SUBARRAYS\_SUPPORTED equals .FALSE..
- For S1, only a preliminary implementation is possible. The following changes are necessary:
  - \* TYPE(\*), DIMENSION(..) is substituted by non-standardized extensions like !\$PRAGMA IGNORE\_TKR.
- The specific procedure names are specified in Section 19.1.5.
- With S1, the ASYNCHRONOUS is required as specified in the second Fortran interfaces. With S2 and S3 the implementation can also add this attribute if explicit interfaces are used.

1

8

10

11

1213

14

1516

17

18

19

20

2122

23

 $^{24}$ 

25

26

2728

29

30

31

32

33

34 35

36 37

38

39

40

41

42

43 44

45

46

1 2 3 4	<ul> <li>The ASYNCHRONOUS Fortran attribute can be used in applications to <i>try to</i> protect buffers in nonblocking MPI calls, but the protection can work only if the compiler is able to protect asynchronous Fortran I/O and makes no difference between such asynchronous Fortran I/O and MPI communication.</li> </ul>
5 6 7	<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>
8 9 10	<ul> <li>be used only for Fortran types that are C compatible.</li> <li>The same restriction as for Fortran 90 applies if non-standardized extensions like !\$PRAGMA IGNORE_TKR are not available.</li> </ul>
11 12	• For Fortran 2008 + TS 29113 and later and For Fortran 2003 + TS 29113: The maximum factome that are useded from TS 20112 and
13 14	The major feature that are needed from TS 29113 are:
15 16	<ul> <li>TYPE(*), DIMENSION() is available.</li> <li>The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI communication.</li> </ul>
17 18 19	<ul> <li>The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not restricted to Fortran types for which a corresponding type in C exists.</li> </ul>
20	Using these features, MPI-3.0 can be implemented without any restrictions.
21 22	- With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE The
23	ASYNCHRONOUS attribute can be used to protect buffers in nonblocking MPI calls.
24	The TYPE(C_PTR) 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	- With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation
28	dependent. A high quality implementation will also provide
29 30	MPI_SUBARRAYS_SUPPORTED set to .TRUE. and will use the ASYNCHRONOUS attribute in the same way as in S1.
31 32	$-$ If non-standardized extensions like <b>!\$PRAGMA IGNORE_TKR</b> are not available then
33	S2 must be implemented with TYPE(*), DIMENSION().
34	Advice to implementors. If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice
35	argument may be implemented with an explicit interface using compiler directives,
36	for example:
37	INTERFACE
38 39	SUBROUTINE MPI(buf,)
40	!DEC\$ ATTRIBUTES NO_ARG_CHECK :: buf
41	!\$PRAGMA IGNORE_TKR buf
42	!DIR\$ IGNORE_TKR buf
43	!IBM* IGNORE_TKR buf REAL, DIMENSION(*) :: buf
44	! declarations of the other arguments
45	END SUBROUTINE
46	END INTERFACE
47	
48	(End of advice to implementors.)

# 19.1.7 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 19.1.11 through 19.1.19 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 [46] 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 19.1.12 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 set to .FALSE.. Observation of these rules by the MPI application developer is especially recommended for backward compatibility of existing applications that use the mpi module or the mpif.h include file. The rules are as follows:

 $\overline{7}$ 

 $^{24}$ 

- 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 818 and Section 19.1.8, and DD on page 819) solve the problems described in Section 19.1.17.
  - The problems with temporary data movement (described in detail in Section 19.1.18) 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.
  - Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 19.1.19) are resolved **without** any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in invoking MPI procedures.

All of these rules are valid for the mpi\_f08 and mpi modules and independently of whether mpif.h uses explicit interfaces.

Advice to implementors. Some of these rules are already part of the Fortran 2003 standard, some of these requirements require the Fortran TS 29113 [46], and some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (End of advice to implementors.)

 $^{24}$ 

# 19.1.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 19.1.17, 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.

- <sup>29</sup> MPI\_F\_SYNC\_REG(buf)
  - INOUT

initial address of buffer (choice)

Fortran 2008 binding

buf

MPI\_F\_sync\_reg(buf)

TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: buf

<sup>36</sup> Fortran binding
 <sup>37</sup> MPI\_F\_SYNC\_REG(BUF)
 <sup>38</sup> <type> BUF(\*)

This routine has no executable statements. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

*Rationale.* This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. (*End of rationale.*)

 $\mathbf{2}$ 

Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is explicitly allowed to implement this routine within the mpi\_f08 module according to the definition for the mpi module or mpif.h to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (End of advice to implementors.)

Rationale. This routine is not defined with TYPE(\*), DIMENSION(\*), i.e., assumed size instead of assumed rank, because this would restrict the usability to "simply contiguous" arrays and would require overloading with another interface for scalar arguments. (End of rationale.)

If only a part of an array (e.g., defined by a subscript triplet) is Advice to users. 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.)

#### Additional Support for Fortran Numeric Intrinsic Types 19.1.9

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 27CHARACTER) with an optional integer KIND parameter that selects from among one or more 2829variants. The specific meaning of different KIND values themselves are implementation 30 dependent and not specified by the language. Fortran provides the KIND selection functions 31selected\_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. 33 These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX, and 34 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 3536 PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two 37 declarations are equivalent:

double precision x real(KIND(0.0d0)) x

41 MPI provides two orthogonal methods for handling communication buffers of numeric 42intrinsic 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 4344with 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 4546conversion in heterogeneous environments. The second method (see "Support for size-47specific MPI Datatypes" on page 802) gives the user complete control over communication 48 by exposing machine representations.

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

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

7

8

9

10

11

1213

14

15

16

17

18

1920

21

22

23 $^{24}$ 

25

26

32

38

39

1 2

## Parameterized Datatypes with Specified Precision and Exponent Range

 $_{\scriptscriptstyle 3}$   $\,$   $\,$  MPI provides named data types corresponding to standard Fortran 77 numeric types:

<sup>4</sup> MPI\_INTEGER, MPI\_COMPLEX, MPI\_REAL, MPI\_DOUBLE\_PRECISION and

<sup>5</sup> MPI\_DOUBLE\_COMPLEX. MPI automatically selects the correct data size and provides rep-<sup>6</sup> resentation conversion in heterogeneous environments. The mechanism described in this <sup>7</sup> 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 p is decimal digits of precision and 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.)

32 33 34

45

19

20

21

22 23

24

25

26

27

28

29

30

31

MPI\_TYPE\_CREATE\_F90\_REAL(p, r, newtype)

ĬN	р	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested $MPI$ data type (handle)

<sup>40</sup> C binding

```
<sup>41</sup> int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
```

```
42
43 Fortran 2008 binding
```

```
MPI_Type_create_f90_real(p, r, newtype, ierror)
```

```
INTEGER, INTENT(IN) :: p, r
```

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

```
47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

<sup>48</sup> Fortran binding

#### MPI\_TYPE\_CREATE\_F90\_REAL(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR

This function returns a predefined MPI datatype that matches a REAL variable of KIND selected\_real\_kind(p, r). In the model described above it returns a handle for the element D(p, r). Either p or r may be omitted from calls to selected\_real\_kind(p, r) (but not both). Analogously, either p or r may be set to MPI\_UNDEFINED. In communication, an MPI datatype A returned by MPI\_TYPE\_CREATE\_F90\_REAL matches a datatype B if and only if B was returned by MPI\_TYPE\_CREATE\_F90\_REAL called with the same values for  $\mathbf{p}$  and  $\mathbf{r}$  or  $\mathbf{B}$  is a duplicate of such a datatype. Restrictions on using the returned datatype with the "external32" data representation are given on page 801.

It is erroneous to supply values for p and r not supported by the compiler.

MPI.	_TYPE_	_CREATE_	_F90_	_COMPLEX	Х(р,	r,	newtype)	ļ
------	--------	----------	-------	----------	------	----	----------	---

IN	р	precision, in decimal digits (integer)	
IN	r	decimal exponent range (integer)	
OUT	newtype	the requested MPI datatype (handle)	

#### C binding

int MPI\_Type\_create\_f90\_complex(int p, int r, MPI\_Datatype \*newtype)

Fortran 2008 binding
<pre>MPI_Type_create_f90_complex(p, r, newtype, ierror)</pre>
INTEGER, INTENT(IN) :: p, r
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding

# Fortran binding

MPI.	_TYPE_	CRE	EATE	_F9	90_	COMPL	.EX	(P,	R,	NEWTYF	ΡE,	IERROR)	
	INTEG	ER	Ρ.	R.	NE	WTYPE		IERF	ROR				

This function returns a predefined MPI datatype that matches a COMPLEX variable of KIND selected\_real\_kind(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. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI\_TYPE\_CREATE\_F90\_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 801.

It is erroneous to supply values for p and r not supported by the compiler.

	*						
MPI_TYPE	_CREATE_F90_INTEGER(r, n	ewtype)	41				
IN	r	decimal exponent range, i.e., number of decimal	42				
		digits (integer)	43				
OUT	newtype	the requested MPI datatype (handle)	44				
001	newtype	the requested with datatype (handle)	45				
<b>a</b> 1 1 11			46				
C binding			47				

int MPI\_Type\_create\_f90\_integer(int r, MPI\_Datatype \*newtype)

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

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

21

33

34

35

36

37 38

39

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Type_create_f90_integer(r, newtype, ierror)
3
          INTEGER, INTENT(IN) :: r
4
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
5
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     Fortran binding
7
     MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
8
          INTEGER R, NEWTYPE, IERROR
9
10
          This function returns a predefined MPI datatype that matches a INTEGER variable of
11
     KIND selected_int_kind(r). Matching rules for datatypes created by this function are
12
     analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL.
13
     Restrictions on using the returned datatype with the "external32" data representation are
14
     given on page 801.
15
         It is erroneous to supply a value for r that is not supported by the compiler.
16
         Example:
17
                     longtype, quadtype
     integer
18
     integer, parameter :: long = selected_int_kind(15)
19
     integer(long) ii(10)
20
     real(selected_real_kind(30)) x(10)
21
     call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
22
     call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
23
     . . .
24
25
     call MPI_SEND(ii, 10, longtype, ...)
26
     call MPI_SEND(x,
                         10, quadtype, ...)
27
28
           Advice to users.
                              The datatypes returned by the above functions are predefined
29
           datatypes. They cannot be freed; they do not need to be committed; they can be
30
           used with predefined reduction operations. There are two situations in which they
31
           behave differently syntactically, but not semantically, from the MPI named predefined
32
           datatypes.
33
34
            1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to
35
               retrieve the values of p and r.
36
            2. Because the datatypes are not named, they cannot be used as compile-time
37
               initializers or otherwise accessed before a call to one of the
38
               MPI_TYPE_CREATE_F90_XXX routines.
39
           If a variable was declared specifying a non-default KIND value that was not obtained
40
41
           with selected_real_kind() or selected_int_kind(), the only way to obtain a
42
           matching MPI datatype is to use the size-based mechanism described in the next
           section.
43
44
           (End of advice to users.)
45
46
           Advice to implementors.
                                     An application may often repeat a call to
47
           MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The appli-
48
```

cation is not allowed to free the returned predefined, unnamed datatype handles. To

prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/ INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI\_TYPE\_CREATE\_F90\_XXX and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (*End of advice to implementors.*)

*Rationale.* The MPI\_TYPE\_CREATE\_F90\_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 14.5.2) or user-defined (Section 14.5.3) 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 14.5.2.

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:

					26
if	(p > 33) q	or (r > 4931)	then exte	rnal32 representa	ation 27
			is u	ndefined	
else if	(n > 15)	or (r > 307)	then exte	ernal32_size = 16	28
	-			—	29
	(p > 6) c	r(r > 37)		rnal32_size = 8	30
else			exte	$rnal32_size = 4$	31
			-V. +	-: f	32
		E_F90_COMPLE	:A: twice the	size as for	-
MPI_TYPE_C	CREATE_F9	0_REAL.			33
For MPI_TYF	PE_CREATE	E_F90_INTEGEI	र:		34
	-				35
if	(r > 38) t	then externa	132 represe	ntation is undefi	ined 36
else if	(r > 18) t	then externa	132_size =	16	37
else if	(r > 9) t	then externa	132_size =	8	38
else if	(r > 4) t	then externa	132_size =	4	39
else if	(r > 2) t	then externa	132_size =	2	40
else		externa	132_size =	1	41

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.

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

Support for Size-specific MPI Datatypes 2 MPI provides named datatypes corresponding to optional Fortran 77 numeric types that 3 contain explicit byte lengths—MPI\_REAL4, MPI\_INTEGER8, etc. This section describes a 4 mechanism that generalizes this model to support all Fortran numeric intrinsic types. 5We assume that for each **typeclass** (integer, real, complex) and each word size there is 6 a unique machine representation. For every pair (typeclass, n) supported by a compiler, 7 MPI must provide a named size-specific datatype. The name of this datatype is of the form 8 MPI\_<TYPE>n in C and Fortran where <TYPE> is one of REAL, INTEGER and COMPLEX, and 9 **n** is the length in bytes of the machine representation. This datatype locally matches all 10 variables of type (typeclass, n) in Fortran. The list of names for such types includes: 11 MPI\_REAL4 12MPI\_REAL8 13 MPI\_REAL16 14MPI\_COMPLEX8 1516MPI\_COMPLEX16 17MPI\_COMPLEX32 MPI\_INTEGER1 18 19MPI\_INTEGER2 MPI\_INTEGER4 2021MPI\_INTEGER8 MPI\_INTEGER16 22 23One datatype is required for each representation supported by the Fortran compiler. 24Particularly for the longer floating-point types, C and Fortran may use Rationale. 25different representations. For example, a Fortran compiler may define a 16-byte REAL 26type with 33 decimal digits of precision while a C compiler may define a 16-byte long 27double type that implements an 80-bit (10 byte) extended precision floating point 28 value. Both of these types are 16 bytes long, but they are not interoperable. Thus, 29 these types are defined by Fortran, even though C may define types of the same length. 30 (End of rationale.)  $^{31}$ 32 To be backward compatible with the interpretation of these types in MPI-1, we assume 33 that the nonstandard declarations REAL\*n, INTEGER\*n, always create a variable whose rep-34resentation is of size **n**. These datatypes may also be used for variables declared with 35 KIND=INT8/16/32/64 or KIND=REAL32/64/128, which are defined in the ISO\_FORTRAN\_ENV 36 intrinsic module. Note that the MPI datatypes and the REAL\*n, INTEGER\*n declarations 37 count bytes whereas the Fortran KIND values count bits. All these datatypes are predefined. 38 The following function allows a user to obtain a size-specific MPI datatype for any 39 intrinsic Fortran type. 40 41 42MPI\_TYPE\_MATCH\_SIZE(typeclass, size, datatype) 43 IN typeclass generic type specifier (integer) 44 IN size size, in bytes, of representation (integer) 45OUT datatype datatype with correct type, size (handle) 464748 C binding

```
1
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)
                                                                                          2
Fortran 2008 binding
MPI_Type_match_size(typeclass, size, datatype, ierror)
    INTEGER, INTENT(IN) :: typeclass, size
                                                                                          5
    TYPE(MPI_Datatype), INTENT(OUT) :: datatype
                                                                                          6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
                                                                                          a
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
                                                                                         10
    INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
                                                                                         11
    typeclass is one of MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER and
                                                                                         12
MPI_TYPECLASS_COMPLEX, corresponding to the desired typeclass. The function returns
                                                                                         13
an MPI datatype matching a local variable of type (typeclass, size).
                                                                                         14
    This function returns a reference (handle) to one of the predefined named datatypes,
                                                                                         15
not a duplicate. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to
                                                                                         16
obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling
                                                                                         17
storage_size() in order to compute the variable size in bits, dividing it by eight, and then
                                                                                         18
calling MPI_TYPE_MATCH_SIZE to find a suitable datatype. In C, one can use the C
                                                                                         19
function sizeof() (which returns the size in bytes) instead of storage_size() (which returns
                                                                                         20
the size in bits). In addition, for variables of default kind the variable's size can be computed
                                                                                         21
by a call to MPI_TYPE_GET_EXTENT, if the typeclass is known. It is erroneous to specify
                                                                                         22
a size not supported by the compiler.
                                                                                         23
                                                                                         ^{24}
     Rationale. This is a convenience function. Without it, it can be tedious to find the
                                                                                         25
     correct named type. See note to implementors below. (End of rationale.)
                                                                                         26
                                                                                         27
     Advice to implementors. This function could be implemented as a series of tests.
                                                                                         28
                                                                                         29
     int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
                                                                                         30
     ſ
                                                                                         31
       switch(typeclass) {
                                                                                         32
            case MPI_TYPECLASS_REAL: switch(size) {
                                                                                         33
              case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
                                                                                         34
              case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
                                                                                         35
              default: error(...);
                                                                                         36
            }
                                                                                         37
            case MPI_TYPECLASS_INTEGER: switch(size) {
                                                                                         38
               case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
                                                                                         39
               case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
                                                                                         40
               default: error(...);
                                                                                         41
            }
                                                                                         42
           ... etc. ...
                                                                                         43
        }
                                                                                         44
                                                                                         45
        return MPI_SUCCESS;
                                                                                         46
     }
                                                                                         47
                                                                                         48
```

```
(End of advice to implementors.)
```

## Communication With Size-specific Types

The usual type matching rules apply to size-specific datatypes: a value sent with datatype MPI\_<TYPE>n can be received with this same datatype on another process. Most modern computers use 2's complement for integers and IEEE format for floating point. Thus, communication using these size-specific datatypes will not entail loss of precision or truncation errors.

> Advice to users. Care is required when communicating in a heterogeneous environment. Consider the following code:

This may not work in a heterogeneous environment if the value of size is not the same on process 1 and process 0. There should be no problem in a homogeneous environment. To communicate in a heterogeneous environment, there are at least four options. The first is to declare variables of default type and use the MPI datatypes for these types, e.g., declare a variable of type REAL and use MPI\_REAL. The second is to use selected\_real\_kind or selected\_int\_kind and with the functions of the previous section. The third is to declare a variable that is known to be the same size on all architectures (e.g., selected\_real\_kind(12) on almost all compilers will result in an 8-byte representation). The fourth is to carefully check representation size before communicated and handshaking between sender and receiver to agree on a size.

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

Unofficial Draft for Comment Only

This section discusses a number of problems that may arise when using MPI in a Fortran program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It is intended to clarify, not add to, this standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these may cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. With Fortran 2008 and the new semantics defined in 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.

- 1. An MPI subroutine with a choice argument may be called with different argument types. When using the mpi\_f08 module together with a compiler that supports Fortran 2008 + TS 29113, this problem is resolved.
- 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(..).
- 3. 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.

1 2 3 4 5 6	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 Sec- tion 2.5.4 for more information.
6 7 8 9 10 11 12 13 14	<ul> <li>6. The memory allocation routine MPI_ALLOC_MEM cannot be used from Fortran 77/90/95 without a language extension (for example, Cray pointers) that allows the allocated memory to be associated with a Fortran variable. Therefore, address sized integers were used in MPI-2.0 – MPI-2.2. In Fortran 2003, TYPE(C_PTR) entities were added, which allow a standard-conforming implementation of the semantics of MPI_ALLOC_MEM. In MPI-3.0 and later, MPI_ALLOC_MEM has an additional, overloaded interface to support this language feature. The use of Cray pointers is deprecated. The mpi_f08 module only supports TYPE(C_PTR) pointers.</li> </ul>
15	Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.
16 17	• MPI identifiers exceed 6 characters.
18	• MPI identifiers may contain underscores after the first character.
19 20 21	• MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.
22 23 24 25	• 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.
26 27 28 29 30 31 32 33 34 35 36 37	MPI-1 contained several routines that take address-sized information as input or return address-sized information as output. In C such arguments were of type MPI_Aint and in Fortran of type INTEGER. On machines where integers are smaller than addresses, these routines can lose information. In MPI-2 the use of these functions has been deprecated and they have been replaced by routines taking INTEGER arguments of KIND=MPI_ADDRESS_KIND. A number of MPI-2 functions also take INTEGER arguments of non-default KIND. See Section 2.6 and Section 5.1.1 for more information. Sections 19.1.11 through 19.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 19.1.7.
38 39	19.1.11 Problems Due to Strong Typing
40 41 42 43 44 45	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 19.1.6). 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(*),
16	

<sup>46</sup> DIMENSION(..), i.e., as assumed-type and assumed-rank dummy arguments.

<sup>47</sup> Using INCLUDE 'mpif.h', the following code fragment is technically invalid and may
 <sup>48</sup> generate a compile-time error.

integer i(5) real x(5)	1
	3
call mpi_send(x, 5, MPI_REAL,)	4
call mpi_send(i, 5, MPI_INTEGER,)	Ę
	e

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.

19.1.12 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:

• If MPI\_SUBARRAYS\_SUPPORTED equals .TRUE.:

Choice buffer arguments are declared as TYPE(\*), DIMENSION(...). For example, consider the following code fragment:

```
REAL s(100), r(100)
CALL MPI_Isend(s(1:100:5), 3, MPI_REAL, ..., rq, ierror)
CALL MPI_Wait(rq, status, ierror)
CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL, ..., rq, ierror)
CALL MPI_Wait(rq, status, ierror)
```

In this case, the individual elements s(1), s(6), and s(11) are sent between the start of MPI\_ISEND and the end of MPI\_WAIT even though the compiled code will not copy s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code will pass a descriptor to MPI\_ISEND that allows MPI to operate directly on s(1), s(6), 

 $s(11), \ldots, s(96)$ . The called MPI\_ISEND routine will take only the first three of these elements due to the type signature "3, MPI\_REAL".

<sup>3</sup> All nonblocking MPI functions (e.g., MPI\_ISEND, MPI\_PUT,

MPI\_FILE\_WRITE\_ALL\_BEGIN) behave as if the user-specified elements of choice buffers are copied to a contiguous scratch buffer in the MPI runtime environment. All datatype descriptions (in the example above, "3, MPI\_REAL") read and store data from and to this virtual contiguous scratch buffer. Displacements in MPI derived datatypes are relative to the beginning of this virtual contiguous scratch buffer. Upon completion of a nonblocking receive operation (e.g., when MPI\_WAIT on a corresponding MPI\_Request returns), it is as if the received data has been copied from the virtual contiguous scratch buffer back to the non-contiguous application buffer. In the example above, r(1), r(6), and r(11) are guaranteed to be defined with the received data when MPI\_WAIT returns.

Note that the above definition does not supercede restrictions about buffers used with nonblocking operations (e.g., those specified in Section 3.7.2).

Advice to implementors. The Fortran descriptor for TYPE(\*), DIMENSION(..) arguments contains enough information that, if desired, the MPI library can make a real contiguous copy of non-contiguous user buffers when the nonblocking operation is started, and release this buffer not before the nonblocking communication has completed (e.g., the MPI\_WAIT routine). Efficient implementations may avoid such additional memory-to-memory data copying. (End of advice to implementors.)

*Rationale.* If MPI\_SUBARRAYS\_SUPPORTED equals .TRUE., non-contiguous buffers are handled inside the MPI library instead of by the compiler through argument association conventions. Therefore, the scope of MPI library scratch buffers can be from the beginning of a nonblocking operation until the completion of the operation although beginning and completion are implemented in different routines. (*End of rationale.*)

• If MPI\_SUBARRAYS\_SUPPORTED equals .FALSE.:

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.<sup>1</sup>

Because MPI dummy buffer arguments are assumed-size arrays if MPI\_SUBARRAYS\_SUPPORTED equals .FALSE., this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

real a(100)
call MPI\_IRECV(a(1:100:2), MPI\_REAL, 50, ...)

Since the first dummy argument to MPI\_IRECV is an assumed-size array (<type> buf(\*)), the array section a(1:100:2) is copied to a temporary before being passed to MPI\_IRECV, so that it is contiguous in memory. MPI\_IRECV returns immediately, and data is copied from the temporary back into the array a. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem for MPI\_ISEND since the temporary array may be deallocated before the data has all been sent from it.

Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a "simply contiguous" section such as A(1:N) of such an array. ("Simply contiguous" is defined in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontiguous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

According to the Fortran 2008 Standard, Section 6.5.4, a "simply contiguous" array section is

name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )

That is, there are zero or more dimensions that are selected in full, then one dimension selected without a stride, then zero or more dimensions that are selected with a simple subscript. The compiler can detect from analyzing the source code that the array is contiguous. Examples are

Because of Fortran's column-major ordering, where the first index varies fastest, a "simply contiguous" section of a contiguous array will also be contiguous.

The same problem can occur with a scalar argument. A compiler may make a copy of scalar dummy arguments within a called procedure when passed as an actual argument to a choice buffer routine. That this can cause a problem is illustrated by the example

 $^{24}$ 

 $45 \\ 46$ 

<sup>&</sup>lt;sup>1</sup>Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the CONTIGUOUS attribute.

1	real :: a
2	call user1(a,rq)
3	call MPI_WAIT(rq,status,ierr)
4	write (*,*) a
5	
6	subroutine user1(buf,request)
7	call MPI_IRECV(buf,,request,)
8	end
9	
10	If a is copied, MPI_IRECV will alter the copy when it completes the communication
11	and will not alter a itself.
12	Note that copying will almost certainly occur for an argument that is a non-trivial
13	expression (one with at least one operator or function call), a section that does not
14	select a contiguous part of its parent (e.g., $A(1:n:2)$ ), a pointer whose target is such
15	
16	a section, or an assumed-shape array that is (directly or indirectly) associated with
17	such a section.
18	If a compiler option exists that inhibits copying of arguments, in either the calling or
19	called procedure, this must be employed.
20	If a compiler makes copies in the calling procedure of arguments that are explicit-
21	shape or assumed-size arrays, "simply contiguous" array sections of such arrays, or
22	scalars, and if no compiler option exists to inhibit such copying, then the compiler
23	
23	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
25	is no compiler option to inhibit this, then this compiler cannot be used for applications
26	that use memory references across subroutine calls as in the example above.
27	
28 29	19.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts
30	Fortran arrays with <b>vector</b> subscripts describe subarrays containing a possibly irregular
31	set of elements
32	
33	REAL a(100)
34	CALL MPI_Send(A((/7,9,23,81,82/)), 5, MPI_REAL,)
35	Fortran arrays with a vector subscript must not be used as actual choice buffer argu-
36	ments in any nonblocking or split collective MPI operations. They may, however, be used
37	in blocking MPI operations.
38	
39	19.1.14 Special Constants
40	
41	MPI requires a number of special "constants" that cannot be implemented as normal Fortran
42	constants, e.g., MPI_BOTTOM. The complete list can be found in Section 2.5.4. In C, these
43	are implemented as constant pointers, usually as NULL and are used where the function
44	prototype calls for a pointer to a variable, not the variable itself.
45	

<sup>45</sup> In Fortran, using special values for the constants (e.g., by defining them through <sup>46</sup> **parameter** statements) is not possible because an implementation cannot distinguish these <sup>47</sup> values from valid data. Typically these constants are implemented as predefined static vari-<sup>48</sup> ables (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.

#### 19.1.15 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
disp(3) = disp(3) - base
disp(4) = disp(4) - base
blocklen(1) = 1
blocklen(2) = 1
blocklen(3) = 1
blocklen(4) = 1
type(1) = MPI_INTEGER
type(2) = MPI_REAL
type(3) = MPI_DOUBLE_PRECISION
type(4) = MPI_LOGICAL
```

1

 $\mathbf{2}$ 

3

4

5 6

7

8 9

10

11

12

13

14

15 16

17

18

19

20

21

22 23

24

25

26 27

28

29

30

31 32

33

34

35

36

37 38

39

40

41

42 43

44

45

46

```
1
     call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
\mathbf{2}
     call MPI_TYPE_COMMIT(newtype, ierr)
3
4
     call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
5
     ! or
6
     call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
\overline{7}
     ! expects that base == address(foo%i) == address(foo)
8
9
     call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
10
     call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
11
     extent = disp(2) - disp(1)
12
     1b = 0
13
     call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
14
     call MPI_TYPE_COMMIT(newarrtype, ierr)
15
16
     call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
17
```

Using the derived type variable foo instead of its first basic type element foo%i may be impossible if the MPI library implements choice buffer arguments through overloading instead of using TYPE(\*), DIMENSION(..), or through a non-standardized extension such as !\$PRAGMA IGNORE\_TKR; see Section 19.1.6.

To use a derived type in an array requires a correct extent of the datatype handle 22 to take care of the alignment rules applied by the compiler. These alignment rules may 23imply that there are gaps between the components of a derived type, and also between the 24subsuguent elements of an array of a derived type. The extent of an interoperable derived 25type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may 26be different because C and Fortran may apply different alignment rules. As recommended 27in the advice to users in Section 5.1.6, one should add an additional fifth structure element 28with one numerical storage unit at the end of this structure to force in most cases that 29 the array of structures is contiguous. Even with such an additional element, one should 30 keep this resizing due to the special alignment rules that can be used by the compiler for  $^{31}$ structures, as also mentioned in this advice. 32

Using the extended semantics defined in TS 29113, it is also possible to use entities or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer arguments; some additional constraints must be observed, e.g., no ALLOCATABLE or POINTER type components may exist. In this case, the base address in the example must be changed to become the address of foo instead of foo%i, because the Fortran compiler may rearrange type components or add padding. Sending the structure foo should then also be performed by providing it (and not foo%i) as actual argument for MPI\_Send.

40 41

#### 19.1.16 Optimization Problems, an Overview

<sup>42</sup> MPI provides operations that may be hidden from the user code and run concurrently <sup>43</sup> with it, accessing the same memory as user code. Examples include the data transfer <sup>44</sup> for an MPI\_IRECV. The optimizer of a compiler will assume that it can recognize periods <sup>46</sup> when a copy of a variable can be kept in a register without reloading from or storing to <sup>47</sup> memory. When the user code is working with a register copy of some variable while the <sup>48</sup> 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 persis	stent requ	uests (Nor	nbl.).		5
• Use of one-sided routines (1-sided).					7
• Use of MPI parallel file I/O split coll	lective op	erations (	(Split).	<u> </u>	9
• Use of MPI_BOTTOM together with a ative displacements between two variations of the second		-			rel- 11
The following compiler optimization s lems in MPI applications:	trategies	(valid for	• serial o	code) may cause pr	rob- 13 14
• Code movement and register optimiz	ation pro	oblems; se	e Sectio	on 19.1.17.	16
• Temporary data movement and temp	orary me	mory mod	lificatio	ns; see Section 19.1	.18. <sub>18</sub>
• Permanent data movement (e.g., thr	ough garl	oage colle	ction);	see Section 19.1.19	. 19
Table 19.2 shows the only usage areas whe	ere these	optimizat	ion prol	blems may occur.	20 21
Optimization		ay cause llowing us 1-sided	sage are		22 23 24
Code movement	yes	yes	Split no	yes bottom	25 26
and register optimization	y 00	<i>y</i> 00	110		27
Temporary data movement Permanent data movement	yes	yes	yes	no	28 29
i enhanent data movement	yes	yes	yes	yes	30
Table 19.2. Occurrence of Fortran optimization problems in several usage areas				31 32	
The solutions in the following sections					
The solutions in the following sections are based on compromises:					
• to minimize the burden for the application programmer, e.g., as shown in Sections "Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on pages 816–820, 35 36 37 37 36 37 37 36 37 37 36 37 37 36 37 37 36 37 37 36 37 37 37 36 37 37 37 37 37 37 37 37 37 37					
• to minimize the drawbacks on compiler based optimization, and					
• to minimize the requirements defined in Section 19.1.7.					
					41 42
19.1.17 Problems with Code Movement	and Reg	ister Opti	mizatio	n	43
Nonblocking Operations					44 45
If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the compiler will assume that it cannot be modified by a called subroutine unless it is an actual argument of the call. In the most common linkage convention, the subroutine is expected 40					

1 Example 19.1 Fortran 90 register optimization—extreme.  $\mathbf{2}$ 3 Source compiled as or compiled as 4 REAL :: buf, b1 REAL :: buf, b1 REAL :: buf, b1 5call MPI\_IRECV(buf,..req) call MPI\_IRECV(buf,..req) call MPI\_IRECV(buf,..req) 6 register = buf b1 = buf $\overline{7}$ call MPI\_WAIT(req,..) call MPI\_WAIT(req,..) call MPI\_WAIT(req,..) 8 b1 = bufb1 = register 9 10 11to save and restore certain registers. Thus, the optimizer will assume that a register which 12held a valid copy of such a variable before the call will still hold a valid copy on return. 13Example 19.1 shows extreme, but allowed, possibilities. MPI\_WAIT on a concurrent 14thread modifies buf between the invocation of MPI\_IRECV and the completion of MPI\_WAIT. 15But the compiler cannot see any possibility that buf can be changed after MPI\_IRECV has 16returned, and may schedule the load of buf earlier than typed in the source. The compiler 17has no reason to avoid using a register to hold **buf** across the call to MPI\_WAIT. It also may 18 reorder the instructions as illustrated in the rightmost column. 1920**Example 19.2** Similar example with MPI\_ISEND 21with a possible MPI-internal Source compiled as 22 execution sequence 23 $^{24}$ REAL :: buf, copy REAL :: buf, copy REAL :: buf, copy 25buf = val buf = val buf = val 26call MPI\_ISEND(buf,..req) call MPI\_ISEND(buf,..req) addr = &buf copy = buf 27copy = bufcopy= buf buf = val\_overwrite buf = val\_overwrite 28call MPI\_WAIT(req,..) call send(\*addr) ! within call MPI\_WAIT(req,..) 29 ! MPI\_WAIT 30 buf = val\_overwrite  $^{31}$ 32 Due to valid compiler code movement optimizations in Example 19.2, the content of 33 34buf may already have been overwritten by the compiler when the content of buf is sent. The code movement is permitted because the compiler cannot detect a possible access to 35 buf in MPI\_WAIT (or in a second thread between the start of MPI\_ISEND and the end of 36 MPI\_WAIT). 37 Such register optimization is based on moving code; here, the access to buf was moved 38 39 from after MPI\_WAIT to before MPI\_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined. 40This register optimization/code movement problem for nonblocking operations does 41 not occur with MPI parallel file I/O split collective operations, because in the 42MPI\_XXX\_BEGIN and MPI\_XXX\_END calls, the same buffer has to be provided as an 43 actual argument. The register optimization / code movement problem for MPI\_BOTTOM 44and derived MPI datatypes may occur in each blocking and nonblocking communication 45call, as well as in each parallel file I/O operation. 464748

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 19.1.17.

#### One-sided Communication

An example with instruction reordering due to register optimization can be found in Section 12.7.4.

#### 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 19.3 shows what Fortran compilers are allowed to do.

Example 19.3 Fortran 90 register optimization.

This source	can be compiled as:	1
call MPI_GET_ADDRESS(buf,bufaddr, ierror)	<pre>call MPI_GET_ADDRESS(buf,)</pre>	2
call MPI_TYPE_CREATE_STRUCT(1,1,	call MPI_TYPE_CREATE_STRUCT()	4
bufaddr, MPI_REAL,type,ierror)	*	į
<pre>call MPI_TYPE_COMMIT(type,ierror)</pre>	<pre>call MPI_TYPE_COMMIT()</pre>	:
val_old = buf	register = buf	:
	val_old = register	3
<pre>call MPI_RECV(MPI_BOTTOM,1,type,) val new = buf</pre>	call MPI_RECV(MPI_BOTTOM,)	3
val_new = buf	val_new = register	3

In Example 19.3, 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 19.4, 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.

 $\mathbf{2}$ 

1 Example 19.4 Similar example with MPI\_SEND  $\mathbf{2}$ 3 This source ... can be compiled as: 4 ! buf contains val\_old ! buf contains val\_old 5buf = val\_new 6 call MPI\_SEND(MPI\_BOTTOM,1,type,...) call MPI\_SEND(...) 7 ! with buf as a displacement in type ! i.e. val\_old is sent 8 1 9 ! buf=val\_new is moved to here 10 ! and detected as dead code 11 ! and therefore removed 121 13 buf = val\_overwrite buf = val\_overwrite 141516

17 Solutions

<sup>18</sup> The following sections show in detail how the problems with code movement and register <sup>19</sup> optimization can be portably solved. Application writers can partially or fully avoid these <sup>20</sup> compiler optimization problems by using one or more of the special Fortran declarations <sup>21</sup> with the send and receive buffers used in nonblocking operations, or in operations in which <sup>22</sup> MPI\_BOTTOM is used, or if datatype handles that combine several variables are used:

- Use of the Fortran ASYNCHRONOUS attribute.
- Use of the helper routine MPI\_F\_SYNC\_REG, or an equivalent user-written dummy routine.

• Declare the buffer as a Fortran module variable or within a Fortran common block.

• Use of the Fortran VOLATILE attribute.

31Each of these methods solves the problems of code movement and register optimization, 32 but may incur various degrees of performance impact, and may not be usable in every 33 application context. These methods may not be guaranteed by the Fortran standard, but 34 they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated 35 compiler suite according to the requirements listed in Section 19.1.7. The performance 36 impact of using MPI\_F\_SYNC\_REG is expected to be low, that of using module variables 37 or the ASYNCHRONOUS attribute is expected to be low to medium, and that of using the 38 VOLATILE attribute is expected to be high or very high. Note that there is one attribute 39 that cannot be used for this purpose: the Fortran TARGET attribute does not solve code 40 movement problems in MPI applications.

 $41 \\ 42$ 

43

24 25

26

27 28

29

30

# The Fortran ASYNCHRONOUS Attribute

<sup>44</sup> Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping <sup>45</sup> unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed <sup>46</sup> while the buffer is affected by a pending asynchronous Fortran input/output operation (since <sup>47</sup> Fortran 2003) or by an asynchronous communication (TS 29113 extension). Without the <sup>48</sup> 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 19.1.6), 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 19.5 Case (a) on page 823, the read accesses to b within function(b(i-1), b(i), b(i+1)) cannot be moved by compiler optimizations to before the wait call because b was declared as ASYNCHRONOUS. Note that only the elements 0, 1, 100, and 101 of b are involved in asynchronous communication but by definition, the total variable b is the pending communication affector and is usable for input and output asynchronous communication between the MPI\_IXXX routines and MPI\_Waitall. Case (a) works fine because the read accesses to b occur after the communication has completed.

In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to a pending communication affector while input communication (i.e., the two MPI\_Irecv calls) is pending. This is a contradiction to the rule that *for input communication*, a *pending communication affector shall not be referenced*. The problem can be solved by using separate variables for the halos and the inner array, or by splitting a common array into disjoint subarrays which are passed through different dummy arguments into a subroutine, as shown in Example 19.9.

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 19.3 and Example 19.4, can also be solved by declaring the buffer **buf** with the ASYNCHRONOUS attribute.

#### Unofficial Draft for Comment Only

 $\mathbf{5}$ 

 $^{24}$ 

 $45 \\ 46$ 

In some MPI routines, a buffer dummy argument is defined as ASYNCHRONOUS to guarantee passing by reference, provided that the actual argument is also defined as ASYNCHRONOUS. Calling MPI\_F\_SYNC\_REG The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer

as an actual argument. The MPI library provides the MPI\_F\_SYNC\_REG routine for this purpose; see Section 19.1.8.

• The problems illustrated by the Examples 19.1 and 19.2 can be solved by calling MPI\_F\_SYNC\_REG(buf) once immediately after MPI\_WAIT.

Example 19.1	Example 19.2
can be solved with	can be solved with
<pre>call MPI_IRECV(buf,req)</pre>	buf = val
	<pre>call MPI_ISEND(buf,req)</pre>
	copy = buf
<pre>call MPI_WAIT(req,)</pre>	<pre>call MPI_WAIT(req,)</pre>
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
b1 = buf	<pre>buf = val_overwrite</pre>

- The call to MPI\_F\_SYNC\_REG(buf) prevents moving the last line before the MPI\_WAIT call. Further calls to MPI\_F\_SYNC\_REG(buf) are not needed because it is still correct if the additional read access copy=buf is moved below MPI\_WAIT and before buf=val\_overwrite.
- The problems illustrated by the Examples 19.3 and 19.4 can be solved with two additional MPI\_F\_SYNC\_REG(buf) statements; one directly before MPI\_RECV/ MPI\_SEND, and one directly after this communication operation.

Example 19.3	Example 19.4
can be solved with	can be solved with
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
<pre>call MPI_RECV(MPI_BOTTOM,)</pre>	<pre>call MPI_SEND(MPI_BOTTOM,)</pre>
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)

The first call to MPI\_F\_SYNC\_REG(buf) is needed to finish all load and store references to buf prior to MPI\_RECV/MPI\_SEND; the second call is needed to assure that any subsequent access to buf is not moved before MPI\_RECV/MPI\_SEND.

In the example in Section 12.7.4, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 19.1, 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.

# Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 

Source of Process 1	Source of Process 2
bbbb = 777	buff = 999
	call MPI_F_SYNC_REG(buff)
call MPI_WIN_FENCE	call MPI_WIN_FENCE
call MPI_PUT(bbbb	
into buff of process 2)	
call MPI_WIN_FENCE	call MPI_WIN_FENCE
call MPI_F_SYNC_REG(bbbb)	call MPI_F_SYNC_REG(buff)
	ccc = buff

• The temporary memory modification problem, i.e., Example 19.6, 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)	
<pre>call MPI_RECV(MPI_BOTTOM,)</pre>	
call DD(buf)	

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer MPI\_F\_SYNC\_REG or one of the other possibilities. In an existing application, calls to such a user-written routine should be substituted by a call to MPI\_F\_SYNC\_REG because the user-written routine may not be implemented in accordance with the rules specified in Section 19.1.7.

#### 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. 48

#### Unofficial Draft for Comment Only

• Unfortunately, this method does **not** solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.

# The (Poorly Performing) Fortran VOLATILE Attribute

The VOLATILE attribute gives the buffer or variable the properties needed to avoid register optimization or code movement problems, but it may inhibit optimization of any code containing references or definitions of the buffer or variable. On many modern systems, the performance impact will be large because not only register, but also cache optimizations will not be applied. Therefore, use of the VOLATILE attribute to enforce correct execution of MPI programs is discouraged.

13 14

1

 $\mathbf{2}$ 

3

4 5

6

## 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.

- $^{21}$
- 22
- $^{23}$

*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.*)

24

# <sup>25</sup> 19.1.18 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 19.5, Case (b) on page 823. Example 19.6 also shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 19.7, buf(100,100) from Example 19.6 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 19.8 shows a second possible optimization. The whole array is temporarily moved to local\_buf. 19.8 shows a second possible optimization

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).

42 43 44

45

46 47

48

39

40

41

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 12.
- 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 MPI\_XXX\_BEGIN and MPI\_XXX\_END calls; see Section 14.4.5.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 816 of Section 19.1.17, 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 19.9 and in Example 19.10.

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 19.6 and 19.7.

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 restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and together with asynchronous operations outside the scope of Fortran. If such a feature 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 19.9 (which is a solution for the problem shown in Example 19.5 and in Example 19.10 (which is a solution for the problem shown in Example 19.8), the array 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 area are strided arrays. Those can be used in nonblocking communication only with a TS 29113 based MPI library.

#### 19.1.19 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:

#### 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

- - 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 19.1.7.

 $\overline{7}$ 

# 19.1.20 Comparison with C

In C, subroutines which modify variables that are not in the argument list will not cause register optimization problems. This is because taking pointers to storage objects by using the & operator and later referencing the objects by indirection on the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe. Problems due to temporary memory modifications can also occur in C. As above, the best advice is to avoid the problem: use different variables for buffers in nonblocking MPI operations and computation that is executed while a nonblocking operation is pending.

**Unofficial Draft for Comment Only** 

 $^{24}$ 

**Example 19.5** Protecting nonblocking communication with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
                               ! elements 1 and 100 are newly computed
REAL :: bnew(0:101)
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
                                                                                 21
#endif
                                                                                 22
                                                                                 23
#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_Waital1(4, req, ...)
  i=1 ! compute leftmost element
                                                                                 30
    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
                                                                                 34
                                                                                 35
                                                                                 36
Example 19.6 Overlapping Communication and Computation.
                                                                                 37
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,...)
```

2 3 4

5

6

9

10

11

12

13

1415

16

17

18

19

20

 $^{24}$ 

25

26

27

28

29

31

32

33

38

39

40

41

42

43

44

45

46

```
1
     Example 19.7 The compiler may substitute the nested loops through loop fusion.
\mathbf{2}
3
     REAL :: buf(100,100), buf_1dim(10000)
4
     EQUIVALENCE (buf(1,1), buf_1dim(1))
\mathbf{5}
     CALL MPI_Irecv(buf(1,1:100),..., req,...)
6
     tmp(1:100) = buf(1,1:100)
\overline{7}
     DO j=1,10000
8
       buf_1dim(h)=...
9
     END DO
10
     buf(1,1:100) = tmp(1:100)
11
     CALL MPI_Wait(req,...)
12
13
14
15
16
17
18
19
20
21
22
23
^{24}
25
26
27
     Example 19.8 Another optimization is based on the usage of a separate memory storage
28
     area, e.g., in a GPU.
29
30
     REAL :: buf(100,100), local_buf(100,100)
31
     CALL MPI_Irecv(buf(1,1:100),..., req,...)
32
     local_buf = buf
33
     DO j=1,100
34
       DO i=2,100
35
          local_buf(i,j)=...
36
        END DO
37
     END DO
38
     buf = local_buf ! may overwrite asynchronously received
39
                      ! data in buf(1,1:100)
40
     CALL MPI_Wait(req,...)
41
42
43
44
45
46
47
48
```

**Example 19.9** Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

```
4
USE mpi_f08
                                                                                    5
REAL :: b(0:101)
                      ! elements 0 and 101 are halo cells
                                                                                    6
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                    7
INTEGER :: i
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
                                                                                    9
i=1 ! compute leftmost element
                                                                                    10
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                    11
i=100 ! compute rightmost element
                                                                                    12
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                    13
END
                                                                                    14
                                                                                    15
SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
                                                                                    16
USE mpi_f08
                                                                                    17
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
                                                                                    18
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                    19
TYPE(MPI_Request) :: req(4)
                                                                                    20
INTEGER :: left, right, i
                                                                                    21
CALL MPI_Cart_shift(...,left, right,...)
                                                                                    22
CALL MPI_Irecv(b_lefthalo ( 0), ..., left, ..., req(1), ...)
                                                                                    23
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
                                                                                    ^{24}
! b_lefthalo and b_righthalo is written asynchronously.
                                                                                    25
! There is no other concurrent access to b_lefthalo and b_righthalo.
                                                                                    26
CALL MPI_Isend(b_inner( 1),
                                   ..., left, ..., req(3), ...)
                                                                                    27
CALL MPI_Isend(b_inner(100),
                                   ..., right, ..., req(4), ...)
                                                                                    28
                                                                                    29
DO i=2,99 ! compute only elements for which halo data is not needed
                                                                                    30
  bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
                                                                                    31
  ! b_inner is read and sent at the same time.
                                                                                    32
  ! This is allowed based on the rules for ASYNCHRONOUS.
                                                                                    33
END DO
                                                                                    34
CALL MPI_Waitall(4, req,...)
                                                                                    35
END SUBROUTINE
                                                                                    36
                                                                                    37
                                                                                    38
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
                                                                                    46
                                                                                    47
```

1

2

```
1
     Example 19.10 Protecting GPU optimizations with the ASYNCHRONOUS attribute.
\mathbf{2}
3
     USE mpi_f08
4
     REAL :: buf(100,100)
\mathbf{5}
     CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
6
     END
\overline{7}
8
     SUBROUTINE separated_sections(buf_halo, buf_inner)
9
     REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
10
     REAL :: buf_inner(2:100,1:100)
11
     REAL :: local_buf(2:100,100)
12
13
     CALL MPI_Irecv(buf_halo(1,1:100),..., req,...)
14
     local_buf = buf_inner
15
     DO j=1,100
16
       DO i=2,100
17
          local_buf(i,j)=...
18
        END DO
19
     END DO
20
     buf_inner = local_buf ! buf_halo is not touched!!!
21
^{22}
     CALL MPI_Wait(req,...)
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
```

19.2	Support for Large Count and Large Byte Displacement in MPI Lan- guage Bindings	1 2 3	
	lowing types, which were used prior to MPI-4.0, have been deemed too small to hold that applications wish to use:	4 5 6	
• T	The C int type and the Fortran INTEGER type were used for <i>count</i> parameters.	7	
• Т	The C int type and the Fortran INTEGER type were used for some parameters that	8	
	epresent byte displacement in memory.	9	
		10	
	The C MPI_Aint type and the Fortran INTEGER(KIND=MPI_ADDRESS_KIND) type were	11	
	sed for some parameters that represent <i>byte displacement</i> in files (e.g., in constructors f MPI datatypes that can be used with files).	12	
		13	
	er to avoid breaking backwards compatibility, this version of MPI supports larger	14	
01	via separate additional MPI procedures in C (suffixed with "_c") and via interface	15 16	
* °	propriate in Fortran when using USE mpi_f08. No polymorphic support for larger	10	
	s provided in Fortran when using mpif.h and use mpi. r the large count versions of three datatype constructors,	18	
	YPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, and	19	
	YPE_CREATE_STRUCT, absolute addresses shall not be used to specify byte dis-	20	
	ents since the parameter is of type MPI_COUNT instead of type MPI_AINT (see Sec-	21	
tion $2.5$		22	
	addition, the functions MPI_TYPE_GET_ENVELOPE and	23	
	YPE_GET_CONTENTS also support large count types via <i>additional parameters</i> in	24	
separat	e additional MPI procedures in C (suffixed with "_c") and interface polymorphism	25	
in Fort	ran when using USE $mpi_f08$ (see Section 5.1.13). rther, the callbacks MPL USER EUNCTION and	Datare	en la
Iu	the callbacks with over tore and		
	ATAREP_CONVERSION_FUNCTION also support large count types via separate ad-	28	#-error!
	l callback prototypes in C (suffixed with "_c") and multiple abstract interfaces in	29	#-other
	when using USE mpi_f08 (see Sections 6.9.5 and 14.5.3, respectively).		a callback /pe, use
	C bindings, for each MPI procedure that had at least one <i>count</i> or <i>byte displacement</i>		t macro!
-	eter that used the int and/or MPI_Aint types prior to MPI-4.0, an additional MPI ure is provided, with the same name but suffixed by "_c". The MPI procedure	33	Done
_	t the "_c" token has the same name and parameter types as versions prior to MPI-	34	in
	e "_c" suffixed MPI procedure has MPI_Count for all <i>count</i> parameters, MPI_Aint for	35	PR449
	eters that represent byte displacement in memory, MPI_Offset for parameters that	36	
-	nt byte displacement in files, and MPI_Count for parameters that may represent byte	37	
-	ement in both memory and files.	38	
-	Fortran, when using USE mpi_f08, for each MPI procedure that had at least one	39	
count of	or byte displacement parameter that used the INTEGER or	40	
INTEGE	R(KIND=MPI_ADDRESS_KIND) types prior to MPI-4.0, a polymorphic interface con-	41	
0	two specific procedures is provided. One of the specific procedures has the same	42	
	and dummy parameter types as in versions prior to MPI-4.0. INTEGER and/or $M$	43	
	R(KIND=MPI_ADDRESS_KIND) for <i>count</i> and <i>byte displacement</i> parameters. The other	44	
-	procedure has the same name followed by "_c", and then suffixed by the token	45 46	
-	d in Table 19.1 for USE mpi_f08. It also has INTEGER(KIND=MPI_COUNT_KIND)	40 47	
	<i>count</i> parameters, INTEGER(KIND=MPI_ADDRESS_KIND) for parameters that repre- te displacement in memory, INTEGER(KIND=MPI_OFFSET_KIND) for parameters that	48	

<sup>1</sup> represent *byte displacement* in files, and INTEGER(KIND=MPI\_COUNT\_KIND) for parame-<sup>2</sup> ters that may represent *byte displacement* in both memory and files (for more details <sup>3</sup> on specific Fortran procedure names and related calling conventions, refer to Table 19.1 <sup>4</sup> in Section 19.1.5). There is one exception: if the type signatures of the two specific <sup>5</sup> procedures are identical (e.g., if INTEGER(KIND=MPI\_COUNT\_KIND) is the same type as <sup>6</sup> INTEGER(KIND=MPI\_ADDRESS\_KIND)), then the implementation shall not provide the "\_c" <sup>7</sup> specific procedure.

<sup>8</sup> It is erroneous to directly invoke the "\_c" specific procedures in the Fortran mpi\_f08 <sup>9</sup> module with the exception of the following procedures: MPI\_Op\_create\_c and <sup>10</sup> MPI\_Register\_datarep\_c.

<sup>11</sup> In older Fortran bindings (mpif.h and use mpi), no new interfaces and no new specific <sup>12</sup> procedures for larger types are provided beyond what existed in MPI-4.0; all MPI procedures <sup>13</sup> have the same types as in the versions prior to MPI-4.0.

14 15

16

## 19.3 Language Interoperability

## <sup>17</sup><sub>18</sub> 19.3.1 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.

28 Th

 $^{31}$ 

32

33

34

35

38

39

40

There are several issues that need to be addressed in order to achieve interoperability.

<sup>29</sup><sub>30</sub> 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.

<sup>41</sup><sub>42</sub> 19.3.2 Assumptions

<sup>43</sup> We assume that conventions exist for programs written in one language to call routines <sup>44</sup> written in another language. These conventions specify how to link routines in different <sup>45</sup> languages into one program, how to call functions in a different language, how to pass <sup>46</sup> arguments between languages, and the correspondence between basic data types in different <sup>47</sup> languages. In general, these conventions will be implementation dependent. Furthermore, <sup>48</sup> 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 address-sized 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.

#### 19.3.3 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 executing processes. Use of the Fortran version of MPI\_INIT to initialize MPI may result in a loss of this ability. (*End of advice to users.*)

The function MPI\_INITIALIZED returns the same answer in all languages. The function MPI\_FINALIZE finalizes the MPI environments for all languages. The function MPI\_FINALIZED returns the same answer in all languages.

The function MPI\_ABORT kills processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by MPI\_INIT. E.g., MPI\_COMM\_WORLD carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

Information can be added to info objects in one language and retrieved in another.

Advice to users. The use of several languages in one MPI program may require the use of special options at compile and/or link time. (*End of advice to users.*)

Advice to implementors. Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code need perform initialization for a language only if that language library is loaded. (*End of advice to implementors.*)

#### 19.3.4 Transfer of Handles

Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C handles in Fortran.

The type definition MPI\_Fint is provided in C for an integer of the size that matches a Fortran INTEGER; usually, MPI\_Fint will be equivalent to int. With the Fortran mpi module or the mpif.h include file, a Fortran handle is a Fortran INTEGER value that can be used in the following conversion functions. With the Fortran mpi\_f08 module, a Fortran handle is a BIND(C) derived type that contains an INTEGER component named MPI\_VAL. This INTEGER value can be used in the following conversion functions.

The following functions are provided in C to convert from a Fortran communicator handle (which is an integer) to a C communicator handle, and vice versa. See also Section 2.6.4.

 $\mathbf{2}$ 

 $^{24}$ 

1 2	C binding MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
3 4 5 6 7 8	If comm is a valid Fortran handle to a communicator, then MPI_Comm_f2c returns a valid C handle to that same communicator; if comm = MPI_COMM_NULL (Fortran value), then MPI_Comm_f2c returns a null C handle; if comm is an invalid Fortran handle, then MPI_Comm_f2c returns an invalid C handle. MPI_Fint MPI_Comm_c2f (MPI_Comm comm)
9 10 11 12 13	The function MPI_Comm_c2f translates a C communicator handle into a Fortran handle to the same communicator; it maps a null handle into a null handle and an invalid handle into an invalid handle. Similar functions are provided for the other types of opaque objects. MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
14 15	MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
16	MPI_Group MPI_Group_f2c(MPI_Fint group)
17 18	MPI_Fint MPI_Group_c2f(MPI_Group group)
19	MPI_Request MPI_Request_f2c(MPI_Fint request)
20 21	MPI_Fint MPI_Request_c2f(MPI_Request request)
22	MPI_File MPI_File_f2c(MPI_Fint file)
23 24	MPI_Fint MPI_File_c2f(MPI_File file)
25	MPI_Win MPI_Win_f2c(MPI_Fint win)
26 27	MPI_Fint MPI_Win_c2f(MPI_Win win)
28 29	MPI_Op MPI_Op_f2c(MPI_Fint op)
30	MPI_Fint MPI_Op_c2f(MPI_Op op)
31 32	MPI_Info MPI_Info_f2c(MPI_Fint info)
33	MPI_Fint MPI_Info_c2f(MPI_Info info)
34 35	MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
36	MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
37 38	MPI_Message MPI_Message_f2c(MPI_Fint message)
39	MPI_Fint MPI_Message_c2f(MPI_Message message)
40 41	MPI_Session MPI_Session_f2c(MPI_Fint session)
42	MPI_Fint MPI_Session_c2f(MPI_Session session)
43 44	
45	
46	
47 48	
48	

**Example 19.11** The example below illustrates how the Fortran MPI function MPI\_TYPE\_COMMIT can be implemented by wrapping the C MPI function MPI\_Type\_commit with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.

```
! FORTRAN PROCEDURE
SUBROUTINE MPI_TYPE_COMMIT(DATATYPE, IERR)
INTEGER :: DATATYPE, IERR
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
RETURN
END
/* C wrapper */
void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr)
{
    MPI_Datatype datatype;
    datatype = MPI_Type_f2c(*f_handle);
    *ierr = (MPI_Fint)MPI_Type_commit(&datatype);
    *f_handle = MPI_Type_c2f(datatype);
    return;
}
```

The same approach can be used for all other MPI functions. The call to MPI\_XXX\_f2c (resp. MPI\_XXX\_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

*Rationale.* The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.

Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple (e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (*End of rationale.*)

## 19.3.5 Status

The following two procedures are provided in C to convert from a Fortran (with the mpi module or mpif.h) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

int MPI\_Status\_f2c(const MPI\_Fint \*f\_status, MPI\_Status \*c\_status)

 $\mathbf{2}$ 

3

4

5 6

7

9

10

11

12 13

14 15

16

17

18 19

20

21

22

23

24 25

26

27

28 29

30

31

32

33

34

35

36

37

38

39 40 41

42

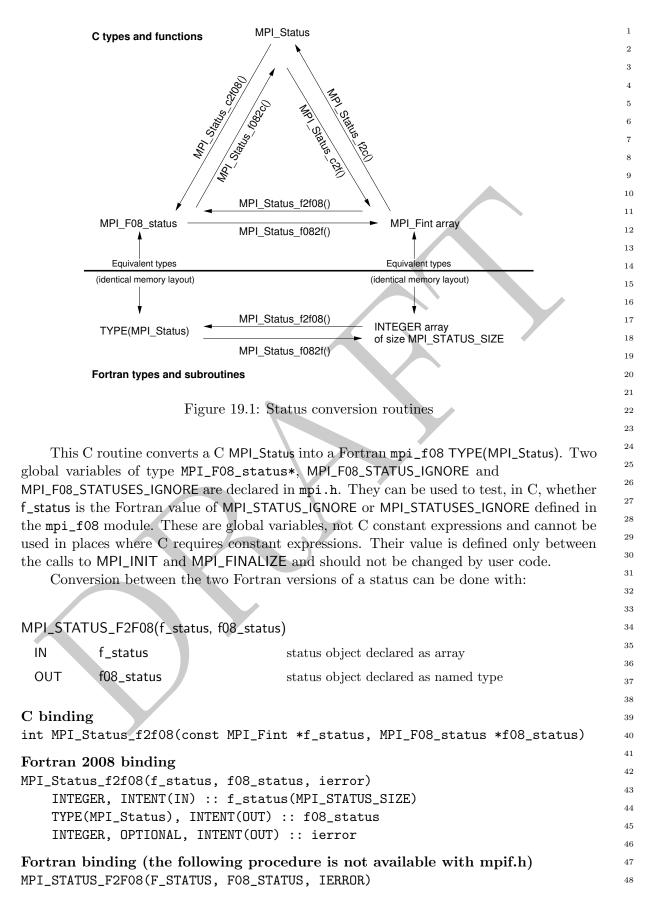
43

44

45

46 47

1 If f\_status is a valid Fortran status, but not the Fortran value of MPI\_STATUS\_IGNORE  $\mathbf{2}$ or MPI\_STATUSES\_IGNORE, then MPI\_Status\_f2c returns in c\_status a valid C status with 3 the same content. If f\_status is the Fortran value of MPI\_STATUS\_IGNORE or 4 MPI\_STATUSES\_IGNORE, or if f\_status is not a valid Fortran status, then the call is erroneous.  $\mathbf{5}$ In C, such an f\_status array can be defined with MPI\_Fint f\_status[ 6 MPI\_F\_STATUS\_SIZE]. Within this array, one can use in C the indexes MPI\_F\_SOURCE,  $\overline{7}$ MPI\_F\_TAG, and MPI\_F\_ERROR, to access the same elements as in Fortran with MPI\_SOURCE, 8 MPI\_TAG and MPI\_ERROR. The C indexes are 1 less than the corresponding indexes in 9 Fortran due to the different default array start indexes in both languages. 10 The C status has the same source, tag and error code values as the Fortran status, 11and returns the same answers when queried for count, elements, and cancellation. The 12conversion function may be called with a Fortran status argument that has an undefined 13error field, in which case the value of the error field in the C status argument is undefined. 14Two global variables of type MPI\_Fint\*, MPI\_F\_STATUS\_IGNORE and 15MPI\_F\_STATUSES\_IGNORE are declared in mpi.h. They can be used to test, in C, whether 16f\_status is the Fortran value of MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE defined in 17 the mpi module or mpif.h. These are global variables, not C constant expressions and 18 cannot be used in places where C requires constant expressions. Their value is defined only 19between the calls to MPI\_INIT and MPI\_FINALIZE and should not be changed by user code. 20To do the conversion in the other direction, we have the following: 21int MPI\_Status\_c2f(const MPI\_Status \*c\_status, MPI\_Fint \*f\_status) 22 23This call converts a C status into a Fortran status, and has a behavior similar to 24MPI\_Status\_f2c. That is, the value of c\_status must not be either MPI\_STATUS\_IGNORE or 25MPI\_STATUSES\_IGNORE. 26Advice to users. There exists no separate conversion function for arrays of statuses, 27since one can simply loop through the array, converting each status with the routines 28in Figure 19.1. (End of advice to users.) 2930 *Rationale.* The handling of MPI\_STATUS\_IGNORE is required in order to layer libraries 31with only a C wrapper: if the Fortran call has passed MPI\_STATUS\_IGNORE, then the 32 C wrapper must handle this correctly. Note that this constant need not have the 33 same value in Fortran and C. If MPI\_Status\_f2c were to handle MPI\_STATUS\_IGNORE, 34 then the type of its result would have to be MPI\_Status\*\*, which was considered an 35inferior solution. (End of rationale.) 36 37 Using the mpi\_f08 Fortran module, a status is declared as TYPE(MPI\_Status). The C 38 type MPI\_F08\_status can be used to pass a Fortran TYPE(MPI\_Status) argument into a C 39 routine. Figure 19.1 illustrates all status conversion routines. Some are only available in 40 C, some in both C and the Fortran mpi and mpi\_f08 interfaces (but not in the mpif.h 41 interface). 4243 int MPI\_Status\_f082c(const MPI\_F08\_status \*f08\_status, 44MPI\_Status \*c\_status) 45This C routine converts a Fortran mpi\_f08 TYPE(MPI\_Status) into a C MPI\_Status. 4647int MPI\_Status\_c2f08(const MPI\_Status \*c\_status, 48 MPI\_F08\_status \*f08\_status)



```
1
          INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
\mathbf{2}
          TYPE(MPI_Status) :: F08_STATUS
3
          This routine converts a Fortran INTEGER, DIMENSION(MPI_STATUS_SIZE) status array
4
     into a Fortran mpi_f08 TYPE(MPI_Status).
5
6
\overline{7}
     MPI_STATUS_F082F(f08_status, f_status)
8
       IN
                 f08_status
                                              status object declared as named type
9
       OUT
10
                 f_status
                                              status object declared as array
11
12
     C binding
13
     int MPI_Status_f082f(const MPI_F08_status *f08_status, MPI_Fint *f_status)
14
     Fortran 2008 binding
15
     MPI_Status_f082f(f08_status, f_status, ierror)
16
          TYPE(MPI_Status), INTENT(IN) :: f08_status
17
          INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
18
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     Fortran binding (the following procedure is not available with mpif.h)
21
     MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
22
          TYPE(MPI_Status) :: F08_STATUS
23
          INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
^{24}
          This routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER,
25
     DIMENSION (MPI_STATUS_SIZE) status array.
26
27
     19.3.6
             MPI Opaque Objects
28
29
     Unless said otherwise, opaque objects are "the same" in all languages: they carry the same
30
     information, and have the same meaning in both languages. The mechanism described
^{31}
     in the previous section can be used to pass references to MPI objects from language to
32
     language. An object created in one language can be accessed, modified or freed in another
33
     language.
34
       We examine below in more detail issues that arise for each type of MPI object.
35
36
     Datatypes
37
38
     Datatypes encode the same information in all languages. E.g., a datatype accessor like
39
     MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype
40
     defined in one language is used for a communication call in another language, then the
41
     message sent will be identical to the message that would be sent from the first language:
42
     the same communication buffer is accessed, and the same representation conversion is per-
43
     formed, if needed. All predefined datatypes can be used in datatype constructors in any
44
     language. If a datatype is committed, it can be used for communication in any language.
45
          The function MPI_GET_ADDRESS returns the same value in all languages. Note that
46
     we do not require that the constant MPI_BOTTOM have the same value in all languages (see
47
     Section 19.3.9).
48
```

```
Example 19.12
                                                                                     2
! FORTRAN CODE
REAL :: R(5)
INTEGER :: TYPE, IERR, AOBLEN(1), AOTYPE(1)
                                                                                     5
INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
                                                                                     6
! create an absolute datatype for array R
AOBLEN(1) = 5
                                                                                     9
CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
                                                                                     10
AOTYPE(1) = MPI_REAL
                                                                                     11
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
                                                                                     12
CALL C_ROUTINE(TYPE)
                                                                                     13
                                                                                     14
/* C code */
                                                                                     15
                                                                                     16
void C_ROUTINE(MPI_Fint *ftype)
                                                                                     17
Ł
                                                                                     18
   int count = 5;
                                                                                     19
   int lens[2] = {1,1};
                                                                                     20
   MPI_Aint displs[2];
                                                                                     21
   MPI_Datatype types[2], newtype;
                                                                                     22
                                                                                     23
   /* create an absolute datatype for buffer that consists
                                                                  */
                                                                                     24
   /* of count, followed by R(5)
                                                                  */
                                                                                     25
                                                                                     26
   MPI_Get_address(&count, &displs[0]);
                                                                                     27
   displs[1] = 0;
                                                                                     28
   types[0] = MPI_INT;
                                                                                     29
   types[1] = MPI_Type_f2c(*ftype);
                                                                                     30
   MPI_Type_create_struct(2, lens, displs, types, &newtype);
                                                                                     31
   MPI_Type_commit(&newtype);
                                                                                     32
                                                                                     33
   MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
                                                                                     34
  /* the message sent contains an int count of 5, followed
                                                                 */
                                                                                     35
   /* by the 5 REAL entries of the Fortran array R.
                                                                 */
                                                                                     36
}
                                                                                     37
```

38 Advice to implementors. The following implementation can be used: MPI addresses, 39 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 40 41 is stored in the datatype, when datatypes with absolute addresses are constructed. 42When 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 4344address 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 4546a call with a regular buffer argument: in both cases the base address is **buf**. On the 47other hand, if MPI\_BOTTOM is not zero, then the implementation has to be slightly 48 different. A test is performed to check whether  $buf = MPI_BOTTOM$ . If true, then the

base address is zero, otherwise it is buf. In particular, if MPI\_BOTTOM does not have the same value in Fortran and C, then an additional test for  $buf = MPI_BOTTOM$  is needed in at least one of the languages.

It may be desirable to use a value other than zero for MPI\_BOTTOM even in C, so as to distinguish it from a NULL pointer. If MPI\_BOTTOM = c then one can still avoid the test buf = MPI\_BOTTOM, by using the displacement from MPI\_BOTTOM, i.e., the regular address - c, as the MPI address returned by MPI\_GET\_ADDRESS and stored in absolute datatypes. (*End of advice to implementors.*)

<sup>10</sup> 11 Callback Functions

MPI calls may associate callback functions with MPI objects: error handlers are associated
 with communicators, files, windows, and sessions; attribute copy and delete functions are
 associated with attribute keys; reduce operations are associated with operation objects, etc.
 In a multilanguage environment, a function passed in an MPI call in one language may be
 invoked by an MPI call in another language. MPI implementations must make sure that
 such invocation will use the calling convention of the language the function is bound to.

- Advice to implementors. Callback functions need to have a language tag. This tag is set when the callback function is passed in by the library function (which is presumably different for each language and language support method), and is used to generate the right calling sequence when the callback function is invoked. (*End of advice to implementors.*)
  - Advice to users. If a subroutine written in one language or Fortran support method wants to pass a callback routine including the predefined Fortran functions (e.g., MPI\_COMM\_NULL\_COPY\_FN) to another application routine written in another language or Fortran support method, then it must be guaranteed that both routines use the callback interface definition that is defined for the argument when passing the callback to an MPI routine (e.g., MPI\_COMM\_CREATE\_KEYVAL); see also the advice to users on page 363. (*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.*)

<sup>38</sup> <sub>39</sub> Reduce Operations

<sup>40</sup> All predefined named and unnamed datatypes as listed in Section 6.9.2 can be used in the
 <sup>41</sup> listed predefined operations independent of the programming language from which the MPI
 <sup>42</sup> 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.)

 $\mathbf{2}$ 

#### 19.3.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI\_TAG\_UB, MPI\_WTIME\_IS\_GLOBAL, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI\_XXX\_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 7.7 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 7.7. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integervalued 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 deprecated attribute functions are used. In C, the deprecated routines MPI\_Attr\_put and MPI\_Attr\_get behave identical to MPI\_Comm\_set\_attr and MPI\_Comm\_get\_attr.

#### Example 19.13

A. Setting an attribute value in C

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

```
1
     int set_val = 3;
\mathbf{2}
     struct foo set_struct;
3
4
     /* Set a value that is a pointer to an int */
5
6
     MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val);
\overline{7}
     /* Set a value that is a pointer to a struct */
8
     MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct);
9
     /* Set an integer value */
10
     MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17);
11
         B. Reading the attribute value in C
12
13
     int flag, *get_val;
14
     struct foo *get_struct;
15
16
     /* Upon successful return, get_val == &set_val
17
        (and therefore *get_val == 3) */
18
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag);
19
     /* Upon successful return, get_struct == &set_struct */
20
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag);
21
     /* Upon successful return, get_val == (void*) 17 */
22
     /*
                i.e., (MPI_Aint) get_val == 17 */
23
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag);
24
25
         C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
26
27
     LOGICAL FLAG
28
     INTEGER IERR, GET_VAL, GET_STRUCT
29
30
     ! Upon successful return, GET_VAL == &set_val, possibly truncated
^{31}
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
32
     ! Upon successful return, GET_STRUCT == &set_struct, possibly truncated
33
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
34
     ! Upon successful return, GET_VAL == 17
35
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
36
37
         D. Reading the attribute value with Fortran MPI-2 calls
38
39
     LOGICAL FLAG
40
     INTEGER IERR
41
     INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
42
43
     ! Upon successful return, GET_VAL == &set_val
44
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
45
     ! Upon successful return, GET_STRUCT == &set_struct
46
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
47
     ! Upon successful return, GET_VAL == 17
48
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
```

**Example 19.14** A. Setting an attribute value with the (deprecated) Fortran MPI-1 call 1  $\mathbf{2}$ INTEGER IERR, VAL 3 VAL = 74 CALL MPI\_ATTR\_PUT(MPI\_COMM\_WORLD, KEYVAL, VAL, IERR) 56 B. Reading the attribute value in C 7 int flag; 9 int \*value; 10 11 /\* Upon successful return, value points to internal MPI storage and 12\*value == (int) 7 \*/ 13 MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval, &value, &flag); 1415C. Reading the attribute value with (deprecated) Fortran MPI-1 calls 1617LOGICAL FLAG 18 INTEGER IERR, VALUE 19 20! Upon successful return, VALUE == 7 21CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL, VALUE, FLAG, IERR) 22 23D. Reading the attribute value with Fortran MPI-2 calls  $^{24}$ 25LOGICAL FLAG 26INTEGER IERR 27INTEGER (KIND=MPI\_ADDRESS\_KIND) VALUE 2829! Upon successful return, VALUE == 7 (sign extended) 30 CALL MPI\_COMM\_GET\_ATTR(MPI\_COMM\_WORLD, KEYVAL, VALUE, FLAG, IERR) 3132 **Example 19.15** A. Setting an attribute value via a Fortran MPI-2 call 33 34 INTEGER IERR 35INTEGER(KIND=MPI\_ADDRESS\_KIND) VALUE1 36 INTEGER(KIND=MPI\_ADDRESS\_KIND) VALUE2 37 VALUE1 = 4238 VALUE2 = INT(2, KIND=MPI\_ADDRESS\_KIND) \*\* 40 39 40 CALL MPI\_COMM\_SET\_ATTR(MPI\_COMM\_WORLD, KEYVAL1, VALUE1, IERR) 41 CALL MPI\_COMM\_SET\_ATTR(MPI\_COMM\_WORLD, KEYVAL2, VALUE2, IERR) 4243 B. Reading the attribute value in C 44454647

```
1
     int flag;
\mathbf{2}
     MPI_Aint *value1, *value2;
3
4
     /* Upon successful return, value1 points to internal MPI storage and
5
         *value1 == 42 */
6
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
7
     /* Upon successful return, value2 points to internal MPI storage and
8
         *value2 == 2^40 */
9
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
10
         C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
11
12
     LOGICAL FLAG
13
     INTEGER IERR, VALUE1, VALUE2
14
15
     ! Upon successful return, VALUE1 == 42
16
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
17
     ! Upon successful return, VALUE2 == 2<sup>40</sup>, or 0 if truncation
18
     ! needed (i.e., the least significant part of the attribute word)
19
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
20
         D. Reading the attribute value with Fortran MPI-2 calls
21
22
     LOGICAL FLAG
23
     INTEGER IERR
24
     INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2
25
26
     ! Upon successful return, VALUE1 == 42
27
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
28
     ! Upon successful return, VALUE2 == 2<sup>40</sup>
29
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
30
31
         The predefined MPI attributes can be integer valued or address-valued. Predefined
32
     integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a call to
33
     the deprecated Fortran routine MPI_ATTR_PUT, i.e., in Fortran,
34
     MPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_TAG_UB, val, flag, ierr) will return
35
     in val the upper bound for tag value; in C, MPI_Comm_get_attr(MPI_COMM_WORLD,
36
     MPI_TAG_UB, &p, &flag) will return in p a pointer to an int containing the upper bound
37
     for tag value.
38
         Address-valued predefined attributes, such as MPI_WIN_BASE behave as if they were
39
     put by a C call, i.e., in Fortran, MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag,
40
     ierror) will return in val the base address of the window, converted to an integer. In C,
41
     MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window
42
     base, cast to (void *).
43
                       The design is consistent with the behavior specified for predefined at-
44
          Rationale.
45
          tributes, and ensures that no information is lost when attributes are passed from
46
          language to language. Because the language interoperability for predefined attributes
47
          was defined based on MPI_ATTR_PUT, this definition is kept for compatibility reasons
48
          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.)

## 19.3.8 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.

#### 19.3.9 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 19.3.4. 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];

(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.

The current MPI standard specifies that MPI\_BOTTOM can be used in Rationale. 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 19.3.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (End of rationale.)

#### Unofficial Draft for Comment Only

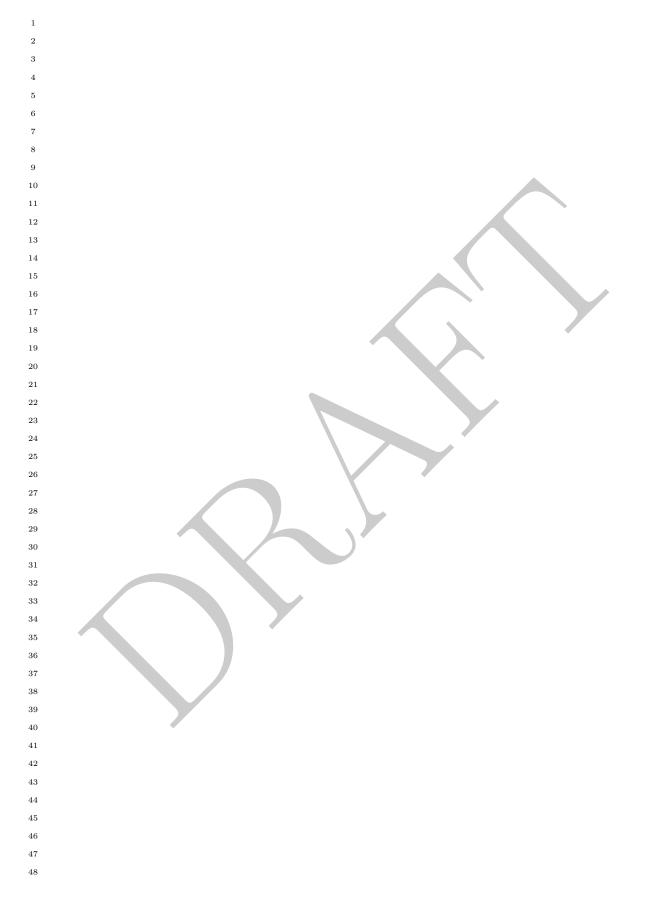
 $\mathbf{2}$ 

 $^{24}$ 

```
1
     19.3.10 Interlanguage Communication
\mathbf{2}
     The type matching rules for communication in MPI are not changed: the datatype specifi-
3
     cation for each item sent should match, in type signature, the datatype specification used to
4
     receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item
5
     should match the type declaration for the corresponding communication buffer location,
6
     unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it
7
     complies with these rules.
8
9
     Example 19.16 In the example below, a Fortran array is sent from Fortran and received
10
     in C.
11
     ! FORTRAN CODE
12
     SUBROUTINE MYEXAMPLE()
13
14
     USE mpi_f08
     REAL :: R(5)
15
16
     INTEGER :: IERR, MYRANK, AOBLEN(1)
17
     TYPE(MPI_Datatype) :: TYPE, AOTYPE(1)
     INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
18
19
     ! create an absolute datatype for array R
20
21
     AOBLEN(1) = 5
     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)
^{24}
     CALL MPI_TYPE_COMMIT(TYPE, IERR)
25
26
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
27
     IF (MYRANK.EQ.O) THEN
28
        CALL MPI_SEND(MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
29
     ELSE
30
        CALL C_ROUTINE(TYPE%MPI_VAL)
^{31}
     END IF
32
     END SUBROUTINE
33
34
35
     /* C code */
36
37
     void C_ROUTINE(MPI_Fint *fhandle)
38
     {
39
        MPI_Datatype type;
40
        MPI_Status status;
41
42
        type = MPI_Type_f2c(*fhandle);
43
44
        MPI_Recv(MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
45
     }
46
47
          MPI implementors may weaken these type matching rules, and allow messages to be sent
```

<sup>47</sup> MPI implementors may weaken these type matching rules, and allow messages to be sent <sup>48</sup> 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.

 $\frac{1}{2}$ 



	#	all changes	
	#-nnn	by issue nnn	
	#-PRnnr	n by pull request	
	Non prol	blematic fomatting-changes and typo-corrections are not listed	
	Embigge	ening: only new _c versions are marked, all other is not listed	1
	New or o	changed content	2
	Problem	atic fomatting-changes are listed	3
#-error	Errors in	RC 40 - also marked with #-error	4
	B.m.n: i.	- refers to Change-Log section B.m.n, Item i	5
Annex A			6
			7
			8

# 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.

I I I I I I I I I I I I I I I I I I I		24
	Error classes	25
-	C type: const int (or unnamed enum)	26
	Fortran type: INTEGER	27
-	MPI_SUCCESS	28
	MPI_ERR_BUFFER	29
	MPI_ERR_COUNT	30
	MPI_ERR_TYPE	31
	MPI_ERR_TAG	32
	MPI_ERR_COMM	33
	MPI_ERR_RANK	34
	MPI_ERR_REQUEST	35
	MPI_ERR_ROOT	36
	MPI_ERR_GROUP	37
	MPI_ERR_OP	38
	MPI_ERR_TOPOLOGY	39
	MPI_ERR_DIMS	40
	MPI_ERR_ARG	41
	MPI_ERR_UNKNOWN	42
	MPI_ERR_TRUNCATE	43
	MPI_ERR_OTHER	44
	MPI_ERR_INTERN	45
-		46
	(Continued on next page)	47
		48

	840	ANNEA A. LANGUAGE
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		MPI_ERR_NAME
24		MPI_ERR_NO_MEM
25		MPI_ERR_NOT_SAME
26		MPI_ERR_NO_SPACE
27		MPI_ERR_NO_SUCH_FILE
28		MPI_ERR_PORT
29		MPI_ERR_PROC_ABORTED
30		MPI_ERR_QUOTA
31		MPI_ERR_READ_ONLY
32		MPI_ERR_RMA_ATTACH
33		MPI_ERR_RMA_CONFLICT
34		
		MPI_ERR_RMA_RANGE
35		MPI_ERR_RMA_SHARED
36		MPI_ERR_RMA_SYNC
37		MPI_ERR_RMA_FLAVOR
38		MPI_ERR_SERVICE
39		MPI_ERR_SIZE
40		MPI_ERR_SPAWN
41		MPI_ERR_UNSUPPORTED_DATAREP
42		MPI_ERR_UNSUPPORTED_OPERATION
43		MPI_ERR_WIN
44		(Continued on next page)
45		
46		
47		

ANNEX A. LANGUAGE BINDINGS SUMMARY

#-???

 $^{48}$ 

	Error classes (continued)	1
	C type: const int (or unnamed enum)	2
	Fortran type: INTEGER	3
	MPI_T_ERR_CANNOT_INIT	4
	MPI_T_ERR_NOT_ACCESSIBLE	5
#-???	MPI_T_ERR_NOT_INITIALIZED	6
•	MPI_T_ERR_NOT_SUPPORTED	7
	MPI_T_ERR_MEMORY	8
	MPI_T_ERR_INVALID	9
	MPI_T_ERR_INVALID_INDEX	10
	MPI_T_ERR_INVALID_ITEM	11
	MPI_T_ERR_INVALID_SESSION	12
	MPI_T_ERR_INVALID_HANDLE	13
	MPI_T_ERR_INVALID_NAME	14
	MPI_T_ERR_OUT_OF_HANDLES	15
	MPI_T_ERR_OUT_OF_SESSIONS	16
	MPI_T_ERR_CVAR_SET_NOT_NOW	17
	MPI_T_ERR_CVAR_SET_NEVER	18
	MPI_T_ERR_PVAR_NO_WRITE	19
	MPI_T_ERR_PVAR_NO_STARTSTOP	20
	MPI_T_ERR_PVAR_NO_ATOMIC	21
	MPI_ERR_LASTCODE	22
	Buffer Address Constants	23
C tvr	pe: void * const	24 25
01	can type: (predefined memory location) <sup><math>1</math></sup>	26
	BOTTOM	20
MPI_	IN_PLACE	28
$^{-1}$ No	te that in Fortran these constants are not usable for initialization	29
exp	pressions or assignment. See Section 2.5.4.	30
	Assorted Constants	31
		32
	C type: const int (or unnamed enum) Fortran type: INTEGER	33
	MPI_PROC_NULL	34
	MPI_PROC_NULL MPI_ANY_SOURCE	35
	MPI_ANY_TAG	36
	MPI_UNDEFINED	37
	MPI_BSEND_OVERHEAD	38
	MPI_KEYVAL_INVALID	39
	MPI_LOCK_EXCLUSIVE	40
	MPI_LOCK_SHARED	41
	MPI_ROOT	42
		43
		44
	No Process Message Handle	45
	C type: $MPI_Message$	46
	Fortran type: INTEGER or TYPE(MPI_Message)	47
	MPI_MESSAGE_NO_PROC	48

	848 ANNEX A. LANGUAGE BINDINGS SUMMARY
1	Fortran Support Method Specific Constants
2	Fortran type: LOGICAL
3	MPI_SUBARRAYS_SUPPORTED (Fortran only)
4	MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
5	
<b>#-???</b> 6	Status array size and reserved index values (Fortran only)
7	Fortran type: INTEGER
8	MPI_STATUS_SIZE
9	MPI_SOURCE
10	MPI_TAG
11	MPI_ERROR
12	
13 # <b>777</b>	Eastern data and a second index (Carla)
<b>#-???</b> 14	Fortran status array size and reserved index values (C only)
15	C type: int
16	MPI_F_STATUS_SIZE
17	MPI_F_SOURCE
18	MPI_F_TAG
19	MPI_F_ERROR
20	
21	Variable Address Size (Fortran only)
22	Fortran type: INTEGER
23	MPI_ADDRESS_KIND
24	MPI_COUNT_KIND
25	MPI_INTEGER_KIND
26	MPI_OFFSET_KIND
27	
28	Error-handling specifiers
29	C type: MPI_Errhandler
30	Fortran type: INTEGER or TYPE(MPI_Errhandler)
31	MPI_ERRORS_ARE_FATAL
32	MPI_ERRORS_RETURN
33	
34	Maximum Sizes for Strings
35	C type: const int (or unnamed enum)
36	Fortran type: INTEGER
37	MPI_MAX_DATAREP_STRING
38	MPI_MAX_ERROR_STRING
39	MPI_MAX_INFO_KEY
40	MPI_MAX_INFO_VAL
41	MPI_MAX_LIBRARY_VERSION_STRING
42	MPI_MAX_OBJECT_NAME
43	MPI_MAX_PORT_NAME
<b>#-???</b> 44	MPI_MAX_PROCESSOR_NAME Yet unclear, whether there will be
#-error!	MPI_MAX_FROM_GROUP_TAGMPI_MAX_FROM_GROUP_TAG or
#-382	MPI_MAX_PSET_NAME_LEN MPI_MAX_FROM_GROUP_STRINGTAG or none of them see Issue #-382.
47	Or hole of them see issue #-582. Decision in PR 400:
#-PR400 48	MPI_MAX_STRINGTAG_LEN
#-update4	
#-TODO Merge of PR400	Unofficial Draft for Comment Only

Named Predefined Datatypes	C types
$\mathrm{C}\ \mathrm{type}$ : MPI_Datatype	
Fortran type: INTEGER	
or TYPE(MPI_Datatype)	
MPI_CHAR	char
	(treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long
MPI_LONG_LONG_INT	signed long long
$MPI_LONG_LONG$ (as a synonym)	signed long long
MPI_SIGNED_CHAR	signed char
	(treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char
	(treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
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)

1	Named Predefined Datatypes	Fortran types
2	C type: MPI_Datatype	v 1
3	Fortran type: INTEGER	
4	or TYPE(MPI_Datatype)	
5	MPI_INTEGER	INTEGER
6	MPI_REAL	REAL
7	MPI_DOUBLE_PRECISION	DOUBLE PRECISION
8	MPI_COMPLEX	COMPLEX
9	MPI_LOGICAL	LOGICAL
10	MPI_CHARACTER	CHARACTER(1)
11	MPI_AINT	INTEGER (KIND=MPI_ADDRESS_KIND)
12	MPI_COUNT	INTEGER (KIND=MPI_COUNT_KIND)
13	MPI_OFFSET	INTEGER (KIND=MPI_OFFSET_KIND)
14	MPI_BYTE	(any Fortran type)
15	MPI_PACKED	(any Fortran type)
16		
17	Named Predefined Datatype	$s^1 \mid C++ types$
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_COMPL	EX std::complex <long double=""></long>
25	$^{-1}$ If an accompanying C++ comp	oiler is missing, then the
26	MPI datatypes in this table are	not defined.
27		
28	Optional datatypes (F	ortran) Fortran types
29	C 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		1
47		

	Datatypes for reduction functions (C)	1
	C type: MPI_Datatype	2
	Fortran type: INTEGER or TYPE(MPI_Datatype)	3
	MPI_FLOAT_INT	4
	MPI_DOUBLE_INT	5
	MPI_LONG_INT	6
	MPI_2INT	7
	MPI_SHORT_INT	8
	MPI_LONG_DOUBLE_INT	9
	Datatypes for reduction functions (Fortran)	10
		11
	C type: MPI_Datatype	12
	Fortran type: INTEGER or TYPE(MPI_Datatype)	13
	MPI_2REAL	14
	MPI_2DOUBLE_PRECISION	15
	MPI_2INTEGER	16
	Reserved communicators	17
	C type: MPI_Comm	18
	Fortran type: INTEGER or TYPE(MPI_Comm)	19
	MPI_COMM_WORLD	20
	MPI_COMM_SELF	21
		22
	Communicator split type constants	23
	C type: const int (or unnamed enum)	24
	Fortran type: INTEGER	25
	MPI_COMM_TYPE_SHARED	26
	MPI_COMM_TYPE_HW_UNGUIDED	27
l	MPI_COMM_TYPE_HW_GUIDED	28
	Results of communicator and group comparisons	29 30
	C type: const int (or unnamed enum)	31
	Fortran type: INTEGER	32
	MPI_IDENT	33
	MPI_CONGRUENT	34
	MPI_SIMILAR	35
	MPI_UNEQUAL	36
	Environmental inquiry info key	37
	Environmental inquiry info key C type: MPI_Info	38
	Fortran type: INTEGER or TYPE(MPI_Info)	39
	MPI_INFO_ENV	40
		41
	Environmental inquiry keys	42
	C type: const int (or unnamed enum)	43
	Fortran type: INTEGER	44
	MPI_TAG_UB	45
	MPI_IO	46
	MPI_HOST	47
	MPI_WTIME_IS_GLOBAL	48

#-???

:	Collective Operations
:	C type: MPI_Op
:	Fortran type: INTEGER or TYPE(MPI_Op)
4	MPI_MAX
Į	MPI_MIN
(	MPI_SUM
,	MPI_PROD
٤	MPI_MAXLOC
9	MPI_MINLOC
1	MPI_BAND
1	1 MPI_BOR
1	<sup>2</sup> MPI_BXOR
1	<sup>3</sup> MPI_LAND
1	4 MPI_LOR
1	5 MPI_LXOR
1	6 MPI_REPLACE
1	7 MPI_NO_OP
1	8
1	9 Null Handles
2	<sup>0</sup> C/Fortran name
2	<sup>1</sup> C type / Fortran type
2	2 MPI_GROUP_NULL
2	<sup>3</sup> MPI_Group / INTEGER or TYPE(MPI_Group)
2	4 MPI_COMM_NULL
2	<sup>5</sup> MPI_Comm / INTEGER or TYPE(MPI_Comm)
2	
2	in <u>Loadatype</u> in <u>Loadatype</u>
2	8 MPI_REQUEST_NULL
2	Win _ Kequest / TATEGER OF THE (Win _ Kequest)
3	
3	WITCOP / INTEGER OF FITE(WITCOP)
3	MIT_ERRITANDEER_NOEE
3	<sup>3</sup> MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)
3	WIT_TIEL_NOEL
3	with the principal of the terms of terms
3	
3	
#-error! <sup>3</sup>	WIT_SESSION_NOEL
<b>#-???</b> 3	MPI_Session / INTEGER or TYPE(MPI_Session)
Done <sup>4</sup>	
in 4 PR444 4	<sup>1</sup> MPI_Win / INTEGER or TYPE(MPI_Win)
FK444 4	<sup>2</sup> MPI_MESSAGE_NULL
4	win _wessage / in Louit of the L(win _wessage)
4	
4	
4	C type. Wi 1_Gloup
4	Pottan type. INTEGER of PTT E(MIT_Gloup)
4	<sup>8</sup> MPI_GROUP_EMPTY

Topologies	1
C type: const int (or unnamed enum)	2
Fortran type: INTEGER	3
MPI_GRAPH	4
MPI_CART	5
MPI_DIST_GRAPH	6
Predefined functions	7 8
C/Fortran name	9
C type	10
/ Fortran type with mpi module / Fortran type with mpi_f08 module	11
MPI_COMM_NULL_COPY_FN All old-style Fortran callback prototypes also	12
MPI_Comm_copy_attr_function referenced in the callback prototype index	<sup>13</sup>
/ COMM_COPY_ATTR_FUNCTION / PROCEDURE(MPI_Comm_copy_attr_function) <sup>1</sup> )	14 #-error!
MPI_COMM_DUP_FN	15 Done
MPI_Comm_copy_attr_function	<sup>16</sup> IN PR449
/ COMM_COPY_ATTR_FUNCTION / PROCEDURE(MPI_Comm_copy_attr_function) <sup>1</sup> )	17
MPI_COMM_NULL_DELETE_FN	18
MPI_Comm_delete_attr_function	19
$/ \text{ COMM_DELETE_ATTR_FUNCTION } / \text{PROCEDURE(MPI_Comm_delete_attr_function)}^1)$	20
MPI_WIN_NULL_COPY_FN	21
MPI_Win_copy_attr_function	22
$/$ WIN_COPY_ATTR_FUNCTION $/$ PROCEDURE(MPI_Win_copy_attr_function) $^{1}$ )	23
MPI_WIN_DUP_FN	24
MPI_Win_copy_attr_function	25
$/$ WIN_COPY_ATTR_FUNCTION $/$ PROCEDURE(MPI_Win_copy_attr_function) $^1)$	26
MPI_WIN_NULL_DELETE_FN	27
MPI_Win_delete_attr_function	28
/ WIN_DELETE_ATTR_FUNCTION / PROCEDURE(MPI_Win_delete_attr_function) <sup>1</sup> )	29 30
MPI_TYPE_NULL_COPY_FN	31
MPI_Type_copy_attr_function	32
/ TYPE_COPY_ATTR_FUNCTION / PROCEDURE(MPI_Type_copy_attr_function) <sup>1</sup> )	33
MPI_TYPE_DUP_FN	34
MPI_Type_copy_attr_function / TYPE_COPY_ATTR_EUNCTION / DROCEDURE(MRL_Type_copy_attr_function) <sup>1</sup>	35
/ TYPE_COPY_ATTR_FUNCTION / PROCEDURE(MPI_Type_copy_attr_function) <sup>1</sup> )	36
MPI_TYPE_NULL_DELETE_FN MPI_Type_delete_attr_function	37
<pre>MP1_Type_delete_attr_function / TYPE_DELETE_ATTR_FUNCTION / PROCEDURE(MP1_Type_delete_attr_function) 1)</pre>	38
MPI_CONVERSION_FN_NULL Many out-commented lines starting with	39 #-error!
MDT Determine function for stice	40 Done
/ DATAREP_CONVERSION_FUNCTION / PROCEDURE(MPI_Datarep_conversion_function) <sup>1</sup> )	41 in
<sup>1</sup> See the advice to implementors (on page 363) and advice to users (on page 363)	42 PR444
on the predefined Fortran functions MPI_COMM_NULL_COPY_FN, in	43
Section 7.7.2.	44
	45
	46
	47

 $^{48}$ 

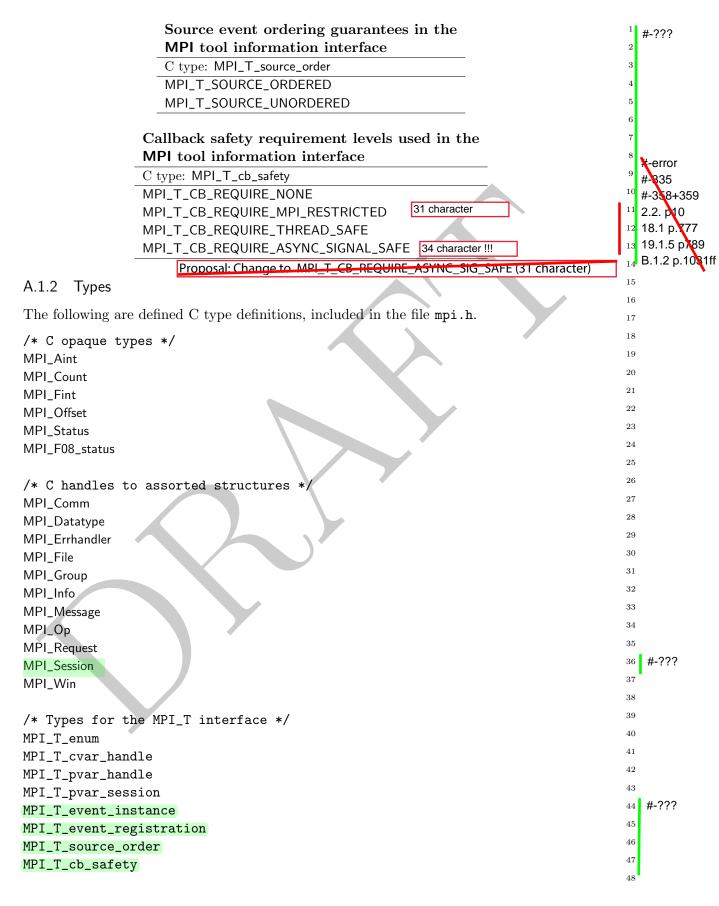
1	Deprecated predefined functions
2	C/Fortran name
3	All old-style Fortran caliback
<b>4</b>	
#-error! 5	MPI_NULL_COPY_FN the callback prototype index MPI_Copy_function / COPY_FUNCTION
Done 6	MPI_DUP_FN
in 7	
PR449 8	MPI_Copy_function / COPY_FUNCTION
	MPI_NULL_DELETE_FN
9	MPI_Delete_function / DELETE_FUNCTION
10 11	Predefined Attribute Keys
12	C type: const int (or unnamed enum)
13	Fortran type: INTEGER
14	MPI_APPNUM
15	
16	MPI_LASTUSEDCODE
17	MPI_UNIVERSE_SIZE
	MPI_WIN_BASE
18	MPI_WIN_DISP_UNIT
19	MPI_WIN_SIZE
20	MPI_WIN_CREATE_FLAVOR
21	MPI_WIN_MODEL
22	
23	MPI Window Create Flavors
24	C type: const int (or unnamed enum)
25	Fortran type: INTEGER
26	MPI_WIN_FLAVOR_CREATE
27	MPI_WIN_FLAVOR_ALLOCATE
28	MPI_WIN_FLAVOR_DYNAMIC
29	MPI_WIN_FLAVOR_SHARED
30	
31	MPI Window Models
32	C type: const int (or unnamed enum)
33	Fortran type: INTEGER
34	MPI_WIN_SEPARATE
35	MPI_WIN_UNIFIED
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

	Mode Constants
_	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_MODE_APPEND
	MPI_MODE_CREATE
	MPI_MODE_DELETE_ON_CLOSE
	MPI_MODE_EXCL
	MPI_MODE_NOCHECK
	MPI_MODE_NOPRECEDE
	MPI_MODE_NOPUT
	MPI_MODE_NOSTORE
	MPI_MODE_NOSUCCEED
	MPI_MODE_RDONLY
	MPI_MODE_RDWR
	MPI_MODE_SEQUENTIAL
	MPI_MODE_UNIQUE_OPEN
	MPI_MODE_WRONLY
	Datatype Decoding Constants
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_COMBINER_CONTIGUOUS
	MPI_COMBINER_DARRAY
	MPI_COMBINER_DUP
	MPI_COMBINER_F90_COMPLEX
	MPI_COMBINER_F90_INTEGER
	MPI_COMBINER_F90_REAL
	MPI_COMBINER_HINDEXED
	MPI_COMBINER_HVECTOR
	MPI_COMBINER_INDEXED_BLOCK
	MPI_COMBINER_HINDEXED_BLOCK
	MPI_COMBINER_INDEXED
	MPI_COMBINER_NAMED
	MPI_COMBINER_RESIZED
	MPI_COMBINER_STRUCT
	MPI_COMBINER_SUBARRAY
	MPI_COMBINER_VECTOR
	Threads Constants
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_THREAD_FUNNELED
	MPI_THREAD_MULTIPLE
	MPI_THREAD_SERIALIZED
	MPI_THREAD_SINGLE

	1	File Operation Constants, Part 1	_
	2	C type: const MPI_Offset (or unnamed enum)	
	3	Fortran type: INTEGER (KIND=MPI_OFFSET_KIND)	_
	4	MPI_DISPLACEMENT_CURRENT	_
	5	Ello Constituto De La	
	6	File Operation Constants, Part 2	
	7 °	C type: const int (or unnamed enum)	
	8	Fortran type: INTEGER	
	9 10	MPI_DISTRIBUTE_BLOCK	
	10	MPI_DISTRIBUTE_CYCLIC	$\frown$
	12	MPI_DISTRIBUTE_DFLT_DARG	
	13	MPI_DISTRIBUTE_NONE	
	14	MPI_ORDER_C MPI_ORDER_FORTRAN	
	15	MPI_SEEK_CUR	
	16	MPI_SEEK_END	
	17	MPI_SEEK_SET	
	18		
	19	F90 Datatype Matching Constants	
	20	C type: const int (or unnamed enum)	
	21	Fortran type: INTEGER	
	22	MPI_TYPECLASS_COMPLEX	
	23	MPI_TYPECLASS_INTEGER	
	24	MPI_TYPECLASS_REAL	
	25		
	26	Constants Specifying Empty or Ignored In	put
	27	C/Fortran name	
	28	C type / Fortran type <sup>1</sup>	
	29	MPI_ARGVS_NULL	
	30	<pre>char*** / 2-dim. array of CHARACTER*(*)</pre>	
	31	MPI_ARGV_NULL	
	32	char** / array of CHARACTER*(*)	
	33	MPI_ERRCODES_IGNORE	
	34	int* / INTEGER array	
	35 36	MPI_STATUSES_IGNORE	
#-error!	37	<pre>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)</pre>	
Done	38	or TYPE(MPI_Status), DIMENSION(*)	TYPE(MPI_Status) should be \mpiftype and ", DIMENSION(*)" should be
in DD444	39	MPI_STATUS_IGNORE	formatted in the same style ???
PR444	40	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)	Compare to here!
	41	or TYPE(MPI_Status) MPI_UNWEIGHTED	
	42	int* / INTEGER array	
	43	MPI_WEIGHTS_EMPTY	
	44	int* / INTEGER array	
	45	<sup>-1</sup> Note that in Fortran these constants are not usable for i	nitialization
	46	expressions or assignment. See Section 2.5.4.	
	47	chrospione of designment, bee beenen area.	
	48		

	C Constants Specify	ing Ignored Input (no Fortran)	<sup>1</sup> #-errc
\mpictype	C type: MPI_Fint*	equivalent to Fortran	2 Done
	MPI_F_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi / mpif.h	<sup>3</sup> in
	MPI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h	4 PR44
\mpictype	C type: MPI_F08_status*	equivalent to Fortran	- 5
	MPI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi_f08	6
	MPI_F08_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi_f08	7
	C proprocessor Con	stants and Fortran Parameters	8
		nacro that expands to an int value	10
	Fortran type: INTEGER		11
	MPI_SUBVERSION		12
	MPI_VERSION		13
	NT. 11 1		14 15
	MPI_T_ENUM_NULL	he MPI tool information interface	16
	MPI_T_enum		17
	MPI_T_CVAR_HANDLE_NU		18
	MPI_T_cvar_handle		19
	MPI_T_PVAR_HANDLE_NU	JLL	20
	MPI_T_pvar_handle		21
	MPI_T_PVAR_SESSION_NU	JLL	22
	MPI_T_pvar_session		23
			24
	Verbosity Levels in th	ne MPI tool information interface	25
	C type: const int (or unn	named enum)	26
	MPI_T_VERBOSITY_USER	R_BASIC	27
	MPI_T_VERBOSITY_USER	R_DETAIL	28
	MPI_T_VERBOSITY_USER		29
	MPI_T_VERBOSITY_TUNE		30
	MPI_T_VERBOSITY_TUNE		31
	MPI_T_VERBOSITY_TUNE		32 33
	MPI_T_VERBOSITY_MPID		34
	MPI_T_VERBOSITY_MPID		35
	MPI_T_VERBOSITY_MPID	DEV_ALL	36
			37
			38
			39
			40
			41
			42
			43
			44
			45
			46
			47
			48

	858	ANNEX A. LANGUAGE BINDINGS SUMMARY
1		Constants to identify associations of variables
2		in the MPI tool information interface
3		C type: const int (or unnamed enum)
4		MPI_T_BIND_NO_OBJECT
5		MPI_T_BIND_MPI_COMM
6		MPI_T_BIND_MPI_DATATYPE
7		MPI_T_BIND_MPI_ERRHANDLER
8		MPI_T_BIND_MPI_FILE
9		MPI_T_BIND_MPI_GROUP
10		MPI_T_BIND_MPI_OP
11		MPI_T_BIND_MPI_REQUEST
12		MPI_T_BIND_MPI_WIN
13		MPI_T_BIND_MPI_MESSAGE
14		MPI_T_BIND_MPI_INFO
15		
16		Constants describing the scope of a control variable
17		in the MPI tool information interface
18		C type: const int (or unnamed enum)
19		MPI_T_SCOPE_CONSTANT
20		MPI_T_SCOPE_READONLY
21		MPI_T_SCOPE_LOCAL
22		MPI_T_SCOPE_GROUP
23		MPI_T_SCOPE_GROUP_EQ
24		MPI_T_SCOPE_ALL
25		MPI_T_SCOPE_ALL_EQ
26		
27		Additional constants used
28 29		by the MPI tool information interface
29 30		C type: MPI_T_pvar_handle
31		MPI_T_PVAR_ALL_HANDLES
32		
33		Performance variables classes used by the MPI tool information interface
34		
35		C type: const int (or unnamed enum)
36		MPI_T_PVAR_CLASS_STATE MPI_T_PVAR_CLASS_LEVEL
37		MPI_T_PVAR_CLASS_LEVEL MPI_T_PVAR_CLASS_SIZE
38		MPI_T_PVAR_CLASS_SIZE MPI_T_PVAR_CLASS_PERCENTAGE
39		MPI_T_PVAR_CLASS_PERCENTAGE MPI_T_PVAR_CLASS_HIGHWATERMARK
40		MPI_T_PVAR_CLASS_HIGHWATERMARK
41		MPI_T_PVAR_CLASS_LOWWATERMARK
42		MPI_T_PVAR_CLASS_COUNTER MPI_T_PVAR_CLASS_AGGREGATE
43		MPI_T_PVAR_CLASS_AGGREGATE MPI_T_PVAR_CLASS_TIMER
44		MPI_T_PVAR_CLASS_TIMER MPI_T_PVAR_CLASS_GENERIC
45		
46		
47		
48		



```
1
        \mathbf{2}
        3
                 The following are defined Fortran type definitions, included in the mpi_f08 and mpi
        4
             modules.
        5
             ! Fortran opaque types in the mpi_f08 and mpi modules
        6
 #-error
             TYPE(MPI_Status)
        7
Done
                Fortran handles in the mpi_f08 and mpi modules
             !
PR444
        9
             TYPE(MPI_Comm)
        10
             TYPE(MPI_Datatype)
        11
                                      All such terms additionally indexed
             TYPE(MPI_Errhandler)
        12
                                       though \cdeclindex(MPI Xxxxx).
             TYPE(MPI_File)
                                      and these should be removed.
        13
             TYPE(MPI_Group)
        14
             TYPE(MPI_Info)
        15
             TYPE(MPI_Message)
        16
             TYPE(MPI_Op)
        17
             TYPE(MPI_Request)
        18
             TYPE(MPI_Session)
        19
             TYPE(MPI_Win)
        20
       21
             A.1.3 Prototype Definitions
       22
       23
             C Bindings
       24
             The following are defined C typedefs for user-defined functions, also included in the file
       25
             mpi.h.
        26
       27
             /* prototypes for user-defined functions */
       28
             typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
       29
                            MPI_Datatype *datatype);
        30
        31
   #-...
             typedef void MPI_User_function_c(void *invec, void *inoutvec,
       32
                            MPI_Count *len, MPI_Datatype *datatype);
       33
       34
             typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
                            void *extra_state, void *attribute_val_in,
       35
                            void *attribute_val_out, int *flag);
       36
       37
             typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
       38
                            void *attribute_val, void *extra_state);
       39
       40
             typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
       41
                            void *extra_state, void *attribute_val_in,
       42
                            void *attribute_val_out, int *flag);
       43
             typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
       44
                            void *attribute_val, void *extra_state);
       45
        46
             typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
        47
                            int type_keyval, void *extra_state, void *attribute_val_in,
        48
                            void *attribute_val_out, int *flag);
```

ANNEX A. LANGUAGE BINDINGS SUMMARY

**Unofficial Draft for Comment Only** 

in

After the merge of PR409, the remaining \typedef.. must be checked and may be removed.

```
862
                                                  ANNEX A. LANGUAGE BINDINGS SUMMARY
                                                           All mpi f08 call back prototypes will be referenced in the
#-error!
              Fortran 2008 Bindings with the mpi_f08 Module
                                                           callback function prototype index
#-update5
         \mathbf{2}
              The callback prototypes when using the Fortran mpi_f08 module are shown below:
         3
                   The user-function argument to MPI_Op_create should be declared according to:
         4
 PR444
              ABSTRACT INTERFACE
         \mathbf{5}
                SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
         6
                   USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
         7
                   TYPE(C_PTR), VALUE :: invec, inoutvec
         8
                   INTEGER :: len
         9
                   TYPE(MPI_Datatype) :: datatype
         10
         11
              ABSTRACT INTERFACE
    #-...
        12
                SUBROUTINE MPI_User_function_c(invec, inoutvec, len, datatype)
         13
                   USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
         14
                   TYPE(C_PTR), VALUE :: invec, inoutvec
         15
                   INTEGER(KIND=MPI_COUNT_KIND) :: len
         16
                   TYPE(MPI_Datatype) :: datatype
         17
                  The copy and delete function arguments to MPI_Comm_create_keyval should be de-
         18
              clared according to:
         19
              ABSTRACT INTERFACE
         20
                 SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
         21
                              attribute_val_in, attribute_val_out, flag, ierror)
         22
                   TYPE(MPI_Comm) :: oldcomm
         23
                   INTEGER :: comm_keyval, ierror
         24
                   INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
         25
                              attribute_val_out
         26
                   LOGICAL :: flag
         27
         28
              ABSTRACT INTERFACE
         29
                SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
         30
                              attribute_val, extra_state, ierror)
         31
                   TYPE(MPI_Comm) :: comm
         32
                   INTEGER :: comm_keyval, ierror
         33
                   INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
         34
                   The copy and delete function arguments to MPI_Win_create_keyval should be declared
         35
              according to:
         36
              ABSTRACT INTERFACE
         37
                SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
         38
                              attribute_val_in, attribute_val_out, flag, ierror)
         39
                   TYPE(MPI_Win) :: oldwin
         40
                   INTEGER :: win_keyval, ierror
         41
                   INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
         42
                              attribute_val_out
         43
                   LOGICAL :: flag
         44
         45
              ABSTRACT INTERFACE
         46
                SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
         47
                              extra_state, ierror)
```

Done

48

lin

TYPE(MPI_Win) :: win INTEGER :: win_keyval, ierror INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state MPI_Session_delete_attr_function() is NOT missing	1 Many out-commented lines starting with %DAN-PyMerge% should be deleted???
<pre>Ine copy and delete function arguments to MPI_Type_create_keyval should be decording to: ABSTRACT INTERFACE SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_st</pre>	6 Done 7 in
ABSTRACT INTERFACE SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,	14 15 16 17 18 19 20
The handler-function argument to MPI_Comm_create_errhandler should be deelike this: ABSTRACT INTERFACE SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) TYPE(MPI_Comm) :: comm INTEGER :: error_code	21 22 23 24 25 26 27
The handler-function argument to MPI_Win_create_errhandler should be declare this: ABSTRACT INTERFACE SUBROUTINE MPI_Win_errhandler_function(win, error_code) TYPE(MPI_Win) :: win INTEGER :: error_code	29 30 31 32 33
The handler-function argument to MPI_File_create_errhandler should be declare this: ABSTRACT INTERFACE SUBROUTINE MPI_File_errhandler_function(file, error_code) TYPE(MPI_File) :: file INTEGER :: error_code	<sup>34</sup> ed like 35 36 37 38 39 40
ABSTRACT INTERFACE SUBROUTINE MP1_File_errhandler_function(file, error_code) TYPE(MPI_File) :: file INTEGER :: error_code	
The handler-function argument to MPI_Session_create_errhandler should be dealike this: ABSTRACT INTERFACE	

```
864
```

ANNEX A. LANGUAGE BINDINGS SUMMARY

```
SUBROUTINE MPI_Session_errhandler_function(session, error_code)
 #-...
        2
                  TYPE(MPI_Session) :: session
        3
                 INTEGER :: error_code
             ABSTRACT INTERFACE
#-error!
               SUBROUTINE_MF1_Session_errhandler_function(session, error_code) Old-style-formatted
Done
        6
                  TYPE(MPI Session) :: session
                                                                                     duplicate should be
lin
        7
                                                                                     removed
PR444
                  INTEGER :: error_code
        8
        9
                  The query, free, and cancel function arguments to MPI_Grequest_start should be de-
        10
             clared according to:
        11
             ABSTRACT INTERFACE
        12
               SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
        13
                  INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
        14
                  TYPE(MPI_Status) :: status
        15
                  INTEGER :: ierror
        16
             ABSTRACT INTERFACE
        17
               SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
        18
                  INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
        19
                  INTEGER :: ierror
        20
        21
             ABSTRACT INTERFACE
        22
               SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
        23
                  INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
        24
                  LOGICAL :: complete
        25
                  INTEGER :: ierror
        26
                 The extent and conversion function arguments to MPI_Register_datarep should be de-
        27
             clared according to:
        28
             ABSTRACT INTERFACE
        29
               SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
        30
                            ierror)
        31
                  TYPE(MPI_Datatype) :: datatype
        32
                  INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
        33
                  INTEGER :: ierror
        34
        35
             ABSTRACT INTERFACE
        36
                SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
        37
                            filebuf, position, extra_state, ierror)
        38
                  USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        39
                  TYPE(C_PTR), VALUE :: userbuf, filebuf
        40
                  TYPE(MPI_Datatype) :: datatype
        41
                  INTEGER :: count, ierror
        42
                  INTEGER(KIND=MPI_OFFSET_KIND) :: position
        43
                  INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
        44
    #-... 45
             ABSTRACT INTERFACE
               SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count,
        46
                            filebuf, position, extra_state, ierror)
        47
                  USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        ^{48}
```

#	TYPE(C_PTR), VALUE :: userbuf, filebuf	1 2
	TYPE(MPI_Datatype) :: datatype INTEGER(KIND=MPI_COUNT_KIND) :: count	3
	INTEGER(KIND=MPI_COUNI_KIND) :: count INTEGER(KIND=MPI_OFFSET_KIND) :: position	4
	INTEGER(KIND-MPI_OFFSEI_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	5
	INTEGER(KIND-MFI_ADDRESS_KIND) extra_state	6
	INTEGER IEITOT	7
	All mpi_f08 call back prototypes will be	<sup>8</sup> #-error!
	Fortran Bindings with mpif.h or the mpi Module referenced in the callback function prototype	9 #-update5
	With the Fortran mpi module or mpif.h, here are examples of how each of the user-defined subroutines should be declared.	10 <b>Done</b> 11 <b>in</b>
	The user-function argument to MPI_OP_CREATE should be declared like this:	12 PR444
Done in	SUBROUTINE USER_FUNCTION (INVEC, INOUTVEC, LEN, DATATYPE)	13
PR449	<pre><type> INVEC(LEN), INOUTVEC(LEN)</type></pre>	14
	INTEGER LEN, DATATYPE	15
		16
	The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be	17
	declared like these:	18
	SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,	19
	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	20
	INTEGER OLDCOMM, COMM_KEYVAL, IERROR	21
	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	22
	ATTRIBUTE_VAL_OUT	23
	LOGICAL FLAG	24
	SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,	25
	EXTRA_STATE, IERROR)	26
	INTEGER COMM, COMM_KEYVAL, IERROR	27
	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	28
	The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be	29
	declared like these:	30
	SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,	31 32
	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	32
	INTEGER OLDWIN, WIN_KEYVAL, IERROR	34
	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	35
	ATTRIBUTE_VAL_OUT	36
	LOGICAL FLAG	37
		38
	SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,	39
	EXTRA_STATE, IERROR)	40
	INTEGER WIN, WIN_KEYVAL, IERROR	41
	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	42
	The delete function argument to MPI_SESSION_CREATE_KEXVAL should be declared	43 #-error!
	like this: MPI_Session_delete_attr_function() is NOT needed	44 Done 45 in
	SUBROUTINE SESSION_DELETE_ATTB_FUNCTION(SESSION, SESSION_KEYVAL,	46 PR444
	ATTRIBUTE_VAL, EXTRA_STATE, IERROR)	47
	INTEGER SESSION, SESSION_KEYVAL, IERROR	48

ANNEX A. LANGUAGE BINDINGS SUMMARY

	1	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
	2 3	The copy and delete function arguments to $MPI\_TYPE\_CREATE\_KEYVAL$ should be
	4	declared like these:
	5	SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
	6	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
	7	INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
	8	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
	9	ATTRIBUTE_VAL_OUT
	10	LOGICAL FLAG
	11	SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
	12	EXTRA_STATE, IERROR)
	13	INTEGER DATATYPE, TYPE_KEYVAL, IERROR
	14 15	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
	16	The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de-
	17	clared like this:
	18	SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
	19	INTEGER COMM, ERROR_CODE
	20	
	21	The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-
	22	clared like this:
	23	SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
	24	INTEGER WIN, ERROR_CODE
	25 26	The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:
	27	SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
	28	INTEGER FILE, ERROR_CODE
#	29 30	The handler-function argument to MPI_SESSION_CREATE_ERRHANDLER should be
π	• 31	declared like this: Many out-commented
	32	SUBROUTINE SESSION_ERRHANDLER_FUNCTION (SESSION, ERROR_CODE) lines starting with NTECED SESSION_EDDOR_CODE // // // // // // // // // // // // //
	33	INTEGER SESSION, ERROR_CODE should be deleted???
#-error! Done	34	The query, free, and cancel function arguments to MPI_GREQUEST_START should be
in	35	declared like these:
PR444	36	SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
	37	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
	38	INTEGER STATUS(MPI_STATUS_SIZE), IERROR
	$\frac{39}{40}$	SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
	40 41	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
	41	INTEGER IERROR
	43	
	44	SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR) INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
	45	LOGICAL COMPLETE
	46	INTEGER IERROR
	47	
	48	

The extent and conversion function arguments to	MPI_REGISTER_DATAREP should	1	
be declared like these:		2 3	
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, F INTEGER DATATYPE, IERROR	LAIENI, EAIRA_SIAIE, IERROR)	4	
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXT	TRA STATE	5	
	_	6	
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBU	F, DATATYPE, COUNT, FILEBUF,	7	
POSITION, EXTRA_STATE, IERROR) <type> USERBUF(*), FILEBUF(*)</type>		8	
INTEGER DATATYPE, COUNT, IERROR		9	
INTEGER(KIND=MPI_OFFSET_KIND) POSITION		10	
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	E	11 12	
		12	
A 1.4 Democrated Dustations Definitions		14	
A.1.4 Deprecated Prototype Definitions		15	
The following are defined C typedefs for deprecated use	er-defined functions, also included in	16	
the file mpi.h.		17	
(* prototypes for year defined functions */		18	
<pre>/* prototypes for user-defined functions */</pre>		19	
typedef int MPI_Copy_function(MPI_Comm oldcomm	n, int keyval,	20 21	
void *extra_state, void *attribu		21	
<pre>void *attribute_val_out, int *fl</pre>	ag);	23	
typedef int MPI_Delete_function(MPI_Comm comm,	. int kevval.	24	
void *attribute_val, void *extra		25	
The following are deprecated Fortran user-defined	callback subrouting prototypes. The	26	
deprecated copy and delete function arguments to MI	* **	27	
clared like these:		28 29	
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTR	RA_STATE, ATTRIBUTE_VAL_IN,	30	
ATTRIBUTE_VAL_OUT, FLAG, IERR)		31	
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTH	RIBUTE_VAL_IN,	32	
ATTRIBUTE_VAL_OUT, IERR		33	
LOGICAL FLAG		34	
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRI		35	
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA	A_STATE, IERR	36 37 <u> </u>	
A.1.5 Process Set Names		<sup>37</sup> #-upda	te5
A.1.5 Info Keys		<sup>39</sup> In	
The following info keys are reserved. They are strings.		<sup>40</sup> PR444	4
"access_style"	Other constants in the Index of Constants	S:	1
"accumulate_ops"		_	
"accumulate_ordering"	"hwloc://L3Cache" example for a PSET names "mpi://" reserved namespace for standardized F		
"alloc_shared_noncontig"	"mpi://SELF" standardized		
"appnum"	"mpi://WORLD" PSET names		
"arch" "cb_block_size"	"mpix://UNIVERSE" example for a PSET nam	es	
	"mpi_thread_support_level"		

	1	"cb_buffer_size"
	2	"cb_nodes"
	3	"chunked_item"
#-error!		"chunked_size"
	5	"chunked"
	6	"collective_buffering"
	7	"file" "file perm" Additionally:
Done in	8	me_perm
PR444	9	"filename" "command" (was already missing in MPI-3.1)
	10	"host" "maxprocs" (was already missing in MPI-3.1)
	11	"io_node_list" "mpi_size"> new for Sessions>PR461
	12	"ip_address" "thread_level" (was already missing in MPI-3.1)
	13	"ip_port"
#	14	"mpi_assert_allow_overtaking"
	15	"mpi_assert_exact_length"
#-error	16	"mpi_assert_no_any_source"
Done	17	"mpi_assert_no_any_tag"
in	18	"mpi_assert_strict_start_ordering" Why is this one not listed in the
PR444	19	"mpi_hw_resource_type" Constants Index ?
	20	"mpi_initial_errhandler"
#-error!	21 _	"mpi_optimization_goal" does not exist in MPI-4.0
#-update9	22 _	"mpi_reuse_count" does not exist in MPI-4.0
Done	23	"mpi_minimum_memory_alignment" Wrong alphabetic order These two entries must be removed, see
in	24	"nb_proc" Done https://lists.mpi-forum.org/pipermail/mpiwg-
PR449	25	"no locks" in persistence/2019-August/000082.html
	26	"num_io_nodes"
	27	"path"
	28	"same_disp_unit"
	29	"same_size"
	30	"soft"
	31	"striping_factor"
	32	"striping_unit"
	33	"wdir"
	34	
	35	
	36	A.1.6 Info Values
	37	
	38	The following info values are reserved. They are strings.
	39	"false"
#	40	"mpi_errors_abort"
π	41	"mpi_errors_are_fatal" Missing:
	42	"mpi_errors_return" "MPI_THREAD_FUNNELED" (was already missing in MPI-3.1)
#-error!	43	"mpi_shared_memory" "MPI_THREAD_MULTIPLE" (was already missing in MPI-3.1)
Done	44	"random" "MPI_THREAD_SERIALIZED" (was already missing in MPI-3.1)
in PR444	45	"rar" "MPI_THREAD_SINGLE" (was already missing in MPI-3.1)
FK444	46	"raw" "none"
	47	"read_mostly"
	48	"read_once"

"reverse_sequential"	1
"same_op"	2
"same_op_no_op"	3
"sequential"	4
"true"	5
"war"	6
"waw"	7
"write_mostly"	8
"write_once"	9
	10
A.2 Summary of the Semantics of all Operation-Related MPI Procedures	11 12 <b>#</b>
A summary of the semantics of all operation-related MPI procedures can be found in [51].	13
It summary of the semantics of an operation related with procedures can be found in [01].	14
	15
	16
	17
	18
	19
	20
	21
	22
	23
	24
	25
	26 27
	28
	28
	30
	31
	32
	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

1 2	A.3 C Bindings	The MPI-3.1 hyperrefs to the routine specification itself and to the primary references for all MPI type declarations are
3	A.3.1 Point-to-Point Communication C Bindings	missing> Issue #413
#-error! <sup>4</sup> #-413 <sup>5</sup> <sub>6</sub>	<pre>int MPI_Bsend(const void *buf, int count, M</pre>	PI_Datatype datatype, int dest,
<b>#-TODO</b> 7 8	<pre>int MPI_Bsend_c(const void *buf, MPI_Count</pre>	VI VI
9 10 11	<pre>int MPI_Bsend_init(const void *buf, int cou int dest, int tag, MPI_Comm co</pre>	
12 13 14	int MPI_Bsend_init_c(const void *buf, MPI_C MPI_Datatype datatype, int des MPI_Request *request)	
15 16	int MPI_Buffer_attach(void *buffer, int siz	e)
17	<pre>int MPI_Buffer_attach_c(void *buffer, MPI_C</pre>	ount size)
18 19	<pre>int MPI_Buffer_detach(void *buffer_addr, in</pre>	t *size)
20	int MPI_Buffer_detach_c(void *buffer_addr, 1	MPI_Count *size)
21 22	<pre>int MPI_Cancel(MPI_Request *request)</pre>	
23	<pre>int MPI_Get_count(const MPI_Status *status,</pre>	MPI_Datatype datatype,
24 25	int *count)	
26 27	<pre>int MPI_Get_count_c(const MPI_Status *statu MPI_Count *count)</pre>	s, MPI_Datatype datatype,
28 29	<pre>int MPI_Ibsend(const void *buf, int count, i</pre>	VI VI
30 31 32	<pre>int MPI_Ibsend_c(const void *buf, MPI_Count</pre>	VI VI
33 34	int MPI_Improbe(int source, int tag, MPI_Com MPI_Message *message, MPI_Stat	6
35 36	int MPI_Imrecv(void *buf, int count, MPI_Da	
37	MPI_Message *message, MPI_Requ	
38 39	<pre>int MPI_Imrecv_c(void *buf, MPI_Count count</pre>	VI VI
40 41 42	int MPI_Iprobe(int source, int tag, MPI_Com MPI_Status *status)	n comm, int *flag,
43 44	int MPI_Irecv(void *buf, int count, MPI_Dat int tag, MPI_Comm comm, MPI_Re	
45 $46$ $47$	<pre>int MPI_Irecv_c(void *buf, MPI_Count count,</pre>	MPI_Datatype datatype,
48	int source, int tag, mri_00mm	comm, in r_nequest *request)

870

int	int tag, MPI_Comm comm, MPI_Request *request)	1 2
int	MPI_Irsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype, int destint_tagMPI_Comm_commMPI_Request *request)	3 4 5
int	int tag, MPI_Comm comm, MPI_Request *request)	6 7 8
int	MPI_Isend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,	9 10
int	<pre>MPI_Isendrecv(const void *sendbul, int sendcount,</pre>	11 12 13 14 15
int	<pre>MPI_Isendrecv_replace(void *buf, int count, MPI_Datatype datatype,</pre>	15 16 17 18
int	MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	19 20 21
int	MPI_Issend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,	22 23
int	MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)	24 25 26
int		27 28
int	MPI_Mrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Message *message, MPI_Status *status)	29 30 31
int	MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)	32
int	<pre>MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,</pre>	33 34 35
int	int source, int tag, MPI_Comm comm, MPI_Status *status)	36 37
int	MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, int tag. MPI Comm comm. MPI Request *request)	38 39 40
int	int source, int tag, MPI_Comm comm, MPI_Request *request)	41 42
int	MPT Request free(MPT Request *request)	$\frac{43}{44}$
int	MPI_Status *status)	45 46
int	MPT Reand(const void thuf int count MPT Datatype datatype int dest	47 48

1	int tag, MPI_Comm comm)
2 3 4	<pre>int MPI_Rsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>
5 6 7	<pre>int MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype,</pre>
8 9 10	<pre>int MPI_Rsend_init_c(const void *buf, MPI_Count count,</pre>
11 12 13	<pre>int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
13 14 15	<pre>int MPI_Send_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>
16 17 18	<pre>int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype,</pre>
19 20 21	<pre>int MPI_Send_init_c(const void *buf, MPI_Count count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)</pre>
22 23 24 25 26	<pre>int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,</pre>
27 28 29	<pre>int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,</pre>
30 31 32	<pre>int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
33 34	<pre>int MPI_Ssend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>
35 36 37	<pre>int MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype,</pre>
38 39 40	<pre>int MPI_Ssend_init_c(const void *buf, MPI_Count count,</pre>
41 42	<pre>int MPI_Start(MPI_Request *request)</pre>
43	<pre>int MPI_Startall(int count, MPI_Request array_of_requests[])</pre>
44 45	<pre>int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)</pre>
46	<pre>int MPI_Test_cancelled(const MPI_Status *status, int *flag)</pre>
47 48	<pre>int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag,</pre>

<pre>MPI_Status array_of_statuses[])</pre>	1 2
<pre>int MPI_Testany(int count, MPI_Request array_of_requests[], int *index,</pre>	3
int *flag, MPI_Status *status)	4
<pre>int MPI_Testsome(int incount, MPI_Request array_of_requests[],</pre>	5
int *outcount, int array_of_indices[],	6
MPI_Status array_of_statuses[])	7
int MPI_Wait(MPI_Request *request, MPI_Status *status)	8
Int Mri_wait(Mri_Nequest *iequest, Mri_Status *status)	9
<pre>int MPI_Waitall(int count, MPI_Request array_of_requests[],</pre>	10 11
<pre>MPI_Status array_of_statuses[])</pre>	12
<pre>int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index,</pre>	13
MPI_Status *status)	14
<pre>int MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>	15
int *outcount, int array_of_indices[],	16
MPI_Status array_of_statuses[])	17
	18
A.3.2 Partitioned Communication C Bindings	19 20
A.J.2 Tartitioned communication C bindings	20
<pre>int MPI_Parrived(MPI_Request *request, int partition, int *flag)</pre>	22
<pre>int MPI_Pready(int partition, MPI_Request *request)</pre>	23
<pre>int MPI_Pready_list(int length, int array_of_partitions[],</pre>	24
MPI_Request *request)	25
	26
int MPI_Pready_range(int partition_low, int partition_high,	27 28
MPI_Request *request)	29
<pre>int MPI_Precv_init(void *buf, int partitions, MPI_Count count,</pre>	30
MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,	31
MPI_Info info, MPI_Request *request)	32
<pre>int MPI_Psend_init(void *buf, int partitions, MPI_Count count,</pre>	33
MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,	34
MPI_Info info, MPI_Request *request)	35 36
	37
A.3.3 Datatypes C Bindings	38
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp) These routine were lost due to	<sup>39</sup> #-error!
MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)	40 41 <b>#-413</b>
<pre>int MPI_Get_address(const void *location, MPI_Aint *address)</pre>	42 <b>#-PR414</b>
	<sup>43</sup> #-update2
<pre>int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,</pre>	<sup>44</sup> Done <sup>45</sup> in
	46 PR431
<pre>int MPI_Get_elements_c(const MPI_Status *status, MPI_Datatype datatype,</pre>	47
MPI_Count *count)	48

1 int MPI\_Get\_elements\_x(const MPI\_Status \*status, MPI\_Datatype datatype,  $\mathbf{2}$ MPI\_Count \*count) 3 int MPI\_Pack(const void \*inbuf, int incount, MPI\_Datatype datatype, 4 void \*outbuf, int outsize, int \*position, MPI\_Comm comm) 56 int MPI\_Pack\_c(const void \*inbuf, MPI\_Count incount, MPI\_Datatype datatype, 7 void \*outbuf, MPI\_Count outsize, MPI\_Count \*position, 8 MPI\_Comm comm) 9 int MPI\_Pack\_external(const char datarep[], const void \*inbuf, int incount, 10 MPI\_Datatype datatype, void \*outbuf, MPI\_Aint outsize, 11 MPI\_Aint \*position) 1213int MPI\_Pack\_external\_c(const char datarep[], const void \*inbuf, 14MPI\_Count incount, MPI\_Datatype datatype, void \*outbuf, 15MPI\_Count outsize, MPI\_Count \*position) 16int MPI\_Pack\_external\_size(const char datarep[], int incount, 17MPI\_Datatype datatype, MPI\_Aint \*size) 18 19 int MPI\_Pack\_external\_size\_c(const char datarep[], MPI\_Count incount, 20MPI\_Datatype datatype, MPI\_Count \*size) 21int MPI\_Pack\_size(int incount, MPI\_Datatype datatype, MPI\_Comm comm, 22 int \*size) 2324int MPI\_Pack\_size\_c(MPI\_Count incount, MPI\_Datatype datatype, 25MPI\_Comm comm, MPI\_Count \*size) 26int MPI\_Type\_commit(MPI\_Datatype \*datatype) 2728int MPI\_Type\_contiguous(int count, MPI\_Datatype oldtype, 29 MPI\_Datatype \*newtype) 30 int MPI\_Type\_contiguous\_c(MPI\_Count count, MPI\_Datatype oldtype,  $^{31}$ MPI\_Datatype \*newtype) 32 33 int MPI\_Type\_create\_darray(int size, int rank, int ndims, 34 const int array\_of\_gsizes[], const int array\_of\_distribs[], 35 const int array\_of\_dargs[], const int array\_of\_psizes[], 36 int order, MPI\_Datatype oldtype, MPI\_Datatype \*newtype) 37 int MPI\_Type\_create\_darray\_c(int size, int rank, int ndims, 38 const MPI\_Count array\_of\_gsizes[], 39 const int array\_of\_distribs[], const int array\_of\_dargs[], 4041 const int array\_of\_psizes[], int order, MPI\_Datatype oldtype, 42MPI\_Datatype \*newtype) 43 int MPI\_Type\_create\_hindexed(int count, const int array\_of\_blocklengths[], 44 const MPI\_Aint array\_of\_displacements[], MPI\_Datatype oldtype, 45MPI\_Datatype \*newtype) 4647int MPI\_Type\_create\_hindexed\_block(int count, int blocklength, 48 const MPI\_Aint array\_of\_displacements[], MPI\_Datatype oldtype,

	MPI_Datatype *newtype)	1
int MDT Trees		2
IIIC MPI_Type	_create_hindexed_block_c(MPI_Count count, MPI_Count blocklength,	3
	const MPI_Count array_of_displacements[],	4
	MPI_Datatype oldtype, MPI_Datatype *newtype)	5 6
int MDT Trong		7
IIIC MFI_Type	<pre>_create_hindexed_c(MPI_Count count, const MPI_Count array_of_blocklengths[],</pre>	8
	const MPI_Count array_of_displacements[],	9
	MPI_Datatype oldtype, MPI_Datatype *newtype)	10
int MDT Turno	_create_hvector(int count, int blocklength, MPI_Aint stride,	11
int mri_iype	MPI_Datatype oldtype, MPI_Datatype *newtype)	12
		13 14
int MPI_Type	_create_hvector_c(MPI_Count count, MPI_Count blocklength,	15
	MPI_Count stride, MPI_Datatype oldtype, MPI_Datatype *newtype)	16
int MPI_Type	_create_indexed_block(int count, int blocklength,	17
	<pre>const int array_of_displacements[], MPI_Datatype oldtype,</pre>	18
	MPI_Datatype *newtype)	19
int MPI_Type	_create_indexed_block_c(MPI_Count count, MPI_Count blocklength,	20
	<pre>const MPI_Count array_of_displacements[],</pre>	21 22
	MPI_Datatype oldtype, MPI_Datatype *newtype)	22
int MPI_Type	<pre>_create_resized(MPI_Datatype oldtype, MPI_Aint lb,</pre>	24
	MPI_Aint extent, MPI_Datatype *newtype)	25
int MPI Type	_create_resized_c(MPI_Datatype oldtype, MPI_Count lb,	26
ing uni_iybo	MPI_Count extent, MPI_Datatype *newtype)	27
		28
int MPI_Type	<pre>_create_struct(int count, const int array_of_blocklengths[], const MPI_Aint array_of_displacements[],</pre>	29
	const MPI_Datatype array_of_types[], MPI_Datatype *newtype)	30 31
		32
int MPL_Type	_create_struct_c(MPI_Count count,	33
	<pre>const MPI_Count array_of_blocklengths[], const MPI_Count array_of_displacements[],</pre>	34
	const MPI_Datatype array_of_types[], MPI_Datatype *newtype)	35
		36
int MPI_Type	_create_subarray(int ndims, const int array_of_sizes[],	37
	<pre>const int array_of_subsizes[], const int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>	38 39
		40
int MPI_Type	_create_subarray_c(int ndims, const MPI_Count array_of_sizes[],	41
	<pre>const MPI_Count array_of_subsizes[], const MPI_Count array of starts[] int order</pre>	42
	const MPI_Count array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)	43
		44
int MPI_Type	_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)	45
int MPI_Type	_free(MPI_Datatype *datatype)	46 47
	_get_contents(MPI_Datatype datatype, int max_integers,	47 48
THE HTTTYPE	Int max_integers,	

```
1
                   int max_addresses, int max_datatypes, int array_of_integers[],
\mathbf{2}
                   MPI_Aint array_of_addresses[],
3
                   MPI_Datatype array_of_datatypes[])
4
     int MPI_Type_get_contents_c(MPI_Datatype datatype, MPI_Count max_integers,
5
                   MPI_Count max_addresses, MPI_Count max_large_counts,
6
                   MPI_Count max_datatypes, int array_of_integers[],
7
                   MPI_Aint array_of_addresses[],
8
                   MPI_Count array_of_large_counts[],
9
                   MPI_Datatype array_of_datatypes[])
10
11
     int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
12
                   int *num_addresses, int *num_datatypes, int *combiner)
13
     int MPI_Type_get_envelope_c(MPI_Datatype datatype, MPI_Count *num_integers,
14
                   MPI_Count *num_addresses, MPI_Count *num_large_counts,
15
                   MPI_Count *num_datatypes, int *combiner)
16
17
     int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
18
                   MPI_Aint *extent)
19
     int MPI_Type_get_extent_c(MPI_Datatype datatype, MPI_Count *lb,
20
                   MPI_Count *extent)
21
22
     int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb,
23
                   MPI_Count *extent)
^{24}
     int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
25
                   MPI_Aint *true_extent)
26
27
     int MPI_Type_get_true_extent_c(MPI_Datatype datatype, MPI_Count *true_lb,
28
                   MPI_Count *true_extent)
29
     int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
30
                   MPI_Count *true_extent)
31
32
     int MPI_Type_indexed(int count, const int array_of_blocklengths[],
33
                   const int array_of_displacements[], MPI_Datatype oldtype,
34
                   MPI_Datatype *newtype)
35
     int MPI_Type_indexed_c(MPI_Count count,
36
                   const MPI_Count array_of_blocklengths[],
37
                   const MPI_Count array_of_displacements[],
38
                   MPI_Datatype oldtype, MPI_Datatype *newtype)
39
40
     int MPI_Type_size(MPI_Datatype datatype, int *size)
41
42
     int MPI_Type_size_c(MPI_Datatype datatype, MPI_Count *size)
43
     int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)
44
45
     int MPI_Type_vector(int count, int blocklength, int stride,
46
                   MPI_Datatype oldtype, MPI_Datatype *newtype)
47
     int MPI_Type_vector_c(MPI_Count count, MPI_Count blocklength,
48
```

ANNEX A. LANGUAGE BINDINGS SUMMARY

876

<pre>MPI_Count stride, MPI_Datatype oldtype, MPI_Dataty</pre>	
<pre>int MPI_Unpack(const void *inbuf, int insize, int *position, voi</pre>	5
<pre>int MPI_Unpack_c(const void *inbuf, MPI_Count insize, MPI_Count void *outbuf, MPI_Count outcount, MPI_Datatype dat MPI_Comm comm)</pre>	•
<pre>int MPI_Unpack_external(const char datarep[], const void *inbuf</pre>	<b>,</b> 9
<pre>int MPI_Unpack_external_c(const char datarep[], const void *inbu MPI_Count insize, MPI_Count *position, void *outbu MPI_Count outcount, MPI_Datatype datatype)</pre>	
A.3.4 Collective Communication C Bindings	17
<pre>int MPI_Allgather(const void *sendbuf, int sendcount,</pre>	20
<pre>int MPI_Allgather_c(const void *sendbuf, MPI_Count sendcount,</pre>	21 22 cvcount, 23 24
<pre>int MPI_Allgather_init(const void *sendbuf, int sendcount,</pre>	27
<pre>int MPI_Allgather_init_c(const void *sendbuf, MPI_Count sendcour MPI_Datatype sendtype, void *recvbuf, MPI_Count re MPI_Datatype recvtype, MPI_Comm comm, MPI_Info inf MPI_Request *request)</pre>	ecvcount, <sup>31</sup>
<pre>int MPI_Allgatherv(const void *sendbuf, int sendcount,</pre>	30
<pre>int MPI_Allgatherv_c(const void *sendbuf, MPI_Count sendcount,</pre>	38 39
<pre>int MPI_Allgatherv_init(const void *sendbuf, int sendcount,</pre>	44
<pre>int MPI_Allgatherv_init_c(const void *sendbuf, MPI_Count sendcom MPI_Datatype sendtype, void *recvbuf,</pre>	unt, 47 48

	878	ANNEX A. LANGUAGE BINDINGS SUMMARY
1 2 3		<pre>const MPI_Count recvcounts[], const MPI_Aint displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>
4 5 6	int	<pre>MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>
7 8 9	int	<pre>MPI_Allreduce_c(const void *sendbuf, void *recvbuf, MPI_Count count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>
10 11 12	int	<pre>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)</pre>
13 14 15	int	<pre>MPI_Allreduce_init_c(const void *sendbuf, void *recvbuf,</pre>
16 17 18 19	int	<pre>MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</pre>
20 21 22 23	int	<pre>MPI_Alltoall_c(const void *sendbuf, MPI_Count sendcount,</pre>
24 25 26 27	int	<pre>MPI_Alltoall_init(const void *sendbuf, int sendcount,</pre>
28 29 30 31 32	int	<pre>MPI_Alltoall_init_c(const void *sendbuf, MPI_Count sendcount,</pre>
33 34 35 36	int	<pre>MPI_Alltoallv(const void *sendbuf, const int sendcounts[],</pre>
37 38 39 40 41 42	int	<pre>MPI_Alltoallv_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>
43 44 45 46 47 48	int	<pre>MPI_Alltoallv_init(const void *sendbuf, const int sendcounts[],</pre>

int	<pre>MPI_Alltoallv_init_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	1
	<pre>const MPI_Aint sdispls[], MPI_Datatype sendtype,</pre>	2
	<pre>void *recvbuf, const MPI_Count recvcounts[],</pre>	3
	<pre>const MPI_Aint rdispls[], MPI_Datatype recvtype,</pre>	4
	MPI_Comm comm, MPI_Info info, MPI_Request *request)	5
	MDT Allterland and a down buf or stirt and for the []	6
int	MPI_Alltoallw(const void *sendbuf, const int sendcounts[],	7
	<pre>const int sdispls[], const MPI_Datatype sendtypes[],</pre>	8
	<pre>void *recvbuf, const int recvcounts[], const int rdispls[],</pre>	9
	<pre>const MPI_Datatype recvtypes[], MPI_Comm comm)</pre>	10
int	MPI_Alltoallw_c(const void *sendbuf, const MPI_Count sendcounts[],	11
	<pre>const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],</pre>	12
	void *recvbuf, const MPI_Count recvcounts[],	13
	<pre>const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],</pre>	14
	MPI_Comm comm)	15
		16
int	<pre>MPI_Alltoallw_init(const void *sendbuf, const int sendcounts[],</pre>	17
	<pre>const int sdispls[], const MPI_Datatype sendtypes[],</pre>	18
	<pre>void *recvbuf, const int recvcounts[], const int rdispls[],</pre>	19
	const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,	20
	MPI_Request *request)	21
int	<pre>MPI_Alltoallw_init_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	22
1110	<pre>const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],</pre>	23
	void *recvbuf, const MPI_Count recvcounts[],	24
	<pre>const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],</pre>	25
	MPI_Comm comm, MPI_Info info, MPI_Request *request)	26
	In 1_00mm comm, In 1_Inio Inio, In 1_Request (Iequest)	27
int	MPI_Barrier(MPI_Comm comm)	$^{28}$
int	<pre>MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	29 30
int	MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,	31
	MPI_Comm comm)	32
		33
int	MPI_Bcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype,	34
	int root, MPI_Comm comm)	35
int	MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype,	36
	int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)	37
		38
int	MPI_Bcast_init_c(void *buffer, MPI_Count count, MPI_Datatype datatype,	39
	int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)	40
int	MPI_Exscan(const void *sendbuf, void *recvbuf, int count,	41
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	42
		43
int	<pre>MPI_Exscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>	44
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	45
int	MPI_Exscan_init(const void *sendbuf, void *recvbuf, int count,	46
0	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	47
	MPI_Info info, MPI_Request *request)	48
	in r_ruro ruro, in r_nequebo "requebo)	-

1int MPI\_Exscan\_init\_c(const void \*sendbuf, void \*recvbuf, MPI\_Count count,  $\mathbf{2}$ MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm, 3 MPI\_Info info, MPI\_Request \*request) 4 int MPI\_Gather(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 5void \*recvbuf, int recvcount, MPI\_Datatype recvtype, int root, 6 MPI Comm comm) 7 8 int MPI\_Gather\_c(const void \*sendbuf, MPI\_Count sendcount, 9 MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 10 MPI\_Datatype recvtype, int root, MPI\_Comm comm) 11 int MPI\_Gather\_init(const void \*sendbuf, int sendcount, 12MPI\_Datatype sendtype, void \*recvbuf, int recvcount, 13 MPI\_Datatype recvtype, int root, MPI\_Comm comm, MPI\_Info info, 14MPI\_Request \*request) 1516int MPI\_Gather\_init\_c(const void \*sendbuf, MPI\_Count sendcount, 17 MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 18 MPI\_Datatype recvtype, int root, MPI\_Comm comm, MPI\_Info info, 19MPI\_Request \*request) 20int MPI\_Gatherv(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 21void \*recvbuf, const int recvcounts[], const int displs[], 22 MPI\_Datatype recvtype, int root, MPI\_Comm comm) 2324int MPI\_Gatherv\_c(const void \*sendbuf, MPI\_Count sendcount, 25MPI\_Datatype sendtype, void \*recvbuf, 26const MPI\_Count recvcounts[], const MPI\_Aint displs[], 27MPI\_Datatype recvtype, int root, MPI\_Comm comm) 28int MPI\_Gatherv\_init(const void \*sendbuf, int sendcount, 29 MPI\_Datatype sendtype, void \*recvbuf, const int recvcounts[], 30 const int displs[], MPI\_Datatype recvtype, int root, 31MPI\_Comm comm, MPI\_Info info, MPI\_Request \*request) 32 33int MPI\_Gatherv\_init\_c(const void \*sendbuf, MPI\_Count sendcount, 34 MPI\_Datatype sendtype, void \*recvbuf, 35 const MPI\_Count recvcounts[], const MPI\_Aint displs[], 36 MPI\_Datatype recvtype, int root, MPI\_Comm comm, MPI\_Info info, 37 MPI\_Request \*request) 38 int MPI\_Iallgather(const void \*sendbuf, int sendcount, 39 MPI\_Datatype sendtype, void \*recvbuf, int recvcount, 40MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Request \*request) 41 42int MPI\_Iallgather\_c(const void \*sendbuf, MPI\_Count sendcount, 43 MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 44 MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Request \*request) 45int MPI\_Iallgatherv(const void \*sendbuf, int sendcount, 46MPI\_Datatype sendtype, void \*recvbuf, const int recvcounts[], 47const int displs[], MPI\_Datatype recvtype, MPI\_Comm comm, 48

	MPI_Request *request)	1
int MPI_Ial	lgatherv_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf,	2 3 4
	<pre>const MPI_Count recvcounts[], const MPI_Aint displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	4 5 6
int MPI_Ial	<pre>lreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)</pre>	7 8 9
int MPI_Ial	<pre>lreduce_c(const void *sendbuf, void *recvbuf, MPI_Count count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)</pre>	10 11 12 13
int MPI_Ial	<pre>ltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	14 15 16 17
int MPI_Ial	<pre>ltoall_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	18 19 20
int MPI_Ial	<pre>ltoallv(const void *sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	21 22 23 24 25
int MPI_Ial	<pre>ltoallv_c(const void *sendbuf, const MPI_Count sendcounts[], const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf, const MPI_Count recvcounts[], const MPI_Aint rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	26 27 28 29 30
int MPI_Ial	<pre>ltoallw(const void *sendbuf, const int sendcounts[], const int sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const int recvcounts[], const int rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)</pre>	31 32 33 34 35 36
int MPI_Ial	<pre>ltoallw_c(const void *sendbuf, const MPI_Count sendcounts[], const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const MPI_Count recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)</pre>	37 38 39 40 41
int MPI_Iba	rrier(MPI_Comm comm, MPI_Request *request)	42 43
int MPI_Ibc	ast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)	44 45
int MPI_Ibc	ast_c(void *buffer, MPI_Count count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)	46 47 48

1int MPI\_Iexscan(const void \*sendbuf, void \*recvbuf, int count,  $\mathbf{2}$ MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm, 3 MPI\_Request \*request) 4 int MPI\_Iexscan\_c(const void \*sendbuf, void \*recvbuf, MPI\_Count count, 5MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm, 6 MPI\_Request \*request) 7 8 int MPI\_Igather(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 9 void \*recvbuf, int recvcount, MPI\_Datatype recvtype, int root, 10 MPI\_Comm comm, MPI\_Request \*request) 11 int MPI\_Igather\_c(const void \*sendbuf, MPI\_Count sendcount, 12MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 13 MPI\_Datatype recvtype, int root, MPI\_Comm comm, 14MPI\_Request \*request) 1516int MPI\_Igatherv(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 17 void \*recvbuf, const int recvcounts[], const int displs[], 18 MPI\_Datatype recvtype, int root, MPI\_Comm comm, 19MPI\_Request \*request) 20int MPI\_Igatherv\_c(const void \*sendbuf, MPI\_Count sendcount, 21MPI\_Datatype sendtype, void \*recvbuf, 22 const MPI\_Count recvcounts[], const MPI\_Aint displs[], 23MPI\_Datatype recvtype, int root, MPI\_Comm comm, 24MPI\_Request \*request) 2526int MPI\_Ireduce(const void \*sendbuf, void \*recvbuf, int count, 27MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm, 28 MPI\_Request \*request) 29 int MPI\_Ireduce\_c(const void \*sendbuf, void \*recvbuf, MPI\_Count count, 30 MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm,  $^{31}$ MPI\_Request \*request) 32 33 int MPI\_Ireduce\_scatter(const void \*sendbuf, void \*recvbuf, 34 const int recvcounts[], MPI\_Datatype datatype, MPI\_Op op, 35 MPI\_Comm comm, MPI\_Request \*request) 36 int MPI\_Ireduce\_scatter\_block(const void \*sendbuf, void \*recvbuf, 37 int recvcount, MPI\_Datatype datatype, MPI\_Op op, 38 MPI\_Comm comm, MPI\_Request \*request) 39 40int MPI\_Ireduce\_scatter\_block\_c(const void \*sendbuf, void \*recvbuf, 41 MPI\_Count recvcount, MPI\_Datatype datatype, MPI\_Op op, 42MPI\_Comm comm, MPI\_Request \*request) 43 int MPI\_Ireduce\_scatter\_c(const void \*sendbuf, void \*recvbuf, 44const MPI\_Count recvcounts[], MPI\_Datatype datatype, 45MPI\_Op op, MPI\_Comm comm, MPI\_Request \*request) 4647int MPI\_Iscan(const void \*sendbuf, void \*recvbuf, int count, 48

	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)	$\frac{1}{2}$
int MPT Isca	n_c(const void *sendbuf, void *recvbuf, MPI_Count count,	3
1110 HI 1_1508	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	4
	MPI_Request *request)	5
	m i_nequest *request)	6
int MPI_Isca	tter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,	7
	<pre>void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,</pre>	8
	MPI_Comm comm, MPI_Request *request)	9
int MPT Isca	tter_c(const void *sendbuf, MPI_Count sendcount,	10
1110 111 1_1000	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	11
	MPI_Datatype recvtype, int root, MPI_Comm comm,	12
	MPI_Request *request)	13 14
		14
int MPI_Isca	tterv(const void *sendbuf, const int sendcounts[],	16
	<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	17
	<pre>int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,</pre>	18
	MPI_Request *request)	19
int MPI_Isca	tterv_c(const void *sendbuf, const MPI_Count sendcounts[],	20
	<pre>const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	21
	MPI_Count recvcount, MPI_Datatype recvtype, int root,	22
	MPI_Comm comm, MPI_Request *request)	23
int MDT On a	commutative (MDI On an int teammuta)	24
Int MP1_0p_C	commutative(MPI_Op op, int *commute)	25
_	reate(MPI_User_function *user_fn, int commute, MPI_Op *op)	26 27
int MPI_Op_c	reate_c(MPI_User_function_c *user_fn, int commute, MPI_Op *op)	28
int MPI Op f	ree(MPI_Op *op)	29
_		30
int MPI_Redu	ce(const void *sendbuf, void *recvbuf, int count,	31
	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)	32
int MPI_Redu	<pre>uce_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>	33
_	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)	34
		35
int MP1_Redu	<pre>uce_init(const void *sendbuf, void *recvbuf, int count, NDL Determine determine NDL On an int work NDL Commencement</pre>	36
	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,	37
	MPI_Info info, MPI_Request *request)	38
int MPI_Redu	<pre>ice_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>	39
	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,	40
	MPI_Info info, MPI_Request *request)	41
int MDT Dade	ce local (const woid kinbuf woid kinoutbuf int count	42
IIIC MFI_REQU	<pre>ice_local(const void *inbuf, void *inoutbuf, int count, MPI_Datatype datatype, MPI_Op op)</pre>	43
	In I_basasype dasasype, In I_op op)	44
int MPI_Redu	ce_local_c(const void *inbuf, void *inoutbuf, MPI_Count count,	45
	MPI_Datatype datatype, MPI_Op op)	46
int MPT Redu	<pre>scatter(const void *sendbuf, void *recvbuf,</pre>	47
		48

	884	ANNEX A. LANGUAGE BINDINGS SUMMARY
1 2		<pre>const int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>
3 4 5 6	int	<pre>MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf,</pre>
7 8 9 10	int	<pre>MPI_Reduce_scatter_block_c(const void *sendbuf, void *recvbuf,</pre>
11 12 13	int	<pre>MPI_Reduce_scatter_block_init(const void *sendbuf, void *recvbuf,</pre>
14 15 16	int	<pre>MPI_Reduce_scatter_block_init_c(const void *sendbuf, void *recvbuf,</pre>
17 18 19 20	int	<pre>MPI_Reduce_scatter_c(const void *sendbuf, void *recvbuf,</pre>
21 22 23 24	int	<pre>MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf,</pre>
25 26 27	int	<pre>MPI_Reduce_scatter_init_c(const void *sendbuf, void *recvbuf,</pre>
28 29 30	int	<pre>MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>
31 32	int	<pre>MPI_Scan_c(const void *sendbuf, void *recvbuf, MPI_Count count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</pre>
33 34 35 36	int	<pre>MPI_Scan_init(const void *sendbuf, void *recvbuf, int count,</pre>
37 38 39	int	<pre>MPI_Scan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>
40 41 42 43	int	<pre>MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>
44 45 46	int	<pre>MPI_Scatter_c(const void *sendbuf, MPI_Count sendcount,</pre>
47 48	int	MPI_Scatter_init(const void *sendbuf, int sendcount,

	MPI_Datatype sendtype, void *recvbuf, int recvcount,	1
	MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	2
	MPI_Request *request)	3
	······································	4
int MPI_Scat	<pre>ter_init_c(const void *sendbuf, MPI_Count sendcount,</pre>	5
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	6
	MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	
	MPI_Request *request)	7
		8
int MPI_Scat	terv(const void *sendbuf, const int sendcounts[],	9
	const int displs[], MPI_Datatype sendtype, void *recvbuf,	10
	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	11
		12
int MPI_Scat	<pre>terv_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	13
	<pre>const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	14
	MPI_Count recvcount, MPI_Datatype recvtype, int root,	15
	MPI_Comm comm)	
		16
int MPI_Scat	<pre>terv_init(const void *sendbuf, const int sendcounts[],</pre>	17
	<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	18
	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,	19
	MPI_Info info, MPI_Request *request)	20
		21
int MPI_Scat	<pre>terv_init_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	22
	<pre>const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	23
	MPI_Count recvcount, MPI_Datatype recvtype, int root,	24
	MPI_Comm comm, MPI_Info info, MPI_Request *request)	25
		26
		27
A.3.5 Groups	s, Contexts, Communicators, and Caching C Bindings	27
int MPT COMM	_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,	29
	void *attribute_val_in, void *attribute_val_out, int *flag)	30
	Void *attribute_val_II, void *attribute_val_out, IIt *iiag)	
int MPI_COMM	_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval,	31
	void *extra_state, void *attribute_val_in,	32
	void *attribute_val_out, int *flag)	33
		34
int MPI_COMM	_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval,	35
	void *attribute_val, void *extra_state)	36
		37
int MPI_Comm	_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)	38
int MPT Comm	_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)	39
1110 111 1_000	_orodoo(in 1_oommi, in 1_orodp Stoup, in 1_oommi Howoomm)	40
int MPI_Comm	_create_from_group(MPI_Group group, const char *stringtag,	41
	MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm)	42
· · NDT G		43
int MPI_Comm	_create_group(MPI_Comm comm, MPI_Group group, int tag,	
	MPI_Comm *newcomm)	44
int MPT Comm	_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,	45
THO IN T_COUM	MPI_Comm_delete_attr_function *comm_delete_attr_fn,	46
		47
	<pre>int *comm_keyval, void *extra_state)</pre>	48

1	int	MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
2 3	int	MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
4	int	MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
5 6	int	MPI_Comm_free(MPI_Comm *comm)
7	int	MPI_Comm_free_keyval(int *comm_keyval)
8 9 10	int	<pre>MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,</pre>
11	int	<pre>MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)</pre>
12 13	int	MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
14 15	int	MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
16	int	MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
17 18 19	int	<pre>MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info,</pre>
20	int	MPI_Comm_rank(MPI_Comm comm, int *rank)
21 22	int	MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
23	int	MPI_Comm_remote_size(MPI_Comm comm, int *size)
24 25	int	MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
26	int	MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
27 28	int	MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
29	int	MPI_Comm_size(MPI_Comm comm, int *size)
30 31	int	MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
32 33	int	MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
34		MPI_Info info, MPI_Comm *newcomm)
35 36		<pre>MPI_Comm_test_inter(MPI_Comm comm, int *flag) NPI_Comm_test_inter(MPI_Comm comm, int *flag)</pre>
37		<pre>MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)</pre>
38 39	int	<pre>MPI_Group_difference(MPI_Group group1, MPI_Group group2,</pre>
40 41 42	int	<pre>MPI_Group_excl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)</pre>
42	int	MPI_Group_free(MPI_Group *group)
44 45	int	<pre>MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,</pre>
46 47 48	int	<pre>MPI_Group_incl(MPI_Group group, int n, const int ranks[],</pre>

<pre>int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>	1 2
<pre>int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)</pre>	3 4
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],	5 6
MPI_Group *newgroup)	7 8
<pre>int MPI_Group_rank(MPI_Group group, int *rank)</pre>	9
<pre>int MPI_Group_size(MPI_Group group, int *size)</pre>	10 11
<pre>int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[],</pre>	12 13
<pre>int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)</pre>	14 15
<pre>int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,</pre>	16 17
<pre>MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)</pre>	18
	19 20
<pre>int MPI_Intercomm_create_from_groups(MPI_Group local_group,</pre>	21
const char *stringtag, MPI_Info info,	22 23
MPI_Errhandler errhandler, MPI_Comm *newintercomm)	24
<pre>int MPI_Intercomm_merge(MPI_Comm intercomm, int high,</pre>	25 26
<pre>int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval,</pre>	27
<pre>void *extra_state, void *attribute_val_in,</pre>	28 29
<pre>void *attribute_val_out, int *flag)</pre>	30
<pre>int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval,</pre>	31 32
void *attribute_val_out, int *flag)	33
<pre>int MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval,</pre>	34
void *attribute_val, void *extra_state)	35 36
<pre>int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,</pre>	37
<pre>MPI_Type_delete_attr_function *type_delete_attr_fn, int *type_keyval, void *extra_state)</pre>	38 39
	40
<pre>int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)</pre>	41
<pre>int MPI_Type_free_keyval(int *type_keyval)</pre>	42 43
<pre>int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,</pre>	44
	45 46
<pre>int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,</pre>	47
	48

1 2	int	<pre>MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,</pre>
3 4	int	<pre>MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)</pre>
5 6	int	<pre>MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
7 8 9	int	<pre>MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
10 11	int	<pre>MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval,</pre>
12 13 14 15	int	<pre>MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,</pre>
16 17	int	MPI_Win_delete_attr(MPI_Win win, int win_keyval)
18	int	MPI_Win_free_keyval(int *win_keyval)
19 20 21	int	<pre>MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,</pre>
22	int	MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
23 24	int	MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
25 26	int	<pre>MPI_Win_set_name(MPI_Win win, const char *win_name)</pre>
27 28	A.3.	6 Process Topologies C Bindings
29 30	int	<pre>MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])</pre>
31 32	int	<pre>MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],</pre>
33 34	int	<pre>MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],</pre>
35 36 37	int	<pre>MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],</pre>
38	int	<pre>MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)</pre>
39 40 41	int	<pre>MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>
42	int	<pre>MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)</pre>
43 44	int	MPI_Cartdim_get(MPI_Comm comm, int *ndims)
$45 \\ 46$	int	<pre>MPI_Dims_create(int nnodes, int ndims, int dims[])</pre>
40 47 48	int	<pre>MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>

	<pre>const int weights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)</pre>	1 2
		3
int	MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,	4
	<pre>const int sources[], const int sourceweights[], int outdegree,</pre>	5
	<pre>const int destinations[], const int destweights[],</pre>	6
	<pre>MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)</pre>	7
÷+	MDT Dist most heighbour (MDT Comm somm int monindomore int sources[]	8
int	<pre>MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>	9
	<pre>int sourceweights[], int maxoutdegree, int destinations[],</pre>	10
	<pre>int destweights[])</pre>	10
int	MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,	11
	int *outdegree, int *weighted)	
		13
int	<pre>MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],</pre>	14
	const int edges[], int reorder, MPI_Comm *comm_graph)	15
	MDI Graph get (MDI Gamm comm int movinder int movednes int inder[]	16
TUC	<pre>MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],</pre>	17
	<pre>int edges[])</pre>	18
int	MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],	19
	<pre>const int edges[], int *newrank)</pre>	20
		21
int	MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,	22
	<pre>int neighbors[])</pre>	23
int	MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)	24
1110	In 1_draph_nerghbors_count (in 1_comm, int rank, int winerghbors)	25
$\operatorname{int}$	MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)	26
	MDT Traighton all methon (songt word transhuf int condecumt	27
TUC	MPI_Ineighbor_allgather(const void *sendbuf, int sendcount,	28
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	29
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	30
int	MPI_Ineighbor_allgather_c(const void *sendbuf, MPI_Count sendcount,	31
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	32
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	33
		34
int	MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount,	35
	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],</pre>	36
	<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>	37
	MPI_Request *request)	38
in+	MDI Incightor all gathery a congt word trandbuf MDI Count condeaunt	39
IIIC	MPI_Ineighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount,	40
	MPI_Datatype sendtype, void *recvbuf,	
	<pre>const MPI_Count recvcounts[], const MPI_Aint displs[], MDI_Detectory of MDI_Commonstrate (MDI_Detectory of the second state)</pre>	41
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	42
int	MPI_Ineighbor_alltoall(const void *sendbuf, int sendcount,	43
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	44
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	45
		46
int	<pre>MPI_Ineighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,</pre>	47
	<pre>MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,</pre>	48

```
1
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
\mathbf{2}
     int MPI_Ineighbor_alltoallv(const void *sendbuf, const int sendcounts[],
3
                   const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
4
                   const int recvcounts[], const int rdispls[],
5
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
6
\overline{7}
     int MPI_Ineighbor_alltoallv_c(const void *sendbuf,
8
                   const MPI_Count sendcounts[], const MPI_Aint sdispls[],
9
                   MPI_Datatype sendtype, void *recvbuf,
10
                   const MPI_Count recvcounts[], const MPI_Aint rdispls[],
11
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
12
     int MPI_Ineighbor_alltoallw(const void *sendbuf, const int sendcounts[],
13
                   const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
14
                   void *recvbuf, const int recvcounts[],
15
                   const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
16
                   MPI_Comm comm, MPI_Request *request)
17
18
     int MPI_Ineighbor_alltoallw_c(const void *sendbuf,
19
                   const MPI_Count sendcounts[], const MPI_Aint sdispls[],
20
                   const MPI_Datatype sendtypes[], void *recvbuf,
21
                   const MPI_Count recvcounts[], const MPI_Aint rdispls[],
22
                   const MPI_Datatype recvtypes[], MPI_Comm comm,
23
                   MPI_Request *request)
24
     int MPI_Neighbor_allgather(const void *sendbuf, int sendcount,
25
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
26
                   MPI_Datatype recvtype, MPI_Comm comm)
27
28
     int MPI_Neighbor_allgather_c(const void *sendbuf, MPI_Count sendcount,
29
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
30
                   MPI_Datatype recvtype, MPI_Comm comm)
31
     int MPI_Neighbor_allgather_init(const void *sendbuf, int sendcount,
32
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
33
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
34
                   MPI_Request *request)
35
36
     int MPI_Neighbor_allgather_init_c(const void *sendbuf, MPI_Count sendcount,
37
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
38
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
39
                   MPI_Request *request)
40
     int MPI_Neighbor_allgatherv(const void *sendbuf, int sendcount,
41
                   MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
42
                   const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
43
44
     int MPI_Neighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount,
45
                   MPI_Datatype sendtype, void *recvbuf,
46
                   const MPI_Count recvcounts[], const MPI_Aint displs[],
47
                   MPI_Datatype recvtype, MPI_Comm comm)
48
```

ANNEX A. LANGUAGE BINDINGS SUMMARY

890

int	MPI_Neighbor_allgatherv_init(const void *sendbuf, int sendcount,	$\frac{1}{2}$
	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],</pre>	2
	<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MDI_Info_infoMDI_Datatype recvtype, MPI_Comm comm,</pre>	4
	MPI_Info info, MPI_Request *request)	5
int	MPI_Neighbor_allgatherv_init_c(const void *sendbuf,	6
	MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf,	7
	<pre>const MPI_Count recvcounts[], const MPI_Aint displs[],</pre>	8
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	9
	MPI_Request *request)	10
int	MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,	11
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	12
	MPI_Datatype recvtype, MPI_Comm comm)	13
		14
int	MPI_Neighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,	15
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	16
	MPI_Datatype recvtype, MPI_Comm comm)	17
int	MPI_Neighbor_alltoall_init(const void *sendbuf, int sendcount,	18
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	19
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	20
	MPI_Request *request)	21
in+	MPI_Neighbor_alltoall_init_c(const void *sendbuf, MPI_Count sendcount,	22
THC	J J J J J J J J J J J J J J J J J J J	23
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	24
	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	25
	MFI_Request *lequest)	26
int	<pre>MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[],</pre>	27
	<pre>const int sdispls[], MPI_Datatype sendtype, void *recvbuf,</pre>	28
	<pre>const int recvcounts[], const int rdispls[],</pre>	29
	MPI_Datatype recvtype, MPI_Comm comm)	30
in+	MPI_Neighbor_alltoallv_c(const void *sendbuf,	31
1110	const MPI_Count sendcounts[], const MPI_Aint sdispls[],	32
	MPI_Datatype sendtype, void *recvbuf,	33
	const MPI_Count recvcounts[], const MPI_Aint rdispls[],	34
	MPI_Datatype recvtype, MPI_Comm comm)	35 36
		30 37
int	<pre>MPI_Neighbor_alltoallv_init(const void *sendbuf,</pre>	38
	<pre>const int sendcounts[], const int sdispls[],</pre>	39
	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],</pre>	40
	const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm,	41
	MPI_Info info, MPI_Request *request)	42
int	MPI_Neighbor_alltoallv_init_c(const void *sendbuf,	43
	const MPI_Count sendcounts[], const MPI_Aint sdispls[],	44
	MPI_Datatype sendtype, void *recvbuf,	45
	const MPI_Count recvcounts[], const MPI_Aint rdispls[],	46
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	47
	MPI_Request *request)	48
	• •	

```
1
                int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[],
           \mathbf{2}
                              const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
           3
                              void *recvbuf, const int recvcounts[],
           4
                              const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
           5
                              MPI_Comm comm)
           6
                int MPI_Neighbor_alltoallw_c(const void *sendbuf,
           7
                              const MPI_Count sendcounts[], const MPI_Aint sdispls[],
           8
                              const MPI_Datatype sendtypes[], void *recvbuf,
           9
                              const MPI_Count recvcounts[], const MPI_Aint rdispls[],
           10
                              const MPI_Datatype recvtypes[], MPI_Comm comm)
           11
           12
                int MPI_Neighbor_alltoallw_init(const void *sendbuf,
           13
                              const int sendcounts[], const MPI_Aint sdispls[],
           14
                              const MPI_Datatype sendtypes[], void *recvbuf,
           15
                              const int recvcounts[], const MPI_Aint rdispls[],
           16
                              const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
           17
                              MPI_Request *request)
           18
                int MPI_Neighbor_alltoallw_init_c(const void *sendbuf,
           19
                              const MPI_Count sendcounts[], const MPI_Aint sdispls[],
           20
                              const MPI_Datatype sendtypes[], void *recvbuf,
           21
                              const MPI_Count recvcounts[], const MPI_Aint rdispls[],
           22
                              const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
           23
                              MPI_Request *request)
           ^{24}
           25
                int MPI_Topo_test(MPI_Comm comm, int *status)
           26
           27
                A.3.7 MPI Environmental Management C Bindings
           28
           29
#-error!
                double MPI_Wtick(void)
                                          These routine were lost due to
           30
                                          a bug in MAKE-APPLANG
#-PR414
           31
                double MPI_Wtime(void)
#-update2
           32
                int MPI_Add_error_class(int *errorclass)
Resolved in <sup>33</sup>
           34
                int MPI_Add_error_code(int errorclass, int *errorcode)
#-PR431
           35
#-update4
                int MPI_Add_error_string(int errorcode, const char *string)
           36
           37
                int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
           38
                int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
           39
           40
                int MPI_Comm_create_errhandler(
           41
                              MPI_Comm_errhandler_function *comm_errhandler_fn,
           42
                              MPI_Errhandler *errhandler)
           43
                int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
           44
           45
                int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
           46
                int MPI_Errhandler_free(MPI_Errhandler *errhandler)
           47
           48
```

Done

PR431

lin

## A.3. C BINDINGS

1 int MPI\_Error\_class(int errorcode, int \*errorclass) 2 int MPI\_Error\_string(int errorcode, char \*string, int \*resultlen) int MPI\_File\_call\_errhandler(MPI\_File fh, int errorcode) int MPI\_File\_create\_errhandler( MPI\_File\_errhandler\_function \*file\_errhandler\_fn, MPI\_Errhandler \*errhandler) int MPI\_File\_get\_errhandler(MPI\_File file, MPI\_Errhandler \*errhandler) 10 int MPI\_File\_set\_errhandler(MPI\_File file, MPI\_Errhandler errhandler) 11 12int MPI\_Free\_mem(void \*base) 13 int MPI\_Get\_library\_version(char \*version, int \*resultlen) 1415int MPI\_Get\_processor\_name(char \*name, int \*resultlen) 16int MPI\_Get\_version(int \*version, int \*subversion) 1718 int MPI\_Session\_call\_errhandler(MPI\_Session session, int errorcode) 19 int MPI\_Session\_create\_errhandler( 20MPI\_Session\_errhandler\_function \*session\_errhandler\_fn, 21MPI\_Errhandler \*errhandler) 22 23int MPI\_Session\_get\_errhandler(MPI\_Session session,  $^{24}$ MPI\_Errhandler \*errhandler) 25int MPI\_Session\_set\_errhandler(MPI\_Session session, 26MPI\_Errhandler errhandler) 2728 int MPI\_Win\_call\_errhandler(MPI\_Win win, int errorcode) 29 int MPI\_Win\_create\_errhandler( 30 31MPI\_Win\_errhandler\_function \*win\_errhandler\_fn, MPI\_Errhandler \*errhandler) 32 33 int MPI\_Win\_get\_errhandler(MPI\_Win win, MPI\_Errhandler \*errhandler) 34 35 int MPI\_Win\_set\_errhandler(MPI\_Win win, MPI\_Errhandler errhandler) 36 37 The Info Object C Bindings A.3.8 38 39 int MPI\_Info\_create(MPI\_Info \*info) 40 int MPI\_Info\_create\_env(int argc, char argv[], MPI\_Info \*info) 41 Deprecated function 42int MPI\_Info\_delete(MPI\_Info info, const char \*key) descriptions must be 43 moved from the int MPI\_Info\_dup(MPI\_Info info, MPI\_Info \*newinfo) 44original chapter to the 45Deprecated Interfaces int MPI\_Info\_free(MPI\_Info \*info) 46chapter !!! int MPI\_Info\_get(MPI\_Info info, const char \*key, int valuelen, char \*value, #-error int \*flag) #-364 Done

Unofficial Draft for Comment Only

in PR443

```
894
                                               ANNEX A. LANGUAGE BINDINGS SUMMARY
       1
            int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
       \mathbf{2}
            int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
       3
       4
            int MPI_Info_get_string(MPI_Info info, const char *key, int *buflen,
       5
                           char *value, int *flag)
       6
#-error
            int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
#-364
                           int *flag)
                                                                                    Deprecated function
                                                                                    descriptions must be
       9
            int MPI_Info_set(MPI_Info info, const char *key, const char *value)
                                                                                    moved from the
       10
PR443
                                                                                    original chapter to the
       11
                                                                                    Deprecated Interfaces
            A.3.9 Process Creation and Management C Bindings
       12
                                                                                    chapter !!!
       13
            int MPI_Abort(MPI_Comm comm, int errorcode)
       14
            int MPI_Close_port(const char *port_name)
       15
       16
            int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
       17
                          MPI_Comm comm, MPI_Comm *newcomm)
       18
       19
            int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
                          MPI_Comm comm, MPI_Comm *newcomm)
       20
       21
            int MPI_Comm_disconnect(MPI_Comm *comm)
       22
       23
            int MPI_Comm_get_parent(MPI_Comm *parent)
       ^{24}
            int MPI_Comm_join(int fd, MPI_Comm *intercomm)
       25
       26
            int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
       27
                          MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
       28
                           int array_of_errcodes[])
       29
            int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
       30
                           char **array_of_argv[], const int array_of_maxprocs[],
       31
                           const MPI_Info array_of_info[], int root, MPI_Comm comm,
       32
                          MPI_Comm *intercomm, int array_of_errcodes[])
       33
       34
            int MPI_Finalize(void)
       35
            int MPI_Finalized(int *flag)
       36
       37
            int MPI_Init(int *argc, char ***argv)
       38
            int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
       39
       40
            int MPI_Initialized(int *flag)
       41
            int MPI_Is_thread_main(int *flag)
       42
       43
            int MPI_Lookup_name(const char *service_name, MPI_Info info,
       44
                           char *port_name)
       45
            int MPI_Open_port(MPI_Info info, char *port_name)
       46
       47
       48
```

Done

lin

	ame(const char *service_name, MPI_Info info, st char *port_name)	12
int MPI_Query_thr	ead(int *provided)	$\frac{3}{4}$
int MPI_Session_f	inalize(MPI_Session *session)	5
int MPI_Session_g	et_info(MPI_Session session, MPI_Info *info_used)	6 7
	et_nth_pset(MPI_Session session, MPI_Info info, int n,	8
0	<pre>*pset_len, char *pset_name)</pre>	9
•	et_num_psets(MPI_Session session, MPI_Info info, *npset_names)	10 11
	<pre>pset_info(MPI_Session session, const char *pset_name,</pre>	12 13
-	Info *info)	14
int MPI_Session_i	nit(MPI_Info info, MPI_Errhandler errhandler,	15 16
	Session *session)	17
int MPI_Unpublish	name(const char *service_name, MPI_Info info,	18 19
cons	st char *port_name)	19 20
		21
A.3.10 One-Sided	Communications C Bindings	22
int MPI_Accumulat	e(const void *origin_addr, int origin_count,	23 24
	Datatype origin_datatype, int target_rank,	25
	_Aint target_disp, int target_count, _Datatype target_datatype, MPI_Op op, MPI_Win win)	26
		27 28
	<pre>e_c(const void *origin_addr, MPI_Count origin_count, _Datatype origin_datatype, int target_rank,</pre>	29
	_Aint target_disp, MPI_Count target_count,	30
MPI_	_Datatype target_datatype, MPI_Op op, MPI_Win win)	31
int MPI_Compare_a	nd_swap(const void *origin_addr, const void *compare_addr,	32 33
	d *result_addr, MPI_Datatype datatype, int target_rank,	34
MP1_	_Aint target_disp, MPI_Win win)	35
	_op(const void *origin_addr, void *result_addr,	36 37
	_Datatype datatype, int target_rank, MPI_Aint target_disp, _Op op, MPI_Win win)	38
		39
	<pre>*origin_addr, int origin_count, _Datatype origin_datatype, int target_rank,</pre>	40
	Aint target_disp, int target_count,	41 42
	Datatype target_datatype, MPI_Win win)	43
int MPI_Get_accum	ulate(const void *origin_addr, int origin_count,	44
MPI_	_Datatype origin_datatype, void *result_addr,	45
	result_count, MPI_Datatype result_datatype,	46 47
int	<pre>target_rank, MPI_Aint target_disp, int target_count,</pre>	48

	896	ANNEX A. LANGUAGE BINDINGS SUMMARY
1		MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
2 3 4 5 6 7	int MPI_Get_	accumulate_c(const void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, void *result_addr, MPI_Count result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, MPI_Count target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
#-364 8 #-error! 10 11 12	int MPI_Get_	c(void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, MPI_Count target_count, MPI_Datatype target_datatype, MPI_Win win) Due to alphabetic rules, MPI_Get_c does not follow directly after MPI_Get.
13 14 15 16	int MPI_Put(	<pre>const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)</pre> Should we fix this? And can we fix this?
17 18 19 20 21 22 23 24 25 26	int MPI_Put_	c(const void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, MPI_Count target_count, MPI_Datatype target_datatype, MPI_Win win)
	int MPI_Racc	<pre>sumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, MPI_Request *request)</pre>
27 28 29 30 31 32	int MPI_Racc	<pre>wumulate_c(const void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, MPI_Count target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, MPI_Request *request)</pre>
33 34 35 36 37	int MPI_Rget	<pre>(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)</pre>
38 39 40 41 42 43 44	int MPI_Rget	<pre>_accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, void *result_addr, int result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, MPI_Request *request)</pre>
44 45 46 47 48	int MPI_Rget	<pre>_accumulate_c(const void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, void *result_addr, MPI_Count result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, MPI_Count target_count,</pre>

<pre>MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, MPI_Request *request)</pre>	1 2
<pre>int MPI_Rget_c(void *origin_addr, MPI_Count origin_count,</pre>	3 4 5 6 7 8
<pre>int MPI_Rput(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)</pre>	9 10 11 12 13 14
<pre>int MPI_Rput_c(const void *origin_addr, MPI_Count origin_count,</pre>	15 16 17 18 19
int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)	20 21 22
<pre>int MPI_Win_allocate_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,</pre>	23 24
<pre>int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,</pre>	25 26 27
<pre>int MPI_Win_allocate_shared_c(MPI_Aint size, MPI_Aint disp_unit,</pre>	28 29
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)	30 31
<pre>int MPI_Win_complete(MPI_Win win)</pre>	32
<pre>int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,</pre>	33 34 35
int MPI_Win_create_c(void *base, MPI_Aint size, MPI_Aint disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)	36 37
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)	38 39
int MPI_Win_detach(MPI_Win win, const void *base)	40
int MPI_Win_fence(int assert, MPI_Win win)	41 42
int MPI_Win_flush(int rank, MPI_Win win)	42 43
int MPI_Win_flush_all(MPI_Win win)	44
int MPI_Win_flush_local(int rank, MPI_Win win)	45 46
<pre>int MPI_Win_flush_local_all(MPI_Win win)</pre>	47
	48

1	int	MPI_Win_free(MPI_Win *win)
2 3	int	MPI_Win_get_group(MPI_Win win, MPI_Group *group)
4	int	MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
5 6	int	MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
7	int	MPI_Win_lock_all(int assert, MPI_Win win)
8 9	int	MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
10 11	int	MPI_Win_set_info(MPI_Win win, MPI_Info info)
12 13	int	<pre>MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,</pre>
14 15 16	int	<pre>MPI_Win_shared_query_c(MPI_Win win, int rank, MPI_Aint *size, MPI_Aint *disp_unit, void *baseptr)</pre>
17	int	MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
18 19	int	MPI_Win_sync(MPI_Win win)
20	int	MPI_Win_test(MPI_Win win, int *flag)
21 22	int	MPI_Win_unlock(int rank, MPI_Win win)
23	int	MPI_Win_unlock_all(MPI_Win win)
24 25	int	MPI_Win_wait(MPI_Win win)
26		
27 28	A.3.	11 External Interfaces C Bindings
29	int	MPI_Grequest_complete(MPI_Request request)
30 31	int	MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
32		<pre>MPI_Grequest_free_function *free_fn, MPI_Grequest_cancel_function *cancel_fn, void *extra_state,</pre>
33		MPI_Request *request)
34 35	int	MPI_Status_set_cancelled(MPI_Status *status, int flag)
36	int	MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
37 38		int count)
39	int	MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
40 41		MPI_Count count)
42	• •	
43	A.3.	12 I/O C Bindings
44 45	int	<pre>MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int count,</pre>
46 47 48	int	<pre>MPI_CONVERSION_FN_NULL_c(void *userbuf, MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position,</pre>

void *extra_state)	1
<pre>int MPI_File_close(MPI_File *fh)</pre>	2 3
<pre>int MPI_File_delete(const char *filename, MPI_Info info)</pre>	4
int MPI_File_get_amode(MPI_File fh, int *amode)	5
ů –	6
<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>	7 8
<pre>int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,</pre>	9
MPI_Offset *disp)	10
<pre>int MPI_File_get_group(MPI_File fh, MPI_Group *group)</pre>	11
<pre>int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)</pre>	12 13
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)	14
	This is okay because
<pre>int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)</pre>	embiggening is needed for
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)	datatype usage in
<pre>int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,</pre>	MPI_Offset based MPI-IO and MPI_Offset can be
MPI_Aint *extent)	larger than MPI Aint.
int MPI_File_get_type_extent_c(MPI_File fh, MPI_Datatype datatype	. <sup>21</sup> #-364
MPI_Count *extent)	22
int MDI File get wiew(MDI File fb MDI Offact which MDI Deteture	23 *etype 24
<pre>int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype</pre>	*etype, <sup>24</sup> 25
	26
int MPI_File_iread(MPI_File fh, void *buf, int count,	27
MPI_Datatype datatype, MPI_Request *request)	28
<pre>int MPI_File_iread_all(MPI_File fh, void *buf, int count,</pre>	29
MPI_Datatype datatype, MPI_Request *request)	30
int MPI_File_iread_all_c(MPI_File fh, void *buf, MPI_Count count,	31 32
MPI_Datatype datatype, MPI_Request *request)	33
<pre>int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, :</pre>	
MPI_Datatype datatype, MPI_Request *request)	35
	36
<pre>int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *bu int count, MPI_Datatype datatype, MPI_Request *reque</pre>	01
	00
int MPI_File_iread_at_all_c(MPI_File fh, MPI_Offset offset, void	10
MPI_Count count, MPI_Datatype datatype, MPI_Request	*request) <sup>10</sup> 41
<pre>int MPI_File_iread_at_c(MPI_File fh, MPI_Offset offset, void *buf</pre>	
<pre>MPI_Count count, MPI_Datatype datatype, MPI_Request</pre>	*request) $_{43}$
<pre>int MPI_File_iread_c(MPI_File fh, void *buf, MPI_Count count,</pre>	44
MPI_Datatype datatype, MPI_Request *request)	45
<pre>int MPI_File_iread_shared(MPI_File fh, void *buf, int count,</pre>	46
MPI_Datatype datatype, MPI_Request *request)	47 48

1 int MPI\_File\_iread\_shared\_c(MPI\_File fh, void \*buf, MPI\_Count count,  $\mathbf{2}$ MPI\_Datatype datatype, MPI\_Request \*request) 3 int MPI\_File\_iwrite(MPI\_File fh, const void \*buf, int count, 4 MPI\_Datatype datatype, MPI\_Request \*request) 56 int MPI\_File\_iwrite\_all(MPI\_File fh, const void \*buf, int count, 7MPI\_Datatype datatype, MPI\_Request \*request) 8 int MPI\_File\_iwrite\_all\_c(MPI\_File fh, const void \*buf, MPI\_Count count, 9 MPI\_Datatype datatype, MPI\_Request \*request) 10  $^{11}$ int MPI\_File\_iwrite\_at(MPI\_File fh, MPI\_Offset offset, const void \*buf, 12int count, MPI\_Datatype datatype, MPI\_Request \*request) 13int MPI\_File\_iwrite\_at\_all(MPI\_File fh, MPI\_Offset offset, const void \*buf, 14int count, MPI\_Datatype datatype, MPI\_Request \*request) 1516int MPI\_File\_iwrite\_at\_all\_c(MPI\_File fh, MPI\_Offset offset, 17const void \*buf, MPI\_Count count, MPI\_Datatype datatype,  $^{18}$ MPI\_Request \*request) 19int MPI\_File\_iwrite\_at\_c(MPI\_File fh, MPI\_Offset offset, const void \*buf, 20MPI\_Count count, MPI\_Datatype datatype, MPI\_Request \*request) 2122int MPI\_File\_iwrite\_c(MPI\_File fh, const void \*buf, MPI\_Count count, 23MPI\_Datatype datatype, MPI\_Request \*request)  $^{24}$ int MPI\_File\_iwrite\_shared(MPI\_File fh, const void \*buf, int count, 25MPI\_Datatype datatype, MPI\_Request \*request) 2627int MPI\_File\_iwrite\_shared\_c(MPI\_File fh, const void \*buf, MPI\_Count count, 28MPI\_Datatype datatype, MPI\_Request \*request) 29int MPI\_File\_open(MPI\_Comm comm, const char \*filename, int amode, 30 MPI\_Info info, MPI\_File \*fh)  $^{31}$ 32int MPI\_File\_preallocate(MPI\_File fh, MPI\_Offset size) 33 34int MPI\_File\_read(MPI\_File fh, void \*buf, int count, MPI\_Datatype datatype, MPI\_Status \*status) 3536 int MPI\_File\_read\_all(MPI\_File fh, void \*buf, int count, 37 MPI\_Datatype datatype, MPI\_Status \*status) 38 39int MPI\_File\_read\_all\_begin(MPI\_File fh, void \*buf, int count, 40MPI\_Datatype datatype) 41 int MPI\_File\_read\_all\_begin\_c(MPI\_File fh, void \*buf, MPI\_Count count, 42MPI\_Datatype datatype) 43 44int MPI\_File\_read\_all\_c(MPI\_File fh, void \*buf, MPI\_Count count, 45MPI\_Datatype datatype, MPI\_Status \*status) 46int MPI\_File\_read\_all\_end(MPI\_File fh, void \*buf, MPI\_Status \*status) 4748

int	<pre>MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	1 2
int	<pre>MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	3 4 5
int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>	6 7 8
int	<pre>MPI_File_read_at_all_begin_c(MPI_File fh, MPI_Offset offset, void *buf,</pre>	9 10
int	<pre>MPI_File_read_at_all_c(MPI_File fh, MPI_Offset offset, void *buf,</pre>	11 12 13
int	<pre>MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	14
int	<pre>MPI_File_read_at_c(MPI_File fh, MPI_Offset offset, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	15 16 17
int	<pre>MPI_File_read_c(MPI_File fh, void *buf, MPI_Count count,</pre>	18 19
int	<pre>MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	20 21 22
int	<pre>MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	22 23 24
int	<pre>MPI_File_read_ordered_begin_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype)</pre>	25 26 27
int	<pre>MPI_File_read_ordered_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	28 29
int	MPI_File_read_ordered_end(MPI_File fh, void *buf, 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
int	<pre>MPI_File_read_shared_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	34 35 36
int	MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)	37
int	MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)	38 39
int	MPI_File_set_atomicity(MPI_File fh, int flag)	40
int	MPI_File_set_info(MPI_File fh, MPI_Info info)	41 42
int	MPI_File_set_size(MPI_File fh, MPI_Offset size)	43
int	MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,	$44 \\ 45$
	MPI_Datatype filetype, const char *datarep, MPI_Info info)	46
int	MPI_File_sync(MPI_File fh)	47 48

1 2	int	<pre>MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>
3 4 5	int	<pre>MPI_File_write_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>
6 7	int	<pre>MPI_File_write_all_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)</pre>
8 9 10	int	<pre>MPI_File_write_all_begin_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype)</pre>
11 12 13	int	<pre>MPI_File_write_all_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>
14 15	int	<pre>MPI_File_write_all_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>
16 17 18	int	<pre>MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
19 20	int	<pre>MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
21 22 23	int	<pre>MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,</pre>
23 24 25	int	<pre>MPI_File_write_at_all_begin_c(MPI_File fh, MPI_Offset offset,</pre>
26 27 28 29	int	<pre>MPI_File_write_at_all_c(MPI_File fh, MPI_Offset offset,</pre>
30 31	int	<pre>MPI_File_write_at_all_end(MPI_File fh, const void *buf, MPI_Status *status)</pre>
32 33 34	int	<pre>MPI_File_write_at_c(MPI_File fh, MPI_Offset offset, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>
35 36	int	<pre>MPI_File_write_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>
37 38 39	int	<pre>MPI_File_write_ordered(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>
40 41	int	<pre>MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count, MPI_Datatype datatype)</pre>
42 43 44	int	<pre>MPI_File_write_ordered_begin_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype)</pre>
45 46	int	<pre>MPI_File_write_ordered_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>
47 48	int	MPI_File_write_ordered_end(MPI_File fh, const void *buf,

		1
MPI_Status *status)		2
<pre>int MPI_File_write_shared(MPI_File fh, const voi</pre>		3
		4 5
<pre>int MPI_File_write_shared_c(MPI_File fh, const v</pre>		6
int MPI_Register_datarep(const char *datarep,		7
MPI_Datarep_conversion_function *re	ead_conversion_fn,	8
MPI_Datarep_conversion_function *wr		10
<pre>MPI_Datarep_extent_function *dtype_ void *extra_state)</pre>	file_extent_in,	11
<pre>int MPI_Register_datarep_c(const char *datarep,</pre>		12 13
MPI_Datarep_conversion_function_c *	read_conversion_fn,	14
MPI_Datarep_conversion_function_c *		15 16
<pre>MPI_Datarep_extent_function *dtype_ void *extra_state)</pre>	_file_extent_fn,	17
		18
A.3.13 Language Bindings C Bindings		19 20
int MPI_Status_f082f(const MPI_F08_status *f08_s	tatus MPI Fint *f status)	<sup>21</sup> #-error!
		<sup>22</sup> <b>#-PR414</b>
<pre>int MPI_Status_f2f08(const MPI_Fint *f_status, M</pre>		<sup>23</sup> <sub>24</sub> #-update2
<pre>int MPI_Type_create_f90_complex(int p, int r, MP</pre>		25 Done
<pre>int MPI_Type_create_f90_integer(int r, MPI_Datat</pre>	ype *newtype)	26 PR431
<pre>int MPI_Type_create_f90_real(int p, int r, MPI_D</pre>	atatype *newtype)	28
<pre>int MPI_Type_match_size(int typeclass, int size,</pre>	MPI_Datatype *datatype)	29
MPI_Fint MPI_Comm_c2f(MPI_Comm comm)	These routine were lost due to	<sup>30</sup> <sub>31</sub> #-error!
MPI_Comm_MPI_Comm_f2c(MPI_Fint comm)	a bug in MAKE-APPLANG	32 <b>#-PR414</b>
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errha	ndler)	<ul> <li><sup>33</sup> #-update2</li> <li><sup>34</sup> Done</li> </ul>
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errha	ndler)	35 in
MPI_Fint MPI_File_c2f(MPI_File file)	36 97	
MPI_File MPI_File_f2c(MPI_Fint file)		38
MPI_Fint MPI_Group_c2f(MPI_Group group)		39 40
MPI_Group MPI_Group_f2c(MPI_Fint group)	41	
MPI_Fint MPI_Info_c2f(MPI_Info info)	42 43	
MPI_Info MPI_Info_f2c(MPI_Fint info)	44	
		45
MPI_Fint MPI_Message_c2f(MPI_Message message)		46 47
MPI_Message MPI_Message_f2c(MPI_Fint message)		48

```
#-error!
               MPI_Fint MPI_Op_c2f(MPI_Op op)
#-PR414
               MPI_Op MPI_Op_f2c(MPI_Fint op)
#-update2
               MPI_Fint MPI_Request_c2f(MPI_Request request)
               MPI_Request MPI_Request_f2c(MPI_Fint request)
          6
 PR431
               MPI_Fint MPI_Session_c2f(MPI_Session session)
               MPI_Session MPI_Session_f2c(MPI_Fint session)
          10
               int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)
               int MPI_Status_c2f08(const MPI_Status *c_status,
          12
                             MPI_F08_status *f08_status)
          13
          14
               int MPI_Status_f082c(const MPI_F08_status *f08_status,
          15
                             MPI_Status *c_status)
          16
          17
               int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)
          18
               MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
          19
          20
               MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
          21
               MPI_Fint MPI_Win_c2f(MPI_Win win)
          22
          23
               MPI_Win MPI_Win_f2c(MPI_Fint win)
          ^{24}
          25
               A.3.14 Tools / Profiling Interface C Bindings
          26
          27
               int MPI_Pcontrol(const int level, ...)
          28
          29
               A.3.15 Tools / MPI Tool Information Interface C Bindings
          30
          ^{31}
               int MPI_T_category_changed(int *stamp)
          32
               int MPI_T_category_get_categories(int cat_index, int len, int indices[])
          33
          34
               int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
          35
          36
               int MPI_T_category_get_events(int cat_index, int len, int indices[])
          37
               int MPI_T_category_get_index(const char *name, int *cat_index)
          38
          39
               int MPI_T_category_get_info(int cat_index, char *name, int *name_len,
          40
                             char *desc, int *desc_len, int *num_cvars, int *num_pvars,
          41
                             int *num_categories)
          42
               int MPI_T_category_get_num(int *num_cat)
          43
          44
               int MPI_T_category_get_num_events(int cat_index, int *num_events)
          45
               int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
          46
          47
               int MPI_T_cvar_get_index(const char *name, int *cvar_index)
          48
```

904

Done lin

<pre>int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len,</pre>	1
int *verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype,	2
char *desc, int *desc_len, int *bind, int *scope)	3
int MPI_T_cvar_get_num(int *num_cvar)	4 5
	6
<pre>int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,</pre>	7
MF1_1_CVar_manule *manule, int *count)	8
<pre>int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)</pre>	9
<pre>int MPI_T_cvar_read(MPI_T_cvar_handle handle, void *buf)</pre>	10 11
int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void *buf)	12
<pre>int MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name,</pre>	13
int *name_len)	14
	15
<pre>int MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value,</pre>	16
Char *name, int *name_ien)	17
<pre>int MPI_T_event_callback_get_info(</pre>	18 19
MPI_T_event_registration event_registration,	20
<pre>MPI_T_cb_safety cb_safety, MPI_Info *info_used)</pre>	20
<pre>int MPI_T_event_callback_set_info(</pre>	22
MPI_T_event_registration event_registration,	23
MPI_T_cb_safety cb_safety, MPI_Info info)	24
int MDI T event conv(MDI T event instance event instance, word thuffer)	25
<pre>int MPI_T_event_copy(MPI_T_event_instance event_instance, void *buffer)</pre>	26
<pre>int MPI_T_event_get_index(const char *name, int *event_index)</pre>	27
<pre>int MPI_T_event_get_info(int event_index, char *name, int *name_len,</pre>	28
<pre>int *verbosity, MPI_Datatype array_of_datatypes[],</pre>	29
MPI_Aint array_of_displacements[], int *num_elements,	30
MPI_T_enum *enumtype, MPI_Info *info, char *desc,	31 32
<pre>int *desc_len, int *bind)</pre>	32
<pre>int MPI_T_event_get_num(int *num_events)</pre>	34
	35
<pre>int MPI_T_event_get_source(MPI_T_event_instance event_instance,</pre>	36
int *source_index)	37
<pre>int MPI_T_event_get_timestamp(MPI_T_event_instance event_instance,</pre>	38
MPI_Count *event_timestamp)	39
int MDI T event handle alloc(int event index word tabi handle	40
<pre>int MPI_T_event_handle_alloc(int event_index, void *obj_handle,</pre>	41
	42
<pre>int MPI_T_event_handle_free(MPI_T_event_registration event_registration,</pre>	43
void *user_data,	44 45
MPI_T_event_free_cb_function free_cb_function)	45 46
<pre>int MPI_T_event_handle_get_info(</pre>	40
MPI_T_event_registration event_registration,	48

1		MPI_Info *info_used)
2 3 4	int	<pre>MPI_T_event_handle_set_info(     MPI_T_event_registration event_registration, MPI_Info info)</pre>
5 6	int	<pre>MPI_T_event_read(MPI_T_event_instance event_instance,</pre>
7 8 9 10 11	int	<pre>MPI_T_event_register_callback(     MPI_T_event_registration event_registration,     MPI_T_cb_safety cb_safety, MPI_Info info, void *user_data,     MPI_T_event_cb_function event_cb_function)</pre>
12 13 14	int	<pre>MPI_T_event_set_dropped_handler(     MPI_T_event_registration event_registration,     MPI_T_event_dropped_cb_function dropped_cb_function)</pre>
15 16	int	MPI_T_finalize(void)
17 18	int	<pre>MPI_T_init_thread(int required, int *provided)</pre>
19	int	<pre>MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)</pre>
20 21 22 23 24	int	<pre>MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,</pre>
25	int	MPI_T_pvar_get_num(int *num_pvar)
26 27 28	int	<pre>MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index, void *obj_handle, MPI_T_pvar_handle *handle, int *count)</pre>
29 30	int	<pre>MPI_T_pvar_handle_free(MPI_T_pvar_session session,</pre>
31 32 33	int	<pre>MPI_T_pvar_read(MPI_T_pvar_session session, MPI_T_pvar_handle handle,</pre>
34 35	int	<pre>MPI_T_pvar_readreset(MPI_T_pvar_session session,</pre>
36 37	int	<pre>MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)</pre>
38	int	<pre>MPI_T_pvar_session_create(MPI_T_pvar_session *session)</pre>
39 40	int	MPI_T_pvar_session_free(MPI_T_pvar_session *session)
41	int	<pre>MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)</pre>
42 43	int	<pre>MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)</pre>
44 45 46	int	<pre>MPI_T_pvar_write(MPI_T_pvar_session session, MPI_T_pvar_handle handle,</pre>
46 47 48	int	<pre>MPI_T_source_get_info(int source_index, char *name, int *name_len,</pre>

MPI\_Count \*ticks\_per\_second, MPI\_Count \*max\_ticks,  $\mathbf{2}$ MPI\_Info \*info) int MPI\_T\_source\_get\_num(int \*num\_sources) int MPI\_T\_source\_get\_timestamp(int source\_index, MPI\_Count \*timestamp) A.3.16 Deprecated C Bindings a int MPI\_Attr\_delete(MPI\_Comm comm, int keyval) int MPI\_Attr\_get(MPI\_Comm comm, int keyval, void \*attribute\_val, int \*flag) int MPI\_Attr\_put(MPI\_Comm comm, int keyval, void \*attribute\_val) int MPI\_DUP\_FN(MPI\_Comm oldcomm, int keyval, void \*extra\_state, void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag) int MPI\_Info\_get(MPI\_Info info, const char \*key, int valuelen, char \*value, int \*flag) int MPI\_Info\_get\_valuelen(MPI\_Info info, const char \*key, int \*valuelen, int \*flag) int MPI\_Keyval\_create(MPI\_Copy\_function \*copy\_fn, MPI\_Delete\_function \*delete\_fn, int \*keyval, void \*extra\_state) int MPI\_Keyval\_free(int \*keyval) int MPI\_NULL\_COPY\_FN(MPI\_Comm oldcomm, int keyval, void \*extra\_state, void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag) int MPI\_NULL\_DELETE\_FN(MPI\_Comm comm, int keyval, void \*attribute\_val, void \*extra\_state) 

```
Fortran 2008 Bindings with the mpi_f08 Module
1
     A.4
\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(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         INTEGER, INTENT(IN) :: dest, tag
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
12
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
13
         INTEGER, INTENT(IN) :: count, dest, tag
14
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
19
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         INTEGER, INTENT(IN) :: dest, tag
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
28
         INTEGER, INTENT(IN) :: count, dest, tag
29
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
31
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         TYPE(MPI_Request), INTENT(OUT) :: request
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Buffer_attach(buffer, size, ierror)
35
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: size
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Buffer_attach(buffer, size, ierror)
40
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
41
         INTEGER, INTENT(IN) :: size
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
     MPI_Buffer_detach(buffer_addr, size, ierror)
44
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
45
         TYPE(C_PTR), INTENT(OUT) :: buffer_addr
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

## 908

1 MPI\_Buffer\_detach(buffer\_addr, size, ierror) 2 USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR TYPE(C\_PTR), INTENT(OUT) :: buffer\_addr INTEGER, INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror 5 6 MPI\_Cancel(request, ierror) TYPE(MPI\_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 MPI\_Get\_count(status, datatype, count, ierror) 11 TYPE(MPI\_Status), INTENT(IN) :: status TYPE(MPI\_Datatype), INTENT(IN) :: datatype 1213 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: count 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 15MPI\_Get\_count(status, datatype, count, ierror) 16TYPE(MPI\_Status), INTENT(IN) :: status 17 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 18 INTEGER, INTENT(OUT) :: count 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI\_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) 22 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 23INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 24TYPE(MPI\_Datatype), INTENT(IN) :: datatype 25INTEGER, INTENT(IN) :: dest, tag 26TYPE(MPI\_Comm), INTENT(IN) :: comm 27TYPE(MPI\_Request), INTENT(OUT) :: request 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 MPI\_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) 30 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 31INTEGER, INTENT(IN) :: count, dest, 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\_Improbe(source, tag, comm, flag, message, status, ierror) 38 INTEGER, INTENT(IN) :: source, tag 39 TYPE(MPI\_Comm), INTENT(IN) :: comm 40 LOGICAL, INTENT(OUT) :: flag 41 TYPE(MPI\_Message), INTENT(OUT) :: message 42TYPE(MPI\_Status) :: status 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44MPI\_Imrecv(buf, count, datatype, message, request, ierror) 45TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 46INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 47TYPE(MPI\_Datatype), INTENT(IN) :: datatype 48

```
1
         TYPE(MPI_Message), INTENT(INOUT) :: message
2
         TYPE(MPI_Request), INTENT(OUT) :: request
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Imrecv(buf, count, datatype, message, request, ierror)
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
6
         INTEGER, INTENT(IN) :: count
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         TYPE(MPI_Message), INTENT(INOUT) :: message
9
         TYPE(MPI_Request), INTENT(OUT) :: request
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Iprobe(source, tag, comm, flag, status, ierror)
13
         INTEGER, INTENT(IN) :: source, tag
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         LOGICAL, INTENT(OUT) :: flag
16
         TYPE(MPI_Status) :: status
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
19
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         INTEGER, INTENT(IN) :: source, tag
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
28
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
29
         INTEGER, INTENT(IN) :: count, source, 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_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
35
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         INTEGER, INTENT(IN) :: dest, tag
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
44
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
45
         INTEGER, INTENT(IN) :: count, dest, tag
46
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)	3 4
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	5
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	6
TYPE(MPI_Datatype), INTENT(IN) :: datatype	7
INTEGER, INTENT(IN) :: dest, tag	8
TYPE(MPI_Comm), INTENT(IN) :: comm	9
TYPE(MPI_Request), INTENT(OUT) :: request	10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	11
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)	12
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	13
INTEGER, INTENT(IN) :: count, dest, tag	14
TYPE(MPI_Datatype), INTENT(IN) :: datatype	15
TYPE(MPI_Comm), INTENT(IN) :: comm	16
TYPE(MPI_Request), INTENT(OUT) :: request	17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18 19
MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,	20
recvcount, recvtype, source, recvtag, comm, request, ierror)	20
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	22
INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,	23
recvtag	24
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	25
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	26
TYPE(MPI_Comm), INTENT(IN) :: comm	27
TYPE(MPI_Request), INTENT(OUT) :: request	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,	30
comm, request, ierror)	31
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	32
INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag	33
TYPE(MPI_Datatype), INTENT(IN) :: datatype	34
TYPE(MPI_Comm), INTENT(IN) :: comm	35 36
TYPE(MPI_Request), INTENT(OUT) :: request	30 37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)	39
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	40
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	41
TYPE(MPI_Datatype), INTENT(IN) :: datatype	42
INTEGER, INTENT(IN) :: dest, tag	43
TYPE(MPI_Comm), INTENT(IN) :: comm	44
TYPE(MPI_Request), INTENT(OUT) :: request	45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46
MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)	47
	48

```
1
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
2
         INTEGER, INTENT(IN) :: count, dest, tag
3
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Mprobe(source, tag, comm, message, status, ierror)
8
         INTEGER, INTENT(IN) :: source, tag
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Message), INTENT(OUT) :: message
11
         TYPE(MPI_Status) :: status
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Mrecv(buf, count, datatype, message, status, ierror)
15
         TYPE(*), DIMENSION(..) :: buf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Message), INTENT(INOUT) :: message
19
         TYPE(MPI_Status) :: status
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Mrecv(buf, count, datatype, message, status, ierror)
22
         TYPE(*), DIMENSION(..) :: buf
23
         INTEGER, INTENT(IN) :: count
24
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
         TYPE(MPI_Message), INTENT(INOUT) :: message
26
         TYPE(MPI_Status) :: status
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_Probe(source, tag, comm, status, ierror)
30
         INTEGER, INTENT(IN) :: source, tag
31
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         TYPE(MPI_Status) :: status
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
35
         TYPE(*), DIMENSION(..) :: buf
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         INTEGER, INTENT(IN) :: source, tag
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         TYPE(MPI_Status) :: status
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
44
         TYPE(*), DIMENSION(..) :: buf
45
         INTEGER, INTENT(IN) :: count, source, tag
46
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

1 TYPE(MPI\_Status) :: status 2 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_Recv\_init(buf, count, datatype, source, tag, comm, request, ierror) TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 6 TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: source, tag TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Request), INTENT(OUT) :: request 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 MPI\_Recv\_init(buf, count, datatype, source, tag, comm, request, ierror) 1213 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 14INTEGER, INTENT(IN) :: count, source, tag 15TYPE(MPI\_Datatype), INTENT(IN) :: datatype 16TYPE(MPI\_Comm), INTENT(IN) :: comm 17 TYPE(MPI\_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI\_Request\_free(request, ierror) 20TYPE(MPI\_Request), INTENT(INOUT) :: request 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI\_Request\_get\_status(request, flag, status, ierror) 24TYPE(MPI\_Request), INTENT(IN) :: request 25LOGICAL, INTENT(OUT) :: flag 26TYPE(MPI\_Status) :: status 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28MPI\_Rsend(buf, count, datatype, dest, tag, comm, ierror) 29 TYPE(\*), DIMENSION(..), INTENT(IN) :: buf 30 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 31TYPE(MPI\_Datatype), INTENT(IN) :: datatype 32 INTEGER, INTENT(IN) :: dest, tag 33 TYPE(MPI\_Comm), INTENT(IN) :: comm 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 35 36 MPI\_Rsend(buf, count, datatype, dest, tag, comm, ierror) 37 TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 38 INTEGER, INTENT(IN) :: count, dest, tag 39 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 40 TYPE(MPI\_Comm), INTENT(IN) :: comm 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42MPI\_Rsend\_init(buf, count, datatype, dest, tag, comm, request, ierror) 43 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 44 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 45TYPE(MPI\_Datatype), INTENT(IN) :: datatype 46INTEGER, INTENT(IN) :: dest, tag 47TYPE(MPI\_Comm), INTENT(IN) :: comm 48

```
1
         TYPE(MPI_Request), INTENT(OUT) :: request
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
4
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
5
         INTEGER, INTENT(IN) :: count, dest, tag
6
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
         TYPE(MPI_Comm), INTENT(IN) :: comm
8
         TYPE(MPI_Request), INTENT(OUT) :: request
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
12
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
13
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
14
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15
         INTEGER, INTENT(IN) :: dest, tag
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
19
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
20
         INTEGER, INTENT(IN) :: count, dest, tag
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
     MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
26
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
27
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         INTEGER, INTENT(IN) :: dest, tag
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         TYPE(MPI_Request), INTENT(OUT) :: request
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
34
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
35
         INTEGER, INTENT(IN) :: count, dest, tag
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         TYPE(MPI_Request), INTENT(OUT) :: request
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
42
                  recvcount, recvtype, source, recvtag, comm, status, ierror)
43
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
44
         INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
45
                   recvtag
46
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47
         TYPE(*), DIMENSION(...) :: recvbuf
48
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   1
    TYPE(MPI_Status) :: status
                                                                                   2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
                                                                                   5
              comm, status, ierror)
                                                                                   6
    TYPE(*). DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   9
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   10
    TYPE(MPI_Status) :: status
                                                                                   11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   12
                                                                                   13
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
                                                                                   14
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   15
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                   16
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   17
    INTEGER, INTENT(IN) :: dest, tag
                                                                                   18
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   19
                                                                                   20
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
                                                                                   21
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   22
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                   23
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   26
                                                                                  27
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                  28
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                   29
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                   30
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   31
    INTEGER, INTENT(IN) :: dest, tag
                                                                                   32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                  36
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  37
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                   38
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   40
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   42
                                                                                   43
MPI_Start(request, ierror)
                                                                                   44
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                   45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   46
MPI_Startall(count, array_of_requests, ierror)
                                                                                   47
    INTEGER, INTENT(IN) :: count
                                                                                   48
```

```
1
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Test(request, flag, status, ierror)
4
         TYPE(MPI_Request), INTENT(INOUT) :: request
5
         LOGICAL, INTENT(OUT) :: flag
6
         TYPE(MPI_Status) :: status
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Test_cancelled(status, flag, ierror)
10
         TYPE(MPI_Status), INTENT(IN) :: status
11
         LOGICAL, INTENT(OUT) :: flag
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
14
         INTEGER, INTENT(IN) :: count
15
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
16
         LOGICAL, INTENT(OUT) :: flag
17
         TYPE(MPI_Status) :: array_of_statuses(*)
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_Testany(count, array_of_requests, index, flag, status, ierror)
21
         INTEGER, INTENT(IN) :: count
22
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
23
         INTEGER, INTENT(OUT) :: index
24
         LOGICAL, INTENT(OUT) :: flag
25
         TYPE(MPI_Status) :: status
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
28
                   array_of_statuses, ierror)
29
         INTEGER, INTENT(IN) :: incount
30
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
31
         INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
32
         TYPE(MPI_Status) :: array_of_statuses(*)
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Wait(request, status, ierror)
36
         TYPE(MPI_Request), INTENT(INOUT) :: request
37
         TYPE(MPI_Status) :: status
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
40
         INTEGER, INTENT(IN) :: count
41
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
42
         TYPE(MPI_Status) :: array_of_statuses(*)
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Waitany(count, array_of_requests, index, status, ierror)
46
         INTEGER, INTENT(IN) :: count
47
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
48
```

```
1
    INTEGER, INTENT(OUT) :: index
                                                                                   2
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
                                                                                   5
              array_of_statuses, ierror)
                                                                                   6
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
                                                                                   9
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                   10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   11
                                                                                   12
                                                                                   13
A.4.2 Partitioned Communication Fortran 2008 Bindings
                                                                                   14
MPI_Parrived(request, partition, flag, ierror)
                                                                                   15
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                   16
    INTEGER, INTENT(IN) :: partition
                                                                                   17
    LOGICAL, INTENT(OUT) :: flag
                                                                                   18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   19
                                                                                   20
MPI_Pready(partition, request, ierror)
                                                                                   21
    INTEGER, INTENT(IN) :: partition
                                                                                   22
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                   23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   24
MPI_Pready_list(length, array_of_partitions, request, ierror)
                                                                                   25
    INTEGER, INTENT(IN) :: length
                                                                                   26
    INTEGER, INTENT(INOUT) :: array_of_partitions(length)
                                                                                   27
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                   28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   29
                                                                                   30
MPI_Pready_range(partition_low, partition_high, request, ierror)
                                                                                   31
    INTEGER, INTENT(IN) :: partition_low, partition_high
                                                                                   32
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                   33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   34
MPI_Precv_init(buf, partitions, count, datatype, dest, tag, comm, info,
                                                                                   35
              request, ierror)
                                                                                   36
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   37
    INTEGER, INTENT(IN) :: partitions, dest, tag
                                                                                   38
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                   39
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   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_Psend_init(buf, partitions, count, datatype, dest, tag, comm, info,
                                                                                   46
              request, ierror)
                                                                                   47
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                   48
```

```
1
         INTEGER, INTENT(IN) :: partitions, dest, tag
2
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
3
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
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
9
     A.4.3 Datatypes Fortran 2008 Bindings
10
11
     INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)
12
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp
13
     INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2)
14
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2
15
16
     MPI_Get_address(location, address, ierror)
17
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: location
18
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_Get_elements(status, datatype, count, ierror)
21
         TYPE(MPI_Status), INTENT(IN) :: status
22
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_Get_elements(status, datatype, count, ierror)
27
         TYPE(MPI_Status), INTENT(IN) :: status
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         INTEGER, INTENT(OUT) :: count
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
     MPI_Get_elements_x(status, datatype, count, ierror)
32
         TYPE(MPI_Status), INTENT(IN) :: status
33
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
38
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
39
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount, outsize
40
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
         TYPE(*), DIMENSION(..) :: outbuf
42
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
46
47
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
48
         INTEGER, INTENT(IN) :: incount, outsize
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  2
    TYPE(*), DIMENSION(..) :: outbuf
    INTEGER, INTENT(INOUT) :: position
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
             position, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                  10
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount, outsize
                                                                                  11
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  12
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                  13
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
                                                                                  16
                                                                                  17
             position, ierror)
                                                                                  18
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                  19
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
    INTEGER, INTENT(IN) :: incount
                                                                                  20
                                                                                  21
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  22
    TYPE(*), DIMENSION(...) :: outbuf
                                                                                  23
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
                                                                                  24
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
                                                                                  27
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                  28
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  30
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
                                                                                  31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  32
                                                                                  33
MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
                                                                                  34
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                  35
    INTEGER, INTENT(IN) :: incount
                                                                                  36
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  37
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
MPI_Pack_size(incount, datatype, comm, size, ierror)
                                                                                  40
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
                                                                                  41
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  43
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
                                                                                  44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  45
                                                                                  46
MPI_Pack_size(incount, datatype, comm, size, ierror)
                                                                                  47
    INTEGER, INTENT(IN) :: incount
                                                                                  48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         INTEGER, INTENT(OUT) :: size
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Type_commit(datatype, ierror)
6
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Type_contiguous(count, oldtype, newtype, ierror)
10
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
11
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
12
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_Type_contiguous(count, oldtype, newtype, ierror)
15
         INTEGER, INTENT(IN) :: count
16
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
17
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
21
                   array_of_distribs, array_of_dargs, array_of_psizes, order,
22
                   oldtype, newtype, ierror)
23
         INTEGER, INTENT(IN) :: size, rank, ndims, array_of_distribs(ndims),
24
                   array_of_dargs(ndims), array_of_psizes(ndims), order
25
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_gsizes(ndims)
26
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
27
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
30
                   array_of_distribs, array_of_dargs, array_of_psizes, order,
31
                   oldtype, newtype, ierror)
32
         INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
33
                   array_of_distribs(ndims), array_of_dargs(ndims),
34
                   array_of_psizes(ndims), order
35
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
36
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Type_create_hindexed(count, array_of_blocklengths,
40
                   array_of_displacements, oldtype, newtype, ierror)
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
42
                   array_of_blocklengths(count), array_of_displacements(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_hindexed(count, array_of_blocklengths,
47
                   array_of_displacements, oldtype, newtype, ierror)
48
```

	1
INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)	1
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::	3
array_of_displacements(count)	4
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	5
TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
INIEGER, OPIIONAL, INIENI(UUI) :: 1error	7
MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,	
oldtype, newtype, ierror)	9
<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength,</pre>	10
array_of_displacements(count)	11
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	12
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,	15
oldtype, newtype, ierror)	16
INTEGER, INTENT(IN) :: count, blocklength	17
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::	18
array_of_displacements(count)	19
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	20
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
	23
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,	24
ierror)	25
<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, stride TYPE(MPI_Datatype), INTENT(IN) :: oldtype</pre>	26
TYPE(MPI_Datatype), INTENT(IN) ofdtype TYPE(MPI_Datatype), INTENT(OUT) :: newtype	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
INTEGER, OFFICIARE, INTENT(COT) TEFTOT	29
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,	30
ierror)	31 32
INTEGER, INTENT(IN) :: count, blocklength	32
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride	34
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	35
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,	38
oldtype, newtype, ierror)	39
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength,	40
array_of_displacements(count)	41
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	42
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,	45
oldtype, newtype, ierror)	46
	47
	48

```
1
         INTEGER, INTENT(IN) :: count, blocklength,
2
                   array_of_displacements(count)
3
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
4
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
    MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
7
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
8
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
9
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
13
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
14
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: lb, extent
15
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Type_create_struct(count, array_of_blocklengths,
18
                  array_of_displacements, array_of_types, newtype, ierror)
19
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
20
                   array_of_blocklengths(count), array_of_displacements(count)
21
         TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
22
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
    MPI_Type_create_struct(count, array_of_blocklengths,
26
                  array_of_displacements, array_of_types, newtype, ierror)
27
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
28
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
29
                   array_of_displacements(count)
30
         TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
31
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
34
                  array_of_starts, order, oldtype, newtype, ierror)
35
         INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
36
                   array_of_subsizes(ndims), array_of_starts(ndims), order
37
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
38
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
42
                  array_of_starts, order, oldtype, newtype, ierror)
43
         INTEGER, INTENT(IN) :: ndims, order
44
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_sizes(ndims),
45
                   array_of_subsizes(ndims), array_of_starts(ndims)
46
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
47
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  2
MPI_Type_dup(oldtype, newtype, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  4
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  5
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  6
MPI_Type_free(datatype, ierror)
    TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
                                                                                  9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  10
MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
                                                                                  11
              array_of_integers, array_of_addresses, array_of_datatypes,
                                                                                  12
              ierror)
                                                                                  13
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  14
    INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
                                                                                  15
    INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
                                                                                  16
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
                                                                                  17
              array_of_addresses(max_addresses)
                                                                                  18
    TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
                                                                                  19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  20
                                                                                  21
MPI_Type_get_contents(datatype, max_integers, max_addresses,
                                                                                  22
             max_large_counts, max_datatypes, array_of_integers,
                                                                                  23
             array_of_addresses, array_of_large_counts, array_of_datatypes,
                                                                                  24
             ierror)
                                                                                  25
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  26
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: max_integers,
                                                                                  27
              max_addresses, max_large_counts, 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
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::
                                                                                  32
              array_of_large_counts(max_large_counts)
                                                                                  33
    TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
                                                                                  36
             combiner, ierror)
                                                                                  37
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  38
    INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,
                                                                                  39
              combiner
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
MPI_Type_get_envelope(datatype, num_integers, num_addresses,
                                                                                  43
             num_large_counts, num_datatypes, combiner, ierror)
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  45
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: num_integers,
                                                                                  46
              num_addresses, num_large_counts, num_datatypes
                                                                                  47
    INTEGER, INTENT(OUT) :: combiner
                                                                                  48
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI\_Type\_get\_extent(datatype, lb, extent, ierror) 3 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 4 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: lb, extent 5INTEGER, OPTIONAL, INTENT(OUT) :: ierror 6 7 MPI\_Type\_get\_extent(datatype, lb, extent, ierror) 8 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 9 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: lb, extent 10INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 MPI\_Type\_get\_extent\_x(datatype, lb, extent, ierror) 12TYPE(MPI\_Datatype), INTENT(IN) :: datatype 13 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: lb, extent 14 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516MPI\_Type\_get\_true\_extent(datatype, true\_lb, true\_extent, ierror) 17 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 18 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: true\_lb, true\_extent 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI\_Type\_get\_true\_extent(datatype, true\_lb, true\_extent, ierror) 21TYPE(MPI\_Datatype), INTENT(IN) :: datatype 22 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: true\_lb, true\_extent 23 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2425MPI\_Type\_get\_true\_extent\_x(datatype, true\_lb, true\_extent, ierror) 26TYPE(MPI\_Datatype), INTENT(IN) :: datatype 27INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: true\_lb, true\_extent 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 MPI\_Type\_indexed(count, array\_of\_blocklengths, array\_of\_displacements, 30 oldtype, newtype, ierror) 31INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count, 32 array\_of\_blocklengths(count), array\_of\_displacements(count) 33 TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 34 TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI\_Type\_indexed(count, array\_of\_blocklengths, array\_of\_displacements, 38 oldtype, newtype, ierror) 39 INTEGER, INTENT(IN) :: count, array\_of\_blocklengths(count), 40array\_of\_displacements(count) 41 TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 42TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 MPI\_Type\_size(datatype, size, ierror) 45TYPE(MPI\_Datatype), INTENT(IN) :: datatype 46INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: size 47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

1 MPI\_Type\_size(datatype, size, ierror) 2 TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror 5 MPI\_Type\_size\_x(datatype, size, ierror) 6 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 7 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10 MPI\_Type\_vector(count, blocklength, stride, oldtype, newtype, ierror) 11 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count, blocklength, stride TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 1213 TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 15MPI\_Type\_vector(count, blocklength, stride, oldtype, newtype, ierror) 16INTEGER, INTENT(IN) :: count, blocklength, stride 17TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 18 TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI\_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, 22 ierror) 23TYPE(\*), DIMENSION(...), INTENT(IN) :: inbuf 24INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: insize, outcount 25INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(INOUT) :: position 26TYPE(\*), DIMENSION(..) :: outbuf 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, 31ierror) 32 TYPE(\*), DIMENSION(...), INTENT(IN) :: inbuf 33 INTEGER, INTENT(IN) :: insize, outcount 34 INTEGER, INTENT(INOUT) :: position 35 TYPE(\*), DIMENSION(...) :: outbuf 36 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 37 TYPE(MPI\_Comm), INTENT(IN) :: comm 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40 MPI\_Unpack\_external(datarep, inbuf, insize, position, outbuf, outcount, 41 datatype, ierror) 42CHARACTER(LEN=\*), INTENT(IN) :: datarep 43 TYPE(\*), DIMENSION(...), INTENT(IN) :: inbuf 44INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: insize 45INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(INOUT) :: position 46TYPE(\*), DIMENSION(..) :: outbuf 47INTEGER, INTENT(IN) :: outcount 48

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
4
                  datatype, ierror)
5
         CHARACTER(LEN=*), INTENT(IN) :: datarep
6
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
7
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
9
         TYPE(*), DIMENSION(..) :: outbuf
10
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
14
     A.4.4 Collective Communication Fortran 2008 Bindings
15
    MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
16
                   comm, ierror)
17
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
18
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
19
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
20
         TYPE(*), DIMENSION(...) :: recvbuf
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
25
                  comm, ierror)
26
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
27
         INTEGER, INTENT(IN) :: sendcount, recvcount
28
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
         TYPE(*), DIMENSION(..) :: recvbuf
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
33
34
                  recvtype, comm, info, request, ierror)
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
36
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
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
43
     MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
44
                  recvtype, comm, info, request, ierror)
45
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
         INTEGER, INTENT(IN) :: sendcount, recvcount
47
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48
```

```
TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  1
                                                                                  2
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  9
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
                                                                                  10
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  11
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  12
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                  13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  16
                                                                                  17
             recvtype, comm, ierror)
                                                                                  18
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  19
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  20
                                                                                  21
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                  25
             displs, recvtype, comm, info, request, ierror)
                                                                                  26
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  27
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  29
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  30
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                  31
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                  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_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                  38
             displs, recvtype, comm, info, request, ierror)
                                                                                  39
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  40
    INTEGER, INTENT(IN) :: sendcount
                                                                                  41
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  42
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  43
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                  44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  45
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  46
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  48
```

```
1
    MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
\mathbf{2}
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
3
         TYPE(*), DIMENSION(..) :: recvbuf
4
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         TYPE(MPI_Op), INTENT(IN) :: op
7
         TYPE(MPI_Comm), INTENT(IN) :: comm
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
    MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
10
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
11
         TYPE(*), DIMENSION(..) :: recvbuf
12
         INTEGER, INTENT(IN) :: count
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         TYPE(MPI_Op), INTENT(IN) :: op
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
19
                   request, ierror)
20
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
21
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
22
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Op), INTENT(IN) :: op
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         TYPE(MPI_Info), INTENT(IN) :: info
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
30
                   request, ierror)
31
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
32
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
33
         INTEGER, INTENT(IN) :: count
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Op), INTENT(IN) :: op
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         TYPE(MPI_Info), INTENT(IN) :: info
38
         TYPE(MPI_Request), INTENT(OUT) :: request
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
42
                   comm, ierror)
43
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
44
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
45
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
46
         TYPE(*), DIMENSION(...) :: recvbuf
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI\_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf 5 INTEGER, INTENT(IN) :: sendcount, recvcount 6 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 7 TYPE(\*), DIMENSION(..) :: recvbuf TYPE(MPI\_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 11 MPI\_Alltoall\_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request, ierror) 1213 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 14INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 15TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 16TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 17TYPE(MPI\_Comm), INTENT(IN) :: comm 18 TYPE(MPI\_Info), INTENT(IN) :: info 19 TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI\_Alltoall\_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, 22 recvtype, comm, info, request, ierror) 23TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 24INTEGER, INTENT(IN) :: sendcount, recvcount 25TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 26TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 27TYPE(MPI\_Comm), INTENT(IN) :: comm 28TYPE(MPI\_Info), INTENT(IN) :: info 29 TYPE(MPI\_Request), INTENT(OUT) :: request 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3132 MPI\_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, 33 rdispls, recvtype, comm, ierror) 34 TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf 35 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcounts(\*), 36 recvcounts(\*) 37 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: sdispls(\*), rdispls(\*) 38 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 39 TYPE(\*), DIMENSION(...) :: recvbuf 40 TYPE(MPI\_Comm), INTENT(IN) :: comm 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 42MPI\_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, 43 rdispls, recvtype, comm, ierror) 44 TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf 45INTEGER, INTENT(IN) :: sendcounts(\*), sdispls(\*), recvcounts(\*), 46rdispls(\*) 47TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 48

```
1
         TYPE(*), DIMENSION(...) :: recvbuf
2
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
5
                  recvcounts, rdispls, recvtype, comm, info, request, ierror)
6
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
7
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
8
                   sendcounts(*), recvcounts(*)
9
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS .: sdispls(*),
10
                   rdispls(*)
11
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
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
18
     MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
19
                  recvcounts, rdispls, recvtype, comm, info, request, ierror)
20
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
21
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
22
                   recvcounts(*), rdispls(*)
23
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
24
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         TYPE(MPI_Info), INTENT(IN) :: info
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
30
                  rdispls, recvtypes, comm, ierror)
31
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
32
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
33
                   recvcounts(*)
34
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
35
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
36
         TYPE(*), DIMENSION(..) :: recvbuf
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
41
                  rdispls, recvtypes, comm, ierror)
42
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
43
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
44
                   rdispls(*)
45
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
46
         TYPE(*), DIMENSION(...) :: recvbuf
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   1
                                                                                   2
MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
              recvcounts, rdispls, recvtypes, comm, info, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   5
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                   6
              sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                   8
              rdispls(*)
                                                                                   9
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                   10
              recvtypes(*)
                                                                                   11
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   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_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                   18
              recvcounts, rdispls, recvtypes, comm, info, request, ierror)
                                                                                   19
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                   20
                                                                                  21
              recvcounts(*), rdispls(*)
                                                                                   22
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                   23
              recvtypes(*)
                                                                                   24
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                   25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   26
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   29
MPI_Barrier(comm, ierror)
                                                                                   30
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   32
                                                                                   33
MPI_Barrier_init(comm, info, request, ierror)
                                                                                   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_Bcast(buffer, count, datatype, root, comm, ierror)
                                                                                   39
    TYPE(*), DIMENSION(..) :: buffer
                                                                                   40
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                   41
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   42
    INTEGER, INTENT(IN) :: root
                                                                                   43
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   45
                                                                                   46
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
                                                                                   47
    TYPE(*), DIMENSION(..) :: buffer
                                                                                   48
```

```
1
         INTEGER, INTENT(IN) :: count, root
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
6
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
7
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         INTEGER, INTENT(IN) :: root
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_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
16
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
17
         INTEGER, INTENT(IN) :: count, root
18
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Info), INTENT(IN) :: info
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(KIND=MPI_COUNT_KIND), 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_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
33
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
34
         TYPE(*), DIMENSION(...) :: recvbuf
35
         INTEGER, INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Op), INTENT(IN) :: op
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
41
                   ierror)
42
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
43
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
44
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         TYPE(MPI_Op), INTENT(IN) :: op
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                                                                                   5
             ierror)
                                                                                   6
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   7
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
                                                                                   9
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  10
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  12
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  13
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
                                                                                  16
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  17
             root, comm, ierror)
                                                                                  18
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  19
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  20
                                                                                  21
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  22
    INTEGER, INTENT(IN) :: root
                                                                                  23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  26
             root, comm, ierror)
                                                                                  27
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  28
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  30
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  35
             root, comm, info, request, ierror)
                                                                                  36
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  37
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  38
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  39
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  40
    INTEGER, INTENT(IN) :: root
                                                                                  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_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  46
             root, comm, info, request, ierror)
                                                                                  47
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  48
```

```
1
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
2
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
3
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
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_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
9
                  recvtype, root, comm, ierror)
10
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
11
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
12
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
13
         TYPE(*), DIMENSION(...) :: recvbuf
14
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
15
         INTEGER, INTENT(IN) :: root
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
20
                  recvtype, root, comm, ierror)
21
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
22
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
23
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
24
         TYPE(*), DIMENSION(...) :: recvbuf
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
28
                  recvtype, root, comm, info, request, ierror)
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
30
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
31
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
33
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
34
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
35
         INTEGER, INTENT(IN) :: root
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         TYPE(MPI_Info), INTENT(IN) :: info
38
         TYPE(MPI_Request), INTENT(OUT) :: request
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
42
                  recvtype, root, comm, info, request, ierror)
43
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
44
         INTEGER, INTENT(IN) :: sendcount, root
45
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
46
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
47
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
48
```

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  5
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  6
              comm, request, ierror)
                                                                                  7
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  9
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  10
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  12
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14
                                                                                  15
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  16
              comm, request, ierror)
                                                                                  17
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  18
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  19
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  20
                                                                                  21
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  25
              recvtype, comm, request, ierror)
                                                                                  26
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  27
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  29
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  30
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                  31
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                  32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
                                                                                  36
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  37
             recvtype, comm, request, ierror)
                                                                                  38
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  39
    INTEGER, INTENT(IN) :: sendcount
                                                                                  40
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  41
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  42
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                  43
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  44
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
                                                                                  47
              ierror)
                                                                                  48
```

```
1
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
2
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
3
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
4
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
5
         TYPE(MPI_Op), INTENT(IN) :: op
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         TYPE(MPI_Request), INTENT(OUT) :: request
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
10
                   ierror)
11
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
12
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
13
         INTEGER, INTENT(IN) :: count
14
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15
         TYPE(MPI_Op), INTENT(IN) :: op
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         TYPE(MPI_Request), INTENT(OUT) :: request
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
21
                   comm, request, ierror)
22
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
23
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
24
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
25
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
26
         TYPE(MPI_Comm), INTENT(IN) :: comm
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
30
                   comm, request, ierror)
31
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
32
         INTEGER, INTENT(IN) :: sendcount, recvcount
33
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         TYPE(MPI_Request), INTENT(OUT) :: request
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
40
                   rdispls, recvtype, comm, request, ierror)
41
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
42
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
43
                   sendcounts(*), recvcounts(*)
44
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
45
                   rdispls(*)
46
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
48
```

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   4
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                   5
              rdispls, recvtype, comm, request, ierror)
                                                                                   6
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   7
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
                                                                                   9
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  10
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  12
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14
                                                                                  15
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  16
              recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                  17
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  18
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                  19
              sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  20
                                                                                  21
              rdispls(*)
                                                                                  22
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  23
              recvtypes(*)
                                                                                  24
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  26
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  27
                                                                                  28
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  29
              recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                  30
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  31
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                  32
              recvcounts(*), rdispls(*)
                                                                                  33
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  34
              recvtypes(*)
                                                                                  35
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  37
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
                                                                                  40
MPI_Ibarrier(comm, request, ierror)
                                                                                  41
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  42
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  44
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
                                                                                  45
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                  46
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  48
```

```
1
         INTEGER, INTENT(IN) :: root
2
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         TYPE(MPI_Request), INTENT(OUT) :: request
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
6
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
7
         INTEGER, INTENT(IN) :: count, root
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_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
14
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
15
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Op), INTENT(IN) :: op
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, 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_Request), INTENT(OUT) :: request
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
33
                  root, comm, request, ierror)
34
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
36
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
38
         INTEGER, INTENT(IN) :: root
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
43
                  root, comm, request, ierror)
44
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
45
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
46
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
47
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
48
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  1
                                                                                  2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm, request, ierror)
                                                                                  6
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  7
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
                                                                                  8
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  9
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  10
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                  11
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                  12
    INTEGER, INTENT(IN) :: root
                                                                                  13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  14
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  16
                                                                                  17
MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  18
             recvtype, root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  19
    INTEGER, INTENT(IN) :: sendcount, root
                                                                                  20
                                                                                  21
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  22
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  23
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                  24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  27
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
                                                                                  28
              ierror)
                                                                                  29
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  30
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  31
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  32
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  33
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  34
    INTEGER, INTENT(IN) :: root
                                                                                  35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  36
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
                                                                                  39
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
                                                                                  40
              ierror)
                                                                                  41
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  42
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  43
    INTEGER, INTENT(IN) :: count, root
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  45
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  47
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
3
                   request, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         TYPE(MPI_Op), INTENT(IN) :: op
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Request), INTENT(OUT) :: request
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
14
                   request, ierror)
15
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
16
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
17
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
18
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
         TYPE(MPI_Op), INTENT(IN) :: op
20
         TYPE(MPI_Comm), INTENT(IN) :: comm
21
         TYPE(MPI_Request), INTENT(OUT) :: request
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
24
                   request, ierror)
25
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Op), INTENT(IN) :: op
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         TYPE(MPI_Request), INTENT(OUT) :: request
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
35
                   request, ierror)
36
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
38
         INTEGER, INTENT(IN) :: recvcount
39
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
         TYPE(MPI_Op), INTENT(IN) :: op
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Request), INTENT(OUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
45
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
47
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  2
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
                                                                                  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_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  17
             root, comm, request, ierror)
                                                                                  18
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  19
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  20
                                                                                  21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  22
    INTEGER, INTENT(IN) :: root
                                                                                  23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  24
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  27
             root, comm, request, ierror)
                                                                                  28
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  29
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  30
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  31
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
                                                                                  36
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                  37
             recvtype, root, comm, request, ierror)
                                                                                  38
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  39
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
                                                                                  40
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                  41
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  42
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  43
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                  44
    INTEGER, INTENT(IN) :: root
                                                                                  45
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  46
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  48
```

```
1
            MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
       \mathbf{2}
                          recvtype, root, comm, request, ierror)
        3
                TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
        4
                 INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
       5
                TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
        6
                TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
        7
                INTEGER, INTENT(IN) :: recvcount, root
        8
                TYPE(MPI_Comm), INTENT(IN) :: comm
        9
                TYPE(MPI_Request), INTENT(OUT) :: request
       10
                INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       11
            MPI_Op_commutative(op, commute, ierror)
       12
                TYPE(MPI_Op), INTENT(IN) :: op
       13
                LOGICAL, INTENT(OUT) :: commute
       14
                INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       15
       16
            MPI_Op_create(user_fn, commute, op, ierror)
                                                                        Pythonization group will fix this globally!!!
#-error! 17
                PROCEDURE(MPI_User_function), INTENT(IN) :: user_fn
                                                                         -> Dan.
       18
                LOGICAL, INTENT(IN) :: commute
       19
                TYPE(MPI_Op), INTENT(OUT) :: op
PR414
       20
                INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       21
            MPI_Op_create_c(user_fn, commute, op, ierror)
       22
                PROCEDURE(MPI_User_function_c), INTENT(IN) :: user_fn
#-error 23
                LOGICAL, INTENT(IN) :: commute
       24
                TYPE(MPI_Op), INTENT(OUT) :: op
       25
PR414
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       26
       27
            MPI_Op_free(op, ierror)
       28
                TYPE(MPI_Op), INTENT(INOUT) :: op
       29
                INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       30
            MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
       31
                TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
       32
                TYPE(*), DIMENSION(..) :: recvbuf
       33
                INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
       34
                TYPE(MPI_Datatype), INTENT(IN) :: datatype
       35
                TYPE(MPI_Op), INTENT(IN) :: op
       36
                INTEGER, INTENT(IN) :: root
       37
                TYPE(MPI_Comm), INTENT(IN) :: comm
       38
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       39
       40
            MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
       41
                TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
       42
                TYPE(*), DIMENSION(...) :: recvbuf
       43
                INTEGER, INTENT(IN) :: count, root
       44
                TYPE(MPI_Datatype), INTENT(IN) :: datatype
       45
                TYPE(MPI_Op), INTENT(IN) :: op
       46
                TYPE(MPI_Comm), INTENT(IN) :: comm
       47
                INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       48
```

Done

Done

in

in

```
1
MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
                                                                                   2
              request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                   5
                                                                                   6
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    INTEGER, INTENT(IN) :: root
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   9
                                                                                   10
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   12
                                                                                   13
MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
                                                                                  14
              request, ierror)
                                                                                   15
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   16
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                   17
    INTEGER, INTENT(IN) :: count, root
                                                                                   18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   19
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  21
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
                                                                                  25
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
                                                                                   26
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                  27
    TYPE(*), DIMENSION(..) :: inoutbuf
                                                                                  28
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  30
                                                                                   31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  32
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
                                                                                  33
    TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
                                                                                  34
    TYPE(*), DIMENSION(..) :: inoutbuf
                                                                                  35
    INTEGER, INTENT(IN) :: count
                                                                                  36
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  37
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
                                                                                   40
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                  41
              ierror)
                                                                                  42
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  43
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                   44
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcounts(*)
                                                                                   45
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   46
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
3
                   ierror)
4
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
5
         TYPE(*), DIMENSION(..) :: recvbuf
6
         INTEGER, INTENT(IN) :: recvcounts(*)
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_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
13
                   ierror)
14
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
15
         TYPE(*), DIMENSION(...) :: recvbuf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Op), INTENT(IN) :: op
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
22
                   ierror)
23
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
24
         TYPE(*), DIMENSION(..) :: recvbuf
25
         INTEGER, INTENT(IN) :: recvcount
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Op), INTENT(IN) :: op
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
32
                   comm, info, request, ierror)
33
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
34
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
35
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Op), INTENT(IN) :: op
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_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
43
                   comm, info, request, ierror)
44
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
45
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: 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
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
             info, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   9
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  10
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                  11
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  12
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  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_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                  18
                                                                                  19
              info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  20
                                                                                  21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  22
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                  23
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  24
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  26
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  29
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
                                                                                  30
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  31
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  32
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  33
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  34
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  37
                                                                                  38
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
                                                                                  39
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  40
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  41
    INTEGER, INTENT(IN) :: count
                                                                                  42
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  43
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                                                                                  47
              ierror)
                                                                                  48
```

1 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 2 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 3 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 4 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 5TYPE(MPI\_Op), INTENT(IN) :: op 6 TYPE(MPI\_Comm), INTENT(IN) :: comm 7 TYPE(MPI\_Info), INTENT(IN) :: info 8 TYPE(MPI\_Request), INTENT(OUT) :: request 9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 MPI\_Scan\_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, 11 ierror) 12TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 13 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 14 INTEGER, INTENT(IN) :: count 15TYPE(MPI\_Datatype), INTENT(IN) :: datatype 16 TYPE(MPI\_Op), INTENT(IN) :: op 17 TYPE(MPI\_Comm), INTENT(IN) :: comm 18 TYPE(MPI\_Info), INTENT(IN) :: info 19 TYPE(MPI\_Request), INTENT(OUT) :: request 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122 MPI\_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 23root, comm, ierror) 24TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 25INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 26TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 27TYPE(\*), DIMENSION(..) :: recvbuf 28INTEGER, INTENT(IN) :: root 29 TYPE(MPI\_Comm), INTENT(IN) :: comm 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 31MPI\_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 32 root, comm, ierror) 33 TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 34 INTEGER, INTENT(IN) :: sendcount, recvcount, root 35TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 36 TYPE(\*), DIMENSION(..) :: recvbuf 37 TYPE(MPI\_Comm), INTENT(IN) :: comm 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40MPI\_Scatter\_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, 41 recvtype, root, comm, info, request, ierror) 42TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 43 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 44 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 45TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 46 INTEGER, INTENT(IN) :: root 47 TYPE(MPI\_Comm), INTENT(IN) :: comm 48

```
TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   1
                                                                                   2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                   5
              recvtype, root, comm, info, request, ierror)
                                                                                   6
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  7
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  9
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  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_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                  16
              recvtype, root, comm, ierror)
                                                                                  17
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  18
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcount
                                                                                  19
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  20
                                                                                  21
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  22
    INTEGER, INTENT(IN) :: root
                                                                                  23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                  26
              recvtype, root, comm, ierror)
                                                                                  27
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  28
    INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  30
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
                                                                                  35
              recvcount, recvtype, root, comm, info, request, ierror)
                                                                                  36
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  37
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
                                                                                  38
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                  39
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  40
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  41
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                  42
    INTEGER, INTENT(IN) :: root
                                                                                  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
```

48

```
1
     MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
2
                   recvcount, recvtype, root, comm, info, request, ierror)
3
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
4
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
5
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
6
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
7
         INTEGER, INTENT(IN) :: recvcount, root
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         TYPE(MPI_Info), INTENT(IN) :: info
10
         TYPE(MPI_Request), INTENT(OUT) :: request
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     A.4.5 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings
14
15
     MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
16
                   attribute_val_out, flag, ierror)
17
         TYPE(MPI_Comm) :: oldcomm
18
         INTEGER :: comm_keyval, ierror
19
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
20
                   attribute_val_out
21
         LOGICAL :: flag
22
     MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
23
                   attribute_val_out, flag, ierror)
24
         TYPE(MPI_Comm) :: oldcomm
25
         INTEGER :: comm_keyval, ierror
26
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
27
                   attribute_val_out
28
         LOGICAL :: flag
29
30
     MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
^{31}
                   ierror)
32
         TYPE(MPI_Comm) :: comm
33
         INTEGER :: comm_keyval, ierror
34
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
35
     MPI_Comm_compare(comm1, comm2, result, ierror)
36
         TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
37
         INTEGER, INTENT(OUT) :: result
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Comm_create(comm, group, newcomm, ierror)
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Group), INTENT(IN) :: group
43
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm,
46
47
                   ierror)
         TYPE(MPI_Group), INTENT(IN) :: group
48
```

1 CHARACTER(LEN=\*), INTENT(IN) :: stringtag TYPE(MPI\_Info), INTENT(IN) :: info 2 TYPE(MPI\_Errhandler), INTENT(IN) :: errhandler TYPE(MPI\_Comm), INTENT(OUT) :: newcomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_Comm\_create\_group(comm, group, tag, newcomm, ierror) TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Group), INTENT(IN) :: group INTEGER, INTENT(IN) :: tag 10 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI\_Comm\_create\_keyval(comm\_copy\_attr\_fn, comm\_delete\_attr\_fn, comm\_keyval, 14extra\_state, ierror) <sup>15</sup> #-error PROCEDURE(MPI\_Comm\_copy\_attr\_function), INTENT(IN) :: comm\_copy\_attr\_fn 16PROCEDURE(MPI\_Comm\_delete\_attr\_function), INTENT(IN) :: Done 17in comm\_delete\_attr\_fn PR414 18 INTEGER, INTENT(OUT) :: comm\_keyval 19 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: extra\_state INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI\_Comm\_delete\_attr(comm, comm\_kevval, ierror) 22 TYPE(MPI\_Comm), INTENT(IN) :: comm 23INTEGER, INTENT(IN) :: comm\_keyval 24INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526MPI\_Comm\_dup(comm, newcomm, ierror) 27TYPE(MPI\_Comm), INTENT(IN) :: comm 28TYPE(MPI\_Comm), INTENT(OUT) :: newcomm 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI\_Comm\_dup\_with\_info(comm, info, newcomm, ierror) 31TYPE(MPI\_Comm), INTENT(IN) :: comm 32 TYPE(MPI\_Info), INTENT(IN) :: info 33 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3536 MPI\_Comm\_free(comm, ierror) 37 TYPE(MPI\_Comm), INTENT(INOUT) :: comm 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 MPI\_Comm\_free\_keyval(comm\_keyval, ierror) 40 INTEGER, INTENT(INOUT) :: comm\_keyval 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4243 MPI\_Comm\_get\_attr(comm, comm\_keyval, attribute\_val, flag, ierror) 44 TYPE(MPI\_Comm), INTENT(IN) :: comm 45INTEGER, INTENT(IN) :: comm\_keyval 46INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: attribute\_val 47LOGICAL, INTENT(OUT) :: flag 48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
    MPI_Comm_get_info(comm, info_used, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Info), INTENT(OUT) :: info_used
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
     MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
10
         INTEGER, INTENT(OUT) :: resultlen
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Comm_group(comm, group, ierror)
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Group), INTENT(OUT) :: group
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Comm_idup(comm, newcomm, request, ierror)
18
         TYPE(MPI_Comm), INTENT(IN) :: comm
19
         TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         TYPE(MPI_Info), INTENT(IN) :: info
25
         TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
26
         TYPE(MPI_Request), INTENT(OUT) :: request
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
    MPI_Comm_rank(comm, rank, ierror)
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, INTENT(OUT) :: rank
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
    MPI_Comm_remote_group(comm, group, ierror)
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(MPI_Group), INTENT(OUT) :: group
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Comm_remote_size(comm, size, ierror)
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         INTEGER, INTENT(OUT) :: size
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
    MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         INTEGER, INTENT(IN) :: comm_keyval
45
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

MPI_Comm_set_info(comm, info, ierror)	1
TYPE(MPI_Comm), INTENT(IN) :: comm	2
TYPE(MPI_Info), INTENT(IN) :: info	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
MPI_Comm_set_name(comm, comm_name, ierror)	5
TYPE(MPI_Comm), INTENT(IN) :: comm	6
CHARACTER(LEN=*), INTENT(IN) :: comm_name	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
	9
MPI_Comm_size(comm, size, ierror)	10
TYPE(MPI_Comm), INTENT(IN) :: comm	11
INTEGER, INTENT(OUT) :: size	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13 14
MPI_Comm_split(comm, color, key, newcomm, ierror)	14
TYPE(MPI_Comm), INTENT(IN) :: comm	16
INTEGER, INTENT(IN) :: color, key	10
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
	20
<pre>MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror) TVDE(MDI_Comm) INTENT(IN) + comm</pre>	21
TYPE(MPI_Comm), INTENT(IN) :: comm	22
INTEGER, INTENT(IN) :: split_type, key	23
TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(OUT) :: newcomm	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
INIEGER, OPTIONAL, INIENI(001) :. Terror	26
MPI_Comm_test_inter(comm, flag, ierror)	27
TYPE(MPI_Comm), INTENT(IN) :: comm	28
LOGICAL, INTENT(OUT) :: flag	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
MPI_Group_compare(group1, group2, result, ierror)	31
TYPE(MPI_Group), INTENT(IN) :: group1, group2	32
INTEGER, INTENT(OUT) :: result	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
	35
<pre>MPI_Group_difference(group1, group2, newgroup, ierror)</pre>	36
TYPE(MPI_Group), INTENT(IN) :: group1, group2	37
TYPE(MPI_Group), INTENT(OUT) :: newgroup	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
MPI_Group_excl(group, n, ranks, newgroup, ierror)	40
TYPE(MPI_Group), INTENT(IN) :: group	41
INTEGER, INTENT(IN) :: n, ranks(n)	42
TYPE(MPI_Group), INTENT(OUT) :: newgroup	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
	45
MPI_Group_free(group, ierror)	46
TYPE(MPI_Group), INTENT(INOUT) :: group	47
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48

```
1
     MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)
\mathbf{2}
         TYPE(MPI_Session), INTENT(IN) :: session
3
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
4
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Group_incl(group, n, ranks, newgroup, ierror)
7
         TYPE(MPI_Group), INTENT(IN) :: group
8
         INTEGER, INTENT(IN) :: n, ranks(n)
9
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Group_intersection(group1, group2, newgroup, ierror)
13
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
14
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
17
         TYPE(MPI_Group), INTENT(IN) :: group
18
         INTEGER, INTENT(IN) :: n, ranges(3, n)
19
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
23
         TYPE(MPI_Group), INTENT(IN) :: group
24
         INTEGER, INTENT(IN) :: n, ranges(3, n)
25
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Group_rank(group, rank, ierror)
28
         TYPE(MPI_Group), INTENT(IN) :: group
29
         INTEGER, INTENT(OUT) :: rank
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Group_size(group, size, ierror)
33
         TYPE(MPI_Group), INTENT(IN) :: group
34
         INTEGER, INTENT(OUT) :: size
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
37
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
38
         INTEGER, INTENT(IN) :: n, ranks1(n)
39
         INTEGER, INTENT(OUT) :: ranks2(n)
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_Group_union(group1, group2, newgroup, ierror)
43
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
44
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
47
                   tag, newintercomm, ierror)
48
```

```
1
    TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
                                                                                   2
    INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
                                                                                   6
              remote_leader, stringtag, info, errhandler, newintercomm,
              ierror)
    TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group
    INTEGER, INTENT(IN) :: local_leader, remote_leader
                                                                                   10
    CHARACTER(LEN=*), INTENT(IN) :: stringtag
                                                                                   11
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   12
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
                                                                                   13
    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
                                                                                   14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   15
                                                                                   16
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
                                                                                   17
    TYPE(MPI_Comm), INTENT(IN) :: intercomm
                                                                                   18
    LOGICAL, INTENT(IN) :: high
    TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
                                                                                   21
MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
                                                                                   22
              attribute_val_out, flag, ierror)
                                                                                   23
    TYPE(MPI_Datatype) :: oldtype
                                                                                   24
    INTEGER :: type_keyval, ierror
                                                                                   25
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                   26
              attribute_val_out
                                                                                   27
    LOGICAL :: flag
                                                                                   28
                                                                                   29
MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
                                                                                   30
              attribute_val_out, flag, ierror)
                                                                                   31
    TYPE(MPI_Datatype) :: oldtype
                                                                                   32
    INTEGER :: type_keyval, ierror
                                                                                   33
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                   34
              attribute_val_out
                                                                                   35
    LOGICAL :: flag
                                                                                   36
MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
                                                                                  37
              ierror)
                                                                                   38
    TYPE(MPI_Datatype) :: datatype
                                                                                   39
    INTEGER :: type_keyval
                                                                                   40
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                   41
    INTEGER, INTENT(OUT) :: ierror
                                                                                   42
                                                                                   43
MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
                                                                                   44
              extra_state, ierror)
                                                                                   <sup>45</sup> #-error
    PROCEDURE(MPI_Type_copy_attr_function), INTENT(IN) :: type_copy_attr_fn
                                                                                   46
    PROCEDURE(MPI_Type_delete_attr_function), INTENT(IN) ::
                                                                                      Done
                                                                                   47
                                                                                      in
              type_delete_attr_fn
                                                                                      PR414
                                                                                   48
```

```
1
         INTEGER, INTENT(OUT) :: type_keyval
2
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Type_delete_attr(datatype, type_keyval, ierror)
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         INTEGER, INTENT(IN) :: type_keyval
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Type_free_keyval(type_keyval, ierror)
10
         INTEGER, INTENT(INOUT) :: type_keyval
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         INTEGER, INTENT(IN) :: type_keyval
15
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
16
         LOGICAL, INTENT(OUT) :: flag
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Type_get_name(datatype, type_name, resultlen, ierror)
20
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
22
         INTEGER, INTENT(OUT) :: resultlen
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         INTEGER, INTENT(IN) :: type_keyval
27
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Type_set_name(datatype, type_name, ierror)
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         CHARACTER(LEN=*), INTENT(IN) :: type_name
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
35
                   attribute_val_out, flag, ierror)
36
         TYPE(MPI_Win) :: oldwin
37
         INTEGER :: win_keyval, ierror
38
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
39
                   attribute_val_out
40
         LOGICAL :: flag
41
42
     MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
43
                   attribute_val_out, flag, ierror)
44
         TYPE(MPI_Win) :: oldwin
45
         INTEGER :: win_keyval, ierror
46
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
47
                   attribute_val_out
48
```

```
LOGICAL :: flag
                                                                                    2
MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
    TYPE(MPI_Win) :: win
    INTEGER :: win_keyval, ierror
                                                                                    5
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                    6
                                                                                    7
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
              extra_state, ierror)
                                                                                   <sup>9</sup> #-error
    PROCEDURE(MPI_Win_copy_attr_function), INTENT(IN) :: win_copy_attr_fn
                                                                                   10
    PROCEDURE(MPI_Win_delete_attr_function), INTENT(IN) ::
                                                                                      Done
                                                                                   11
                                                                                      lin
              win_delete_attr_fn
                                                                                      PR414
                                                                                   12
    INTEGER, INTENT(OUT) :: win_keyval
                                                                                   13
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                   14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   15
MPI_Win_delete_attr(win, win_keyval, ierror)
                                                                                   16
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   17
    INTEGER, INTENT(IN) :: win_keyval
                                                                                   18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   19
                                                                                   20
MPI_Win_free_keyval(win_keyval, ierror)
                                                                                   21
    INTEGER, INTENT(INOUT) :: win_keyval
                                                                                   22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   23
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
                                                                                   24
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   25
    INTEGER, INTENT(IN) :: win_keyval
                                                                                   26
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
                                                                                   27
    LOGICAL, INTENT(OUT) :: flag
                                                                                   28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   29
                                                                                   30
MPI_Win_get_name(win, win_name, resultlen, ierror)
                                                                                   31
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   32
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
                                                                                   33
    INTEGER, INTENT(OUT) :: resultlen
                                                                                   34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   35
MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
                                                                                   36
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   37
    INTEGER, INTENT(IN) :: win_keyval
                                                                                   38
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
                                                                                   39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   40
                                                                                   41
MPI_Win_set_name(win, win_name, ierror)
                                                                                   42
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   43
    CHARACTER(LEN=*), INTENT(IN) :: win_name
                                                                                   44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   45
                                                                                   46
                                                                                   47
                                                                                   48
```

```
1
     A.4.6 Process Topologies Fortran 2008 Bindings
\mathbf{2}
     MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, INTENT(IN) :: rank, maxdims
5
         INTEGER, INTENT(OUT) :: coords(maxdims)
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
10
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
11
         LOGICAL, INTENT(IN) :: periods(ndims), reorder
12
         TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         INTEGER, INTENT(IN) :: maxdims
17
         INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
18
         LOGICAL, INTENT(OUT) :: periods(maxdims)
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
24
         LOGICAL, INTENT(IN) :: periods(ndims)
25
         INTEGER, INTENT(OUT) :: newrank
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Cart_rank(comm, coords, rank, ierror)
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         INTEGER, INTENT(IN) :: coords(*)
30
         INTEGER, INTENT(OUT) :: rank
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         INTEGER, INTENT(IN) :: direction, disp
36
         INTEGER, INTENT(OUT) :: rank_source, rank_dest
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         LOGICAL, INTENT(IN) :: remain_dims(*)
41
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Cartdim_get(comm, ndims, ierror)
45
         TYPE(MPI_Comm), INTENT(IN) :: comm
46
         INTEGER, INTENT(OUT) :: ndims
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Dims_create(nnodes, ndims, dims, ierror)
                                                                                   2
    INTEGER, INTENT(IN) :: nnodes, ndims
    INTEGER, INTENT(INOUT) :: dims(ndims)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
                                                                                   6
              info, reorder, comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*),
                                                                                  9
              weights(*)
                                                                                  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
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                  19
    INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),
              outdegree, destinations(outdegree), destweights(*)
                                                                                  20
                                                                                  21
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  22
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                  23
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
                                                                                  26
              maxoutdegree, destinations, destweights, ierror)
                                                                                  27
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  28
    INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
                                                                                  29
    INTEGER, INTENT(OUT) :: sources(maxindegree),
                                                                                  30
              destinations(maxoutdegree)
                                                                                  31
    INTEGER :: sourceweights(*), destweights(*)
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
                                                                                  35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  36
    INTEGER, INTENT(OUT) :: indegree, outdegree
                                                                                  37
    LOGICAL, INTENT(OUT) :: weighted
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
                                                                                  40
              ierror)
                                                                                  41
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                  42
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
                                                                                  43
    LOGICAL, INTENT(IN) :: reorder
                                                                                  44
    TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
                                                                                  47
MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
                                                                                  48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         INTEGER, INTENT(IN) :: maxindex, maxedges
3
         INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
8
         INTEGER, INTENT(OUT) :: newrank
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         INTEGER, INTENT(IN) :: rank, maxneighbors
14
         INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
17
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         INTEGER, INTENT(IN) :: rank
19
         INTEGER, INTENT(OUT) :: nneighbors
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     MPI_Graphdims_get(comm, nnodes, nedges, ierror)
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         INTEGER, INTENT(OUT) :: nnodes, nedges
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
27
                   recvtype, comm, request, ierror)
28
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
29
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
30
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
31
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Request), INTENT(OUT) :: request
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
37
                  recvtype, comm, request, ierror)
38
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
39
         INTEGER, INTENT(IN) :: sendcount, recvcount
40
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
41
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
43
         TYPE(MPI_Request), INTENT(OUT) :: request
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
46
                   displs, recvtype, comm, request, ierror)
47
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
48
```

1 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount 2 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 3 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(\*) 4 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN), ASYNCHRONOUS :: displs(\*) 56 TYPE(MPI\_Comm), INTENT(IN) :: comm 7 TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 9 MPI\_Ineighbor\_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 10 displs, recvtype, comm, request, ierror) 11 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 12INTEGER, INTENT(IN) :: sendcount 13 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 14TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 15INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(\*), displs(\*) 16TYPE(MPI\_Comm), INTENT(IN) :: comm 17TYPE(MPI\_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI\_Ineighbor\_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, 2021recvtype, comm, request, ierror) TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 22 23INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 24TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 25TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 26TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Request), INTENT(OUT) :: request 2728INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 MPI\_Ineighbor\_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, 30 recvtype, comm, request, ierror) 31TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 32 INTEGER, INTENT(IN) :: sendcount, recvcount 33 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 34 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 35 TYPE(MPI\_Comm), INTENT(IN) :: comm 36 TYPE(MPI\_Request), INTENT(OUT) :: request 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI\_Ineighbor\_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, 40 recvcounts, rdispls, recvtype, comm, request, ierror) 41 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 42INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN), ASYNCHRONOUS :: 43 sendcounts(\*), recvcounts(\*) 44INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(\*), 45rdispls(\*) 46TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 47TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 48

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         TYPE(MPI_Request), INTENT(OUT) :: request
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
5
                  recvcounts, rdispls, recvtype, comm, request, ierror)
6
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
7
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
8
                   recvcounts(*), rdispls(*)
9
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
16
                  recvcounts, rdispls, recvtypes, comm, request, ierror)
17
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
18
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
19
                   sendcounts(*), recvcounts(*)
20
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
21
                   rdispls(*)
22
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
23
                   recvtypes(*)
24
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         TYPE(MPI_Request), INTENT(OUT) :: request
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
29
                  recvcounts, rdispls, recvtypes, comm, request, ierror)
30
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
31
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
32
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
33
                   rdispls(*)
34
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
35
                   recvtypes(*)
36
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         TYPE(MPI_Request), INTENT(OUT) :: request
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
42
                  recvtype, comm, ierror)
43
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
44
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
45
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
46
         TYPE(*), DIMENSION(...) :: recvbuf
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  2
MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  5
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  6
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  7
    TYPE(*), DIMENSION(..) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  10
                                                                                  11
MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm, info, request, ierror)
                                                                                  12
                                                                                  13
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  14
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  15
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  16
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  17
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  18
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  19
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  20
                                                                                  21
MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
                                                                                  22
             recvcount, recvtype, comm, info, request, ierror)
                                                                                  23
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  24
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  25
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  27
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  28
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  29
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  31
                                                                                  32
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                  33
             displs, recvtype, comm, ierror)
                                                                                  34
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  35
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
                                                                                  36
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  37
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  38
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                  39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                  42
             displs, recvtype, comm, ierror)
                                                                                  43
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  44
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
                                                                                  45
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  46
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
3
                   recvcounts, displs, recvtype, comm, info, request, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
6
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
9
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
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_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
16
                   recvcounts, displs, recvtype, comm, info, request, ierror)
17
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
18
         INTEGER, INTENT(IN) :: sendcount, displs(*)
19
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
20
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
21
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         TYPE(MPI_Info), INTENT(IN) :: info
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
27
                   recvtype, comm, ierror)
28
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
29
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
30
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
31
         TYPE(*), DIMENSION(..) :: recvbuf
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
36
                   recvtype, comm, ierror)
37
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
38
         INTEGER, INTENT(IN) :: sendcount, recvcount
39
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40
         TYPE(*), DIMENSION(..) :: recvbuf
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
     MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
44
                   recvcount, recvtype, comm, info, request, ierror)
45
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
47
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48
```

```
TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  1
                                                                                  2
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  5
                                                                                  6
MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  9
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  10
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  11
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  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_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                  18
             recvcounts, rdispls, recvtype, comm, ierror)
                                                                                  19
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
                                                                                  20
                                                                                  21
              recvcounts(*)
                                                                                  22
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                  23
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  24
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  27
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                  28
             recvcounts, rdispls, recvtype, comm, ierror)
                                                                                  29
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  30
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                  31
          rdispls(*)
                                                                                  32
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  33
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  36
                                                                                  37
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
                                                                                  38
             recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
                                                                                  39
             ierror)
                                                                                  40
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  41
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                  42
              sendcounts(*), recvcounts(*)
                                                                                  43
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  44
              rdispls(*)
                                                                                  45
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  46
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  48
```

```
1
         TYPE(MPI_Info), INTENT(IN) :: info
2
         TYPE(MPI_Request), INTENT(OUT) :: request
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
5
                   recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
6
                   ierror)
7
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
9
                   recvcounts(*), rdispls(*)
10
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
11
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
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_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
18
                   recvcounts, rdispls, recvtypes, comm, ierror)
19
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
21
                   recvcounts(*)
22
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
23
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
24
         TYPE(*), DIMENSION(...) :: recvbuf
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
28
                   recvcounts, rdispls, recvtypes, comm, ierror)
29
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
30
         INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
31
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
32
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
33
         TYPE(*), DIMENSION(..) :: recvbuf
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
38
                   recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
39
                   ierror)
40
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
42
                   sendcounts(*), recvcounts(*)
43
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
44
                   rdispls(*)
45
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
46
                   recvtypes(*)
47
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
48
```

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
              recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
              ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   9
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                   10
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                   11
               rdispls(*)
                                                                                   12
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                   13
               recvtvpes(*)
                                                                                   14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   16
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   17
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   19
                                                                                   20
MPI_Topo_test(comm, status, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   21
                                                                                   22
    INTEGER, INTENT(OUT) :: status
                                                                                   23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   24
                                                                                   25
A.4.7 MPI Environmental Management Fortran 2008 Bindings
                                                                                   26
                                                                                   27
DOUBLE PRECISION MPI_Wtick()
                                                                                   28
DOUBLE PRECISION MPI_Wtime()
                                                                                   29
                                                                                   30
MPI_Add_error_class(errorclass, ierror)
                                                                                   31
    INTEGER, INTENT(OUT) :: errorclass
                                                                                   32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   33
MPI_Add_error_code(errorclass, errorcode, ierror)
                                                                                   34
    INTEGER, INTENT(IN) :: errorclass
                                                                                   35
    INTEGER, INTENT(OUT) :: errorcode
                                                                                   36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   37
                                                                                   38
MPI_Add_error_string(errorcode, string, ierror)
                                                                                   39
    INTEGER, INTENT(IN) :: errorcode
                                                                                   40
    CHARACTER(LEN=*), INTENT(IN) :: string
                                                                                   41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   42
MPI_Alloc_mem(size, info, baseptr, ierror)
                                                                                   43
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                   44
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
                                                                                   45
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   46
    TYPE(C_PTR), INTENT(OUT) :: baseptr
                                                                                   47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   48
```

```
1
            MPI_Comm_call_errhandler(comm, errorcode, ierror)
        \mathbf{2}
                 TYPE(MPI_Comm), INTENT(IN) :: comm
        3
                 INTEGER, INTENT(IN) :: errorcode
        4
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        5
             MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
        6
#-error
                 PROCEDURE(MPI_Comm_errhandler_function), INTENT(IN) ::
Done
                           comm_errhandler_fn
        8
in
                 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
        9
PR414
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        10
        11
             MPI_Comm_get_errhandler(comm, errhandler, ierror)
       12
                 TYPE(MPI_Comm), INTENT(IN) :: comm
        13
                 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
        14
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        15
             MPI_Comm_set_errhandler(comm, errhandler, ierror)
        16
                 TYPE(MPI_Comm), INTENT(IN) :: comm
        17
                 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
        18
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        19
       20
             MPI_Errhandler_free(errhandler, ierror)
       21
                 TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
       22
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       23
             MPI_Error_class(errorcode, errorclass, ierror)
       24
                 INTEGER, INTENT(IN) :: errorcode
       25
                 INTEGER, INTENT(OUT) :: errorclass
        26
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       27
       28
             MPI_Error_string(errorcode, string, resultlen, ierror)
       29
                 INTEGER, INTENT(IN) :: errorcode
       30
                 CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
       31
                 INTEGER, INTENT(OUT) :: resultlen
        32
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       33
             MPI_File_call_errhandler(fh, errorcode, ierror)
       34
                 TYPE(MPI_File), INTENT(IN) :: fh
       35
                 INTEGER, INTENT(IN) :: errorcode
       36
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
       37
        38
             MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)
        39
#-error
                 PROCEDURE(MPI_File_errhandler_function), INTENT(IN) ::
        40
Done
                           file_errhandler_fn
        41
                 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
PR414
       42
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        43
            MPI_File_get_errhandler(file, errhandler, ierror)
       44
                 TYPE(MPI_File), INTENT(IN) :: file
        45
                 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
        46
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        47
        48
```

in

MPI_File_set_errhandler(file, errhandler, ierror)	1
TYPE(MPI_File), INTENT(IN) :: file	2
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
MDI Error mom(haga ionnon)	5
MPI_Free_mem(base, ierror)	6
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: base INTEGER, OPTIONAL, INTENT(OUT) :: ierror	7
INTEGER, OFITOWAL, INTENT(DOI) TETTOT	8
<pre>MPI_Get_library_version(version, resultlen, ierror)</pre>	9
CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version	10
INTEGER, INTENT(OUT) :: resultlen	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
MPI_Get_processor_name(name, resultlen, ierror)	13
CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name	14
INTEGER, INTENT(OUT) :: resultlen	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
	17 18
MPI_Get_version(version, subversion, ierror)	19
INTEGER, INTENT(OUT) :: version, subversion	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
MPI_Session_call_errhandler(session, errorcode, ierror)	22
TYPE(MPI_Session), INTENT(IN) :: session	23
INTEGER, INTENT(IN) :: errorcode	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
MDI Coggion excepts exchandlen (coggion exchandlen fr exchandlen isrner)	26
<pre>MPI_Session_create_errhandler(session_errhandler_fn, errhandler, ierror)     PROCEDURE(MPI_Session_errhandler_function), INTENT(IN) ::</pre>	<sup>27</sup> #-error
session_errhandler_fn	<sup>28</sup> Done
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	29 in
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	<sub>30</sub> PR414
	31
MPI_Session_get_errhandler(session, errhandler, ierror)	32
TYPE(MPI_Session), INTENT(IN) :: session	33
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
MPI_Session_set_errhandler(session, errhandler, ierror)	36
TYPE(MPI_Session), INTENT(IN) :: session	37
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
MPI_Win_call_errhandler(win, errorcode, ierror)	40
TYPE(MPI_Win), INTENT(IN) :: win	41
INTEGER, INTENT(IN) :: errorcode	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43 44
INIDADA, OFICAND, INTENT(UOT) ICITOT	44 45
MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)	10.
PROCEDURE(MPI_Win_errhandler_function), INTENT(IN) :: win_errhandler_fn	47 #-error
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	<sup>48</sup> in
	PR414

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Win_get_errhandler(win, errhandler, ierror)
3
         TYPE(MPI_Win), INTENT(IN) :: win
4
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
     MPI_Win_set_errhandler(win, errhandler, ierror)
8
         TYPE(MPI_Win), INTENT(IN) :: win
9
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     A.4.8 The Info Object Fortran 2008 Bindings
13
14
     MPI_Info_create(info, ierror)
15
         TYPE(MPI_Info), INTENT(OUT) :: info
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Info_create_env(info, ierror)
18
         TYPE(MPI_Info), INTENT(OUT) :: info
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Info_delete(info, key, ierror)
22
         TYPE(MPI_Info), INTENT(IN) :: info
23
         CHARACTER(LEN=*), INTENT(IN) :: key
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Info_dup(info, newinfo, ierror)
26
         TYPE(MPI_Info), INTENT(IN) :: info
27
         TYPE(MPI_Info), INTENT(OUT) :: newinfo
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Info_free(info, ierror)
^{31}
         TYPE(MPI_Info), INTENT(INOUT) :: info
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Info_get(info, key, valuelen, value, flag, ierror)
         TYPE(MPI_Info), INTENT(IN) :: info
35
         CHARACTER(LEN=*), INTENT(IN) :: key
36
37
         INTEGER, INTENT(IN) :: valuelen
         CHARACTER(LEN=valuelen), INTENT(OUT) :: value
38
         LOGICAL, INTENT(OUT) :: flag
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Info_get_nkeys(info, nkeys, ierror)
42
         TYPE(MPI_Info), INTENT(IN) :: info
43
         INTEGER, INTENT(OUT) :: nkeys
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Info_get_nthkey(info, n, key, ierror)
46
47
         TYPE(MPI_Info), INTENT(IN) :: info
48
         INTEGER, INTENT(IN) :: n
```

CHARACTER(LEN=*), INTENT(OUT) :: key INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$\frac{1}{2}$
MDI Info got string(info key buflen volue fleg ierner)	3
<pre>MPI_Info_get_string(info, key, buflen, value, flag, ierror)         TYPE(MPI_Info), INTENT(IN) :: info</pre>	4
CHARACTER(LEN=*), INTENT(IN) :: key	5
INTEGER, INTENT(INOUT) :: buflen	6
CHARACTER(LEN=*), INTENT(OUT) :: value	7
LOGICAL, INTENT(OUT) :: flag	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
	10
MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)	11
TYPE(MPI_Info), INTENT(IN) :: info	12
CHARACTER(LEN=*), INTENT(IN) :: key	13
INTEGER, INTENT(OUT) :: valuelen	14
LOGICAL, INTENT(OUT) :: flag	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
MPI_Info_set(info, key, value, ierror)	17 18
TYPE(MPI_Info), INTENT(IN) :: info	19
CHARACTER(LEN=*), INTENT(IN) :: key, value	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
	22
	23
A.4.9 Process Creation and Management Fortran 2008 Bindings	24
MPI_Abort(comm, errorcode, ierror)	25
TYPE(MPI_Comm), INTENT(IN) :: comm	26
INTEGER, INTENT(IN) :: errorcode	27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
	29
MPI_Close_port(port_name, ierror)	30
CHARACTER(LEN=*), INTENT(IN) :: port_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31
INTEGER, OPTIONAL, INTENI(UOI) :: Terror	32
<pre>MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)</pre>	33
CHARACTER(LEN=*), INTENT(IN) :: port_name	34
TYPE(MPI_Info), INTENT(IN) :: info	35
INTEGER, INTENT(IN) :: root	36
TYPE(MPI_Comm), INTENT(IN) :: comm	37
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)	40
CHARACTER(LEN=*), INTENT(IN) :: port_name	41
TYPE(MPI_Info), INTENT(IN) :: info	42
INTEGER, INTENT(IN) :: root	43
TYPE(MPI_Comm), INTENT(IN) :: comm	44
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46
	47 48
MPI_Comm_disconnect(comm, ierror)	48

```
1
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Comm_get_parent(parent, ierror)
4
         TYPE(MPI_Comm), INTENT(OUT) :: parent
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
     MPI_Comm_join(fd, intercomm, ierror)
8
         INTEGER, INTENT(IN) :: fd
9
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
12
                   array_of_errcodes, ierror)
13
         CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
14
         INTEGER, INTENT(IN) :: maxprocs, root
15
         TYPE(MPI_Info), INTENT(IN) :: info
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
18
         INTEGER :: array_of_errcodes(*)
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
22
                   array_of_maxprocs, array_of_info, root, comm, intercomm,
23
                   array_of_errcodes, ierror)
24
         INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
25
         CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
26
                   array_of_argv(count, *)
27
         TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
30
         INTEGER :: array_of_errcodes(*)
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI Finalize(ierror)
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Finalized(flag, ierror)
36
         LOGICAL, INTENT(OUT) :: flag
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Init(ierror)
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Init_thread(required, provided, ierror)
42
         INTEGER, INTENT(IN) :: required
43
         INTEGER, INTENT(OUT) :: provided
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Initialized(flag, ierror)
46
         LOGICAL, INTENT(OUT) :: flag
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

MPI_Is_thread_main(flag, ierror) LOGICAL, INTENT(OUT) :: flag	1 2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3
	4
MPI_Lookup_name(service_name, info, port_name, ierror)	5
CHARACTER(LEN=*), INTENT(IN) :: service_name	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
	10
MPI_Open_port(info, port_name, ierror)	10
TYPE(MPI_Info), INTENT(IN) :: info	12
CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
MPI_Publish_name(service_name, info, port_name, ierror)	14
CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name	15
TYPE(MPI_Info), INTENT(IN) :: info	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
INTEGER, OFFICIARE, INTENT(COT) TOTTOT	18
MPI_Query_thread(provided, ierror)	19
INTEGER, INTENT(OUT) :: provided	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
MDT Carrier finalize (accessor is were)	22
MPI_Session_finalize(session, ierror)	23
TYPE(MPI_Session), INTENT(INOUT) :: session	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
MPI_Session_get_info(session, info_used, ierror)	26
TYPE(MPI_Session), INTENT(IN) :: session	27
TYPE(MPI_Info), INTENT(OUT) :: info_used	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
	30
MPI_Session_get_nth_pset(session, info, n, pset_len, pset_name, ierror)	31
TYPE(MPI_Session), INTENT(IN) :: session	32
TYPE(MPI_Info), INTENT(IN) :: info	33
INTEGER, INTENT(IN) :: n	34
INTEGER, INTENT(INOUT) :: pset_len	35
CHARACTER(LEN=*), INTENT(OUT) :: pset_name	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
MPI_Session_get_num_psets(session, info, npset_names, ierror)	38
	39
TYPE(MPI_Session), INTENT(IN) :: session	40
TYPE(MPI_Info), INTENT(IN) :: info	41
INTEGER, INTENT(OUT) :: npset_names	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
MPI_Session_get_pset_info(session, pset_name, info, ierror)	44
TYPE(MPI_Session), INTENT(IN) :: session	45
CHARACTER(LEN=*), INTENT(IN) :: pset_name	46
TYPE(MPI_Info), INTENT(OUT) :: info	47
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48
,, _,, _	40

```
1
    MPI_Session_init(info, errhandler, session, ierror)
\mathbf{2}
         TYPE(MPI_Info), INTENT(IN) :: info
3
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
4
         TYPE(MPI_Session), INTENT(OUT) :: session
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
    MPI_Unpublish_name(service_name, info, port_name, ierror)
7
         CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
8
         TYPE(MPI_Info), INTENT(IN) :: info
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                    10
11
12
     A.4.10 One-Sided Communications Fortran 2008 Bindings
13
     MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
14
                  target_disp, target_count, target_datatype, op, win, ierror)
15
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
17
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
18
         INTEGER, INTENT(IN) :: target_rank
19
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
20
         TYPE(MPI_Op), INTENT(IN) :: op
21
         TYPE(MPI_Win), INTENT(IN) :: win
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
25
                  target_disp, target_count, target_datatype, op, win, ierror)
26
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
27
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
28
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
29
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
30
         TYPE(MPI_Op), INTENT(IN) :: op
31
         TYPE(MPI Win), INTENT(IN) :: win
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
34
                  target_rank, target_disp, win, ierror)
35
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr,
36
                   compare_addr
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         INTEGER, INTENT(IN) :: target_rank
40
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
41
         TYPE(MPI_Win), INTENT(IN) :: win
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
45
                  target_disp, op, win, ierror)
46
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
47
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  2
    INTEGER, INTENT(IN) :: target_rank
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  4
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  5
                                                                                  6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  7
MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, ierror)
                                                                                  9
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                  10
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
                                                                                  11
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                 12
    INTEGER, INTENT(IN) :: target_rank
                                                                                  13
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                 14
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  16
                                                                                  17
MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                 18
             target_disp, target_count, target_datatype, win, ierror)
                                                                                 19
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                 20
                                                                                 21
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                 22
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                 23
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                 24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 25
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
                                                                                 26
             result_count, result_datatype, target_rank, target_disp,
                                                                                 27
             target_count, target_datatype, op, win, ierror)
                                                                                 28
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                 29
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
                                                                                 30
              target_count
                                                                                  31
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                  32
              target_datatype
                                                                                 33
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
                                                                                 34
    INTEGER, INTENT(IN) :: target_rank
                                                                                 35
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                 36
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 37
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                 38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 39
                                                                                 40
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
                                                                                 41
             result_count, result_datatype, target_rank, target_disp,
                                                                                 42
             target_count, target_datatype, op, win, ierror)
                                                                                 43
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                 44
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
                                                                                 45
              target_count
                                                                                  46
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                  47
              target_datatype
                                                                                  48
```

```
1
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
2
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
3
         TYPE(MPI_Op), INTENT(IN) :: op
4
         TYPE(MPI_Win), INTENT(IN) :: win
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
7
                  target_disp, target_count, target_datatype, win, ierror)
8
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
9
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
10
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
11
         INTEGER, INTENT(IN) :: target_rank
12
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
13
         TYPE(MPI_Win), INTENT(IN) :: win
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
17
                  target_disp, target_count, target_datatype, win, ierror)
18
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
19
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
20
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
21
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
22
         TYPE(MPI_Win), INTENT(IN) :: win
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
25
                   target_disp, target_count, target_datatype, op, win, request,
26
                   ierror)
27
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
28
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
29
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
30
         INTEGER, INTENT(IN) :: target_rank
31
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
32
         TYPE(MPI_Op), INTENT(IN) :: op
33
         TYPE(MPI_Win), INTENT(IN) :: win
34
         TYPE(MPI_Request), INTENT(OUT) :: request
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
38
                  target_disp, target_count, target_datatype, op, win, request,
39
                  ierror)
40
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
41
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
42
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
43
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
44
         TYPE(MPI_Op), INTENT(IN) :: op
45
         TYPE(MPI_Win), INTENT(IN) :: win
46
         TYPE(MPI_Request), INTENT(OUT) :: request
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  2
             target_disp, target_count, target_datatype, win, request,
              ierror)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                  4
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
                                                                                  5
                                                                                  6
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER, INTENT(IN) :: target_rank
                                                                                  7
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  9
                                                                                  10
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  13
             target_disp, target_count, target_datatype, win, request,
                                                                                  14
              ierror)
                                                                                  15
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                  16
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                  17
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  18
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  19
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  20
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  22
                                                                                  23
MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
                                                                                  24
             result_addr, result_count, result_datatype, target_rank,
                                                                                  25
             target_disp, target_count, target_datatype, op, win, request,
                                                                                  26
              ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  27
                                                                                  28
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
                                                                                  29
              target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                  30
                                                                                  31
              target_datatype
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
                                                                                  32
                                                                                  33
    INTEGER, INTENT(IN) :: target_rank
                                                                                  34
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  35
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  36
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  37
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
                                                                                  40
             result_addr, result_count, result_datatype, target_rank,
                                                                                  41
             target_disp, target_count, target_datatype, op, win, request,
                                                                                  42
              ierror)
                                                                                  43
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  44
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
                                                                                  45
              target_count
                                                                                  46
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                  47
              target_datatype
                                                                                  48
```

```
1
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
2
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
3
         TYPE(MPI_Op), INTENT(IN) :: op
4
         TYPE(MPI_Win), INTENT(IN) :: win
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
8
                  target_disp, target_count, target_datatype, win, request,
9
                  ierror)
10
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
11
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
12
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
13
         INTEGER, INTENT(IN) :: target_rank
14
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
15
         TYPE(MPI_Win), INTENT(IN) :: win
16
         TYPE(MPI_Request), INTENT(OUT) :: request
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
20
                  target_disp, target_count, target_datatype, win, request,
21
                  ierror)
22
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
23
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
24
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
25
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
26
         TYPE(MPI_Win), INTENT(IN) :: win
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
30
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
31
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
32
         INTEGER, INTENT(IN) :: disp_unit
33
         TYPE(MPI_Info), INTENT(IN) :: info
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(C_PTR), INTENT(OUT) :: baseptr
36
         TYPE(MPI_Win), INTENT(OUT) :: win
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
40
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
41
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
42
         TYPE(MPI_Info), INTENT(IN) :: info
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         TYPE(C_PTR), INTENT(OUT) :: baseptr
45
         TYPE(MPI_Win), INTENT(OUT) :: win
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
    MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
48
```

```
1
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                   2
    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
                                                                                   5
                                                                                   6
    TYPE(C_PTR), INTENT(OUT) :: baseptr
    TYPE(MPI_Win), INTENT(OUT) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
                                                                                  10
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                  11
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
                                                                                  12
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  14
    TYPE(C_PTR), INTENT(OUT) :: baseptr
                                                                                  15
    TYPE(MPI_Win), INTENT(OUT) :: win
                                                                                  16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  17
                                                                                  18
MPI_Win_attach(win, base, size, ierror)
                                                                                  19
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
                                                                                  20
                                                                                  21
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
MPI_Win_complete(win, ierror)
                                                                                  ^{24}
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
                                                                                  27
MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
                                                                                  28
                                                                                  29
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
                                                                                  30
    INTEGER, INTENT(IN) :: disp_unit
                                                                                  31
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  33
    TYPE(MPI_Win), INTENT(OUT) :: win
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
                                                                                  36
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
                                                                                  37
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
                                                                                  38
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  40
    TYPE(MPI_Win), INTENT(OUT) :: win
                                                                                  41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  42
                                                                                  43
MPI_Win_create_dynamic(info, comm, win, ierror)
                                                                                  44
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  45
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  46
    TYPE(MPI_Win), INTENT(OUT) :: win
                                                                                  47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  48
```

```
1
    MPI_Win_detach(win, base, ierror)
\mathbf{2}
         TYPE(MPI_Win), INTENT(IN) :: win
3
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Win_fence(assert, win, ierror)
6
         INTEGER, INTENT(IN) :: assert
7
         TYPE(MPI_Win), INTENT(IN) :: win
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     MPI_Win_flush(rank, win, ierror)
11
         INTEGER, INTENT(IN) :: rank
12
         TYPE(MPI_Win), INTENT(IN) :: win
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_Win_flush_all(win, ierror)
15
         TYPE(MPI_Win), INTENT(IN) :: win
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Win_flush_local(rank, win, ierror)
19
         INTEGER, INTENT(IN) :: rank
20
         TYPE(MPI_Win), INTENT(IN) :: win
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_Win_flush_local_all(win, ierror)
23
         TYPE(MPI_Win), INTENT(IN) :: win
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_Win_free(win, ierror)
27
         TYPE(MPI_Win), INTENT(INOUT) :: win
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Win_get_group(win, group, ierror)
30
         TYPE(MPI_Win), INTENT(IN) :: win
31
         TYPE(MPI_Group), INTENT(OUT) :: group
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Win_get_info(win, info_used, ierror)
35
         TYPE(MPI_Win), INTENT(IN) :: win
36
         TYPE(MPI_Info), INTENT(OUT) :: info_used
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Win_lock(lock_type, rank, assert, win, ierror)
39
         INTEGER, INTENT(IN) :: lock_type, rank, assert
40
         TYPE(MPI_Win), INTENT(IN) :: win
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Win_lock_all(assert, win, ierror)
44
         INTEGER, INTENT(IN) :: assert
45
         TYPE(MPI_Win), INTENT(IN) :: win
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
    MPI_Win_post(group, assert, win, ierror)
48
```

TYPE(MPI_Group), INTENT(IN) :: group	1
INTEGER, INTENT(IN) :: assert	2
TYPE(MPI_Win), INTENT(IN) :: win	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
MPI_Win_set_info(win, info, ierror)	5
TYPE(MPI_Win), INTENT(IN) :: win	6
TYPE(MPI_Info), INTENT(IN) :: info	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8 9
	9 10
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)	10
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	12
TYPE(MPI_Win), INTENT(IN) :: win	13
INTEGER, INTENT(IN) :: rank	14
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size	15
INTEGER, INTENT(OUT) :: disp_unit TYPE(C_PTR), INTENT(OUT) :: baseptr	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
INTEGER, OFITOWAL, INTENT(001) TETTOT	18
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)	19
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	20
TYPE(MPI_Win), INTENT(IN) :: win	21
INTEGER, INTENT(IN) :: rank	22
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size, disp_unit	23
TYPE(C_PTR), INTENT(OUT) :: baseptr	24
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
MPI_Win_start(group, assert, win, ierror)	26
TYPE(MPI_Group), INTENT(IN) :: group	27
INTEGER, INTENT(IN) :: assert	28
TYPE(MPI_Win), INTENT(IN) :: win	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
	31
MPI_Win_sync(win, ierror)	32
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
INTEGER, OFFICIARE, INTENT(COT) TETTOT	34
MPI_Win_test(win, flag, ierror)	35
TYPE(MPI_Win), INTENT(IN) :: win	36
LOGICAL, INTENT(OUT) :: flag	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38 39
MPI_Win_unlock(rank, win, ierror)	40
INTEGER, INTENT(IN) :: rank	40 41
TYPE(MPI_Win), INTENT(IN) :: win	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
	44
MPI_Win_unlock_all(win, ierror)	45
TYPE(MPI_Win), INTENT(IN) :: win	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
MPI_Win_wait(win, ierror)	48

```
1
                 TYPE(MPI_Win), INTENT(IN) :: win
        2
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        3
        4
             A.4.11 External Interfaces Fortran 2008 Bindings
        5
        6
             MPI_Grequest_complete(request, ierror)
        7
                 TYPE(MPI_Request), INTENT(IN) :: request
        8
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        9
             MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
        10
                           ierror)
        11
#-error
                 PROCEDURE(MPI_Grequest_query_function), INTENT(IN) :: query_fn
        12
                 PROCEDURE(MPI_Grequest_free_function), INTENT(IN) :: free_fn
Done
        13
                 PROCEDURE(MPI_Grequest_cancel_function), INTENT(IN) :: cancel_fn
        14
PR414
                 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
        15
                 TYPE(MPI_Request), INTENT(OUT) :: request
        16
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        17
        18
             MPI_Status_set_cancelled(status, flag, ierror)
        19
                 TYPE(MPI_Status), INTENT(INOUT) :: status
        20
                 LOGICAL, INTENT(IN) :: flag
        21
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        22
             MPI_Status_set_elements(status, datatype, count, ierror)
        23
                 TYPE(MPI_Status), INTENT(INOUT) :: status
        24
                 TYPE(MPI_Datatype), INTENT(IN) :: datatype
        25
                 INTEGER, INTENT(IN) :: count
        26
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        27
        28
             MPI_Status_set_elements_x(status, datatype, count, ierror)
        29
                 TYPE(MPI_Status), INTENT(INOUT) :: status
        30
                 TYPE(MPI_Datatype), INTENT(IN) :: datatype
        31
                 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
        32
                 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
        33
        34
             A.4.12 I/O Fortran 2008 Bindings
        35
        36
             MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,
        37
                           extra_state, ierror)
        38
                 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        39
                 TYPE(C_PTR), VALUE :: userbuf, filebuf
        40
                 TYPE(MPI_Datatype) :: datatype
        41
                 INTEGER(KIND=MPI_COUNT_KIND) :: count
        42
                 INTEGER(KIND=MPI_OFFSET_KIND) :: position
        43
                 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
        44
                 INTEGER :: ierror
        45
        46
             MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,
        47
                           extra_state, ierror)
        48
```

in

1 USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR 2 TYPE(C\_PTR), VALUE :: userbuf, filebuf TYPE(MPI\_Datatype) :: datatype INTEGER :: count, ierror INTEGER(KIND=MPI\_OFFSET\_KIND) :: position 5 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state 6 MPI\_File\_close(fh, ierror) TYPE(MPI\_File), INTENT(INOUT) :: fh INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 11 MPI\_File\_delete(filename, info, ierror) CHARACTER(LEN=\*), INTENT(IN) :: filename 1213 TYPE(MPI\_Info), INTENT(IN) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror 14 15MPI\_File\_get\_amode(fh, amode, ierror) 16TYPE(MPI\_File), INTENT(IN) :: fh 17 INTEGER, INTENT(OUT) :: amode 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 20MPI\_File\_get\_atomicity(fh, flag, ierror) 21TYPE(MPI\_File), INTENT(IN) :: fh 22 LOGICAL, INTENT(OUT) :: flag 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24MPI\_File\_get\_byte\_offset(fh, offset, disp, ierror) 25TYPE(MPI\_File), INTENT(IN) :: fh 26INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 27INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(OUT) :: disp 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 MPI\_File\_get\_group(fh, group, ierror) 31TYPE(MPI\_File), INTENT(IN) :: fh 32 TYPE(MPI\_Group), INTENT(OUT) :: group 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 MPI\_File\_get\_info(fh, info\_used, ierror) 35TYPE(MPI\_File), INTENT(IN) :: fh 36 TYPE(MPI\_Info), INTENT(OUT) :: info\_used 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI\_File\_get\_position(fh, offset, ierror) 40 TYPE(MPI\_File), INTENT(IN) :: fh 41 INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(OUT) :: offset 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 MPI\_File\_get\_position\_shared(fh, offset, ierror) 44 TYPE(MPI\_File), INTENT(IN) :: fh 45INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(OUT) :: offset 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748

```
1
    MPI_File_get_size(fh, size, ierror)
\mathbf{2}
         TYPE(MPI_File), INTENT(IN) :: fh
3
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_File_get_type_extent(fh, datatype, extent, ierror)
6
         TYPE(MPI_File), INTENT(IN) :: fh
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_File_get_type_extent(fh, datatype, extent, ierror)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: extent
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
17
         TYPE(MPI_File), INTENT(IN) :: fh
18
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
19
         TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
20
         CHARACTER(LEN=*), INTENT(OUT) :: datarep
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_File_iread(fh, buf, count, datatype, request, ierror)
^{24}
         TYPE(MPI_File), INTENT(IN) :: fh
25
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
26
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         TYPE(MPI_Request), INTENT(OUT) :: request
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
     MPI_File_iread(fh, buf, count, datatype, request, ierror)
31
         TYPE(MPI_File), INTENT(IN) :: fh
32
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
33
         INTEGER, INTENT(IN) :: count
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Request), INTENT(OUT) :: request
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
39
         TYPE(MPI_File), INTENT(IN) :: fh
40
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         TYPE(MPI_Request), INTENT(OUT) :: request
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
    MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
46
         TYPE(MPI_File), INTENT(IN) :: fh
47
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
48
```

INTEGER, INTENT(IN) :: count	1
TYPE(MPI_Datatype), INTENT(IN) :: datatype	2
TYPE(MPI_Request), INTENT(OUT) :: request	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)	5
TYPE(MPI_File), INTENT(IN) :: fh	6
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset	7
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	8
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	9
TYPE(MPI_Datatype), INTENT(IN) :: datatype	10
TYPE(MPI_Request), INTENT(OUT) :: request	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
	13
MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)	14 15
TYPE(MPI_File), INTENT(IN) :: fh	15
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset	10
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	18
INTEGER, INTENT(IN) :: count	19
TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Request), INTENT(OUT) :: request	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
INTEGER, OFFICURE, INTENT(COT) TETTOT	22
<pre>MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)</pre>	23
TYPE(MPI_File), INTENT(IN) :: fh	24
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset	25
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	26
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	27
TYPE(MPI_Datatype), INTENT(IN) :: datatype	28
TYPE(MPI_Request), INTENT(OUT) :: request	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)	31
TYPE(MPI_File), INTENT(IN) :: fh	32
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset	33
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	34
INTEGER, INTENT(IN) :: count	35
TYPE(MPI_Datatype), INTENT(IN) :: datatype	36
TYPE(MPI_Request), INTENT(OUT) :: request	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)	39
TYPE(MPI_File), INTENT(IN) :: fh	40 41
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	41
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	42 43
TYPE(MPI_Datatype), INTENT(IN) :: datatype	43 44
TYPE(MPI_Request), INTENT(OUT) :: request	44 45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
	40
<pre>MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)</pre>	48

```
1
         TYPE(MPI_File), INTENT(IN) :: fh
2
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
3
         INTEGER, INTENT(IN) :: count
4
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
    MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
8
         TYPE(MPI_File), INTENT(IN) :: fh
9
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
10
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
16
         TYPE(MPI_File), INTENT(IN) :: fh
17
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
18
         INTEGER, INTENT(IN) :: count
19
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
23
         TYPE(MPI_File), INTENT(IN) :: fh
24
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
25
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
31
         TYPE(MPI_File), INTENT(IN) :: fh
32
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
33
         INTEGER, INTENT(IN) :: count
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Request), INTENT(OUT) :: request
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
40
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         TYPE(MPI_Request), INTENT(OUT) :: request
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
47
         TYPE(MPI_File), INTENT(IN) :: fh
48
```

INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 1 2 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: datatype TYPE(MPI\_Request), INTENT(OUT) :: request 5 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 6 MPI\_File\_iwrite\_at\_all(fh, offset, buf, count, datatype, request, ierror) TYPE(MPI\_File), INTENT(IN) :: fh 9 INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 10 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 11 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 12TYPE(MPI\_Datatype), INTENT(IN) :: datatype 13 TYPE(MPI\_Request), INTENT(OUT) :: request 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516MPI\_File\_iwrite\_at\_all(fh, offset, buf, count, datatype, request, ierror) 17TYPE(MPI\_File), INTENT(IN) :: fh 18 INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 19 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count 2021TYPE(MPI\_Datatype), INTENT(IN) :: datatype 22 TYPE(MPI\_Request), INTENT(OUT) :: request 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24MPI\_File\_iwrite\_shared(fh, buf, count, datatype, request, ierror) 25TYPE(MPI\_File), INTENT(IN) :: fh 26TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 27INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 28 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 29 TYPE(MPI\_Request), INTENT(OUT) :: request 30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3132 MPI\_File\_iwrite\_shared(fh, buf, count, datatype, request, ierror) 33 TYPE(MPI\_File), INTENT(IN) :: fh 34 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 35 INTEGER, INTENT(IN) :: count 36 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 37 TYPE(MPI\_Request), INTENT(OUT) :: request 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 MPI\_File\_open(comm, filename, amode, info, fh, ierror) 40 TYPE(MPI\_Comm), INTENT(IN) :: comm 41 CHARACTER(LEN=\*), INTENT(IN) :: filename 42INTEGER, INTENT(IN) :: amode 43 TYPE(MPI\_Info), INTENT(IN) :: info 44TYPE(MPI\_File), INTENT(OUT) :: fh 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4647MPI\_File\_preallocate(fh, size, ierror) 48

```
1
         TYPE(MPI_File), INTENT(IN) :: fh
2
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_File_read(fh, buf, count, datatype, status, ierror)
5
         TYPE(MPI_File), INTENT(IN) :: fh
6
         TYPE(*), DIMENSION(..) :: buf
7
         INTEGER(KIND=MPI_COUNT_KIND), 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_read(fh, buf, count, datatype, status, ierror)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         TYPE(*), DIMENSION(..) :: buf
15
         INTEGER, INTENT(IN) :: count
16
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
17
         TYPE(MPI_Status) :: status
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
     MPI_File_read_all(fh, buf, count, datatype, status, ierror)
20
         TYPE(MPI_File), INTENT(IN) :: fh
21
         TYPE(*), DIMENSION(..) :: buf
22
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Status) :: status
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     MPI_File_read_all(fh, buf, count, datatype, status, ierror)
28
         TYPE(MPI_File), INTENT(IN) :: fh
29
         TYPE(*), DIMENSION(..) :: buf
30
         INTEGER, INTENT(IN) :: count
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Status) :: status
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
35
         TYPE(MPI_File), INTENT(IN) :: fh
36
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
37
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_File_read_all_begin(fh, buf, count, datatype, 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
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
    MPI_File_read_all_end(fh, buf, status, ierror)
48
```

```
1
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  2
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
                                                                                  6
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  10
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  11
    TYPE(MPI_Status) :: status
                                                                                  12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  13
                                                                                  14
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
                                                                                  15
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  16
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  17
    TYPE(*), DIMENSION(..) :: buf
                                                                                  18
    INTEGER, INTENT(IN) :: count
                                                                                  19
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
                                                                                  20
                                                                                  21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  22
MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
                                                                                  23
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  24
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  25
    TYPE(*), DIMENSION(..) :: buf
                                                                                  26
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  28
    TYPE(MPI_Status) :: status
                                                                                  29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  30
                                                                                  31
MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
                                                                                  32
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  33
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  34
    TYPE(*), DIMENSION(..) :: buf
                                                                                  35
    INTEGER, INTENT(IN) :: count
                                                                                  36
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  37
    TYPE(MPI_Status) :: status
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
MPI_File_read_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(...), ASYNCHRONOUS :: buf
                                                                                  43
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
                                                                                  47
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
                                                                                  48
```

```
1
         TYPE(MPI_File), INTENT(IN) :: fh
2
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
3
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
4
         INTEGER, INTENT(IN) :: count
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_File_read_at_all_end(fh, buf, status, ierror)
8
         TYPE(MPI_File), INTENT(IN) :: fh
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
10
         TYPE(MPI_Status) :: status
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
14
         TYPE(MPI_File), INTENT(IN) :: fh
15
         TYPE(*), DIMENSION(..) :: buf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Status) :: status
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
21
         TYPE(MPI_File), INTENT(IN) :: fh
22
         TYPE(*), DIMENSION(..) :: buf
23
         INTEGER, INTENT(IN) :: count
24
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
         TYPE(MPI_Status) :: status
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
29
         TYPE(MPI_File), INTENT(IN) :: fh
30
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
31
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
32
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
35
         TYPE(MPI_File), INTENT(IN) :: fh
36
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
37
         INTEGER, INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_File_read_ordered_end(fh, buf, status, ierror)
42
         TYPE(MPI_File), INTENT(IN) :: fh
43
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
44
         TYPE(MPI_Status) :: status
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
47
         TYPE(MPI_File), INTENT(IN) :: fh
48
```

```
1
    TYPE(*), DIMENSION(..) :: buf
                                                                                   2
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
                                                                                   6
MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
                                                                                   10
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   11
    TYPE(MPI_Status) :: status
                                                                                   12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   13
                                                                                   14
MPI_File_seek(fh, offset, whence, ierror)
                                                                                   15
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   16
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                   17
    INTEGER, INTENT(IN) :: whence
                                                                                   18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   19
MPI_File_seek_shared(fh, offset, whence, ierror)
                                                                                   20
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   21
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                   22
    INTEGER, INTENT(IN) :: whence
                                                                                   23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   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_info(fh, info, ierror)
                                                                                   30
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   31
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   33
                                                                                   34
MPI_File_set_size(fh, size, ierror)
                                                                                   35
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   36
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
                                                                                   37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   38
MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
                                                                                   39
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   40
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
                                                                                   41
    TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
                                                                                   42
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                   43
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   45
                                                                                   46
MPI_File_sync(fh, ierror)
                                                                                   47
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                   48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_File_write(fh, buf, count, datatype, status, ierror)
3
         TYPE(MPI_File), INTENT(IN) :: fh
4
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
5
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
6
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
         TYPE(MPI_Status) :: status
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
     MPI_File_write(fh, buf, count, datatype, status, ierror)
11
         TYPE(MPI_File), INTENT(IN) :: fh
12
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
13
         INTEGER, INTENT(IN) :: count
14
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15
         TYPE(MPI_Status) :: status
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_File_write_all(fh, buf, count, datatype, status, ierror)
18
         TYPE(MPI_File), INTENT(IN) :: fh
19
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Status) :: status
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25
     MPI_File_write_all(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_all_begin(fh, buf, count, datatype, ierror)
33
         TYPE(MPI_File), INTENT(IN) :: fh
34
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
35
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
40
         TYPE(MPI_File), INTENT(IN) :: fh
41
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
42
         INTEGER, INTENT(IN) :: count
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
    MPI_File_write_all_end(fh, buf, status, ierror)
46
         TYPE(MPI_File), INTENT(IN) :: fh
47
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
48
```

1 TYPE(MPI\_Status) :: status 2 INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_File\_write\_at(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI\_File), INTENT(IN) :: fh INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 6 TYPE(\*), DIMENSION(...), INTENT(IN) :: buf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: datatype 9 TYPE(MPI\_Status) :: status 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 MPI\_File\_write\_at(fh, offset, buf, count, datatype, status, ierror) 1213 TYPE(MPI\_File), INTENT(IN) :: fh 14INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 15TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 16INTEGER, INTENT(IN) :: count 17TYPE(MPI\_Datatype), INTENT(IN) :: datatype 18 TYPE(MPI\_Status) :: status 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI\_File\_write\_at\_all(fh, offset, buf, count, datatype, status, ierror) 21TYPE(MPI\_File), INTENT(IN) :: fh 22 INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 23TYPE(\*), DIMENSION(..), INTENT(IN) :: buf 24INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 25TYPE(MPI\_Datatype), INTENT(IN) :: datatype 26TYPE(MPI\_Status) :: status 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28 29MPI\_File\_write\_at\_all(fh, offset, buf, count, datatype, status, ierror) 30 TYPE(MPI\_File), INTENT(IN) :: fh 31INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 32 TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 33 INTEGER, INTENT(IN) :: count 34 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 35 TYPE(MPI\_Status) :: status 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 MPI\_File\_write\_at\_all\_begin(fh, offset, buf, count, datatype, ierror) 38 TYPE(MPI\_File), INTENT(IN) :: fh 39 INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 40 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 41 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 42TYPE(MPI\_Datatype), INTENT(IN) :: datatype 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4445MPI\_File\_write\_at\_all\_begin(fh, offset, buf, count, datatype, ierror) 46TYPE(MPI\_File), INTENT(IN) :: fh 47INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 48

```
1
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
2
         INTEGER, INTENT(IN) :: count
3
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_File_write_at_all_end(fh, buf, status, ierror)
6
         TYPE(MPI_File), INTENT(IN) :: fh
7
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
8
         TYPE(MPI_Status) :: status
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
14
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
15
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
         TYPE(MPI_Status) :: status
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
19
         TYPE(MPI_File), INTENT(IN) :: fh
20
         TYPE(*), DIMENSION(...), INTENT(IN) :: 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_write_ordered_begin(fh, buf, count, datatype, ierror)
27
         TYPE(MPI_File), INTENT(IN) :: fh
28
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
29
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
30
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
    MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
33
         TYPE(MPI_File), INTENT(IN) :: fh
34
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
35
         INTEGER, INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_File_write_ordered_end(fh, buf, status, ierror)
40
         TYPE(MPI_File), INTENT(IN) :: fh
41
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
42
         TYPE(MPI_Status) :: status
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
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(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                     2
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
                                                                                    10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    11
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
                                                                                    12
                                                                                    13
              dtype_file_extent_fn, extra_state, ierror)
                                                                                    14
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                    15
    PROCEDURE(MPI_Datarep_conversion_function), INTENT(IN) ::
                                                                                      #-error!
                                                                                    16
               read_conversion_fn, write_conversion_fn
                                                                                    <sup>17</sup> #-okay?
    PROCEDURE(MPI_Datarep_extent_function) /: dtype_file_extent_fn
                                                                                    18 Or
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                    19 #-error?
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       Done
                                                                                    20
MPI_Register_datarep_c(datarep, read_conversion_fn, write_conversion_fn,
                                                                                       lin
                                                                                    21
                                                                                       PR414
              dtype_file_extent_fn, extra_state, ierror)
                                                                                    22
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                    23
    PROCEDURE(MPI_Datarep_conversion_function_c), INTENT(IN) ::
                                                                                    <sup>24</sup> #-error!
               read_conversion_fn, write_conversion_fn
                                                                                    25
    PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
                                                                                      #-okay?
                                                                                    26
                                                                                       or
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                    27
                                                                                      #-error?
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    28
                                                                                        Done
                                                    Pythonization group will fix this globally!!!
                                                                                        in
                                                     -> Dan.
                                                                                        PR414
A.4.13 Language Bindings Fortran 2008 Bindings
                                                                                    31
MPI_F_sync_reg(buf)
                                                                                    32
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                    33
                                                                                    34
MPI_Status_f082f(f08_status, f_status, ierror)
                                                                                    35
    TYPE(MPI_Status), INTENT(IN) :: f08_status
                                                                                    36
    INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
                                                                                    37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    38
MPI_Status_f2f08(f_status, f08_status, ierror)
                                                                                    39
    INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
                                                                                    40
    TYPE(MPI_Status), INTENT(OUT) :: f08_status
                                                                                    41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    42
                                                                                    43
MPI_Type_create_f90_complex(p, r, newtype, ierror)
                                                                                    44
    INTEGER, INTENT(IN) :: p, r
                                                                                    45
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                    46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    47
MPI_Type_create_f90_integer(r, newtype, ierror)
                                                                                    48
```

```
1
         INTEGER, INTENT(IN) :: r
2
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Type_create_f90_real(p, r, newtype, ierror)
5
         INTEGER, INTENT(IN) :: p, r
6
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Type_match_size(typeclass, size, datatype, ierror)
10
         INTEGER, INTENT(IN) :: typeclass, size
11
         TYPE(MPI_Datatype), INTENT(OUT) :: datatype
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     A.4.14 Tools / Profiling Interface Fortran 2008 Bindings
15
16
     MPI Pcontrol(level)
17
         INTEGER, INTENT(IN) :: level
18
19
     A.4.15 Deprecated Fortran 2008 Bindings
20
21
     MPI_Info_get(info, key, valuelen, value, flag, ierror)
22
         TYPE(MPI_Info), INTENT(IN) :: info
23
         CHARACTER(LEN=*), INTENT(IN) :: key
24
         INTEGER, INTENT(IN) :: valuelen
25
         CHARACTER(LEN=valuelen), INTENT(OUT) :: value
26
         LOGICAL, INTENT(OUT) :: flag
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
29
30
         TYPE(MPI_Info), INTENT(IN) :: info
31
         CHARACTER(LEN=*), INTENT(IN) :: key
32
         INTEGER, INTENT(OUT) :: valuelen
33
         LOGICAL, INTENT(OUT) :: flag
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Sizeof(x, size, ierror)
36
         TYPE(*), DIMENSION(..) :: x
37
         INTEGER, INTENT(OUT) :: size
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
41
42
43
44
45
46
47
48
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	995
A.5 Fortran Bindings with mpif.h or the mpi Module	1
A.5.1 Point-to-Point Communication Fortran Bindings	3
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	4
<type> BUF(*)</type>	5
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	7
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	8
<type> BUF(*)</type>	9
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	10 11
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)	11
<type> BUFFER(*) INTEGER SIZE, IERROR</type>	13
	14
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)	15 16
<type> BUFFER_ADDR(*) INTEGER SIZE, IERROR</type>	10
	18
MPI_CANCEL(REQUEST, IERROR) INTEGER REQUEST, IERROR	19
	20 21
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	21
	23
<pre>MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre>	24
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	25 26
MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)	20
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	28
LOGICAL FLAG	29
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)	30 31
<type> BUF(*)</type>	32
INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR	33
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)	34
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	35 36
LOGICAL FLAG	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
<pre>MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre>	45 46
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	40 47
	48

1 MPI\_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,  $\mathbf{2}$ RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR) 3 <type> SENDBUF(\*), RECVBUF(\*) 4 INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, 5SOURCE, RECVTAG, COMM, REQUEST, IERROR 6 MPI\_ISENDRECV\_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, 7 COMM, REQUEST, IERROR) 8 <type> BUF(\*) 9 INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST, 10 IERROR 11 12MPI\_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 13<type> BUF(\*) 14INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 15MPI\_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR) 16INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 1718MPI\_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR) 19<type> BUF(\*) 20INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 21MPI\_PROBE(SOURCE, TAG, COMM, STATUS, IERROR) 22 INTEGER SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR 23 $^{24}$ MPI\_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) 25<type> BUF(\*) 26INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), 27IERROR 28MPI\_RECV\_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 29 <type> BUF(\*) 30 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR  $^{31}$ 32 MPI\_REQUEST\_FREE(REQUEST, IERROR) 33 INTEGER REQUEST, IERROR 34 MPI\_REQUEST\_GET\_STATUS (REQUEST, FLAG, STATUS, IERROR) 35 INTEGER REQUEST, STATUS(MPI\_STATUS\_SIZE), IERROR 36 LOGICAL FLAG 37 38 MPI\_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 39 <type> BUF(\*) 40INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 41 MPI\_RSEND\_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 42<type> BUF(\*) 43 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 4445MPI\_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 46<type> BUF(\*) 47INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 48

MPI\_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, 1 RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR) 2 <type> SENDBUF(\*), RECVBUF(\*) INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, 4 SOURCE, RECVTAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR 5MPI\_SENDRECV\_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> BUF(\*) 9 INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, 10 STATUS(MPI\_STATUS\_SIZE), IERROR 11 MPI\_SEND\_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 1213 <type> BUF(\*) 14INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 15MPI\_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 16<type> BUF(\*) 17INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 18 19 MPI\_SSEND\_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 20<type> BUF(\*) 21INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 22 MPI\_START(REQUEST, IERROR) 23INTEGER REQUEST, IERROR 2425MPI\_STARTALL(COUNT, ARRAY\_OF\_REQUESTS, IERROR) 26INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), IERROR 27MPI\_TEST(REQUEST, FLAG, STATUS, IERROR) 28 INTEGER REQUEST, STATUS(MPI\_STATUS\_SIZE), IERROR 29LOGICAL FLAG 30 31MPI\_TESTALL(COUNT, ARRAY OF REQUESTS, FLAG, ARRAY OF STATUSES, IERROR) 32 INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), ARRAY\_OF\_STATUSES(MPI\_STATUS\_SIZE, 33 \*), IERROR 34 LOGICAL FLAG 35 MPI\_TESTANY(COUNT, ARRAY\_OF\_REQUESTS, INDEX, FLAG, STATUS, IERROR) 36 INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), INDEX, STATUS(MPI\_STATUS\_SIZE), 37 IERROR 38 LOGICAL FLAG 39 40 MPI\_TESTSOME(INCOUNT, ARRAY\_OF\_REQUESTS, OUTCOUNT, ARRAY\_OF\_INDICES, 41 ARRAY\_OF\_STATUSES, IERROR) 42INTEGER INCOUNT, ARRAY\_OF\_REQUESTS(\*), OUTCOUNT, ARRAY\_OF\_INDICES(\*), 43 ARRAY\_OF\_STATUSES(MPI\_STATUS\_SIZE, \*), IERROR 44 MPI\_TEST\_CANCELLED(STATUS, FLAG, IERROR) 45INTEGER STATUS(MPI\_STATUS\_SIZE), IERROR 46LOGICAL FLAG 4748

```
1
     MPI_WAIT(REQUEST, STATUS, IERROR)
\mathbf{2}
         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
3
     MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
4
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,
5
                    *), IERROR
6
\overline{7}
     MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
8
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
9
                    IERROR
10
     MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
11
                   ARRAY_OF_STATUSES, IERROR)
12
         INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
13
                    ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
14
15
16
     A.5.2 Partitioned Communication Fortran Bindings
17
     MPI_PARRIVED(REQUEST, PARTITION, FLAG, IERROR)
18
         INTEGER REQUEST, PARTITION, IERROR
19
         LOGICAL FLAG
20
21
     MPI_PREADY(PARTITION, REQUEST, IERROR)
22
         INTEGER PARTITION, REQUEST, IERROR
23
     MPI_PREADY_LIST(LENGTH, ARRAY_OF_PARTITIONS, REQUEST, IERROR)
^{24}
         INTEGER LENGTH, ARRAY_OF_PARTITIONS(*), REQUEST, IERROR
25
26
     MPI_PREADY_RANGE(PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR)
27
         INTEGER PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR
28
     MPI_PRECV_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,
29
30
                   REQUEST, IERROR)
31
         <type> BUF(*)
         INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR
32
33
         INTEGER(KIND=MPI_COUNT_KIND) COUNT
34
     MPI_PSEND_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,
35
                   REQUEST, IERROR)
36
         <type> BUF(*)
37
         INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR
38
         INTEGER(KIND=MPI_COUNT_KIND) COUNT
39
40
41
     A.5.3 Datatypes Fortran Bindings
42
     INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP)
43
         INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP
44
45
     INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)
46
         INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2
47
     MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
48
```

<type> LOCATION(*) INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS</type>	1 2
INTEGER IERROR	3
	4
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)	5
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	6
MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)	7
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR	8
INTEGER(KIND=MPI_COUNT_KIND) COUNT	9
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)	10 11
<type> INBUF(*), OUTBUF(*)</type>	12
INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR	13
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	14
POSITION, IERROR)	15
CHARACTER*(*) DATAREP	16
<type> INBUF(*), OUTBUF(*)</type>	17
INTEGER INCOUNT, DATATYPE, IERROR	18
INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION	19
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)	20
CHARACTER*(*) DATAREP	21 22
INTEGER INCOUNT, DATATYPE, IERROR	22
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE	20
NDI DAGU GIZE (INGOUNT DATATUDE GOMA GIZE IEDDOD)	25
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR) INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR	26
INTEGER INCOUNT, DATATIFE, COMP, SIZE, TERROR	27
MPI_TYPE_COMMIT(DATATYPE, IERROR)	28
INTEGER DATATYPE, IERROR	29
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)	30
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR	31
NDT TYPE OPENTE DADDAY OTTE DANK NDIMO ADDAY OF COTTED	32
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES, ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,	33 34
OLDTYPE, NEWTYPE, IERROR)	34 35
INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),	36
ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE,	37
NEWTYPE, IERROR	38
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,	39
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)	40
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR	41
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	42
	43
MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	44
OLDTYPE, NEWTYPE, IERROR)	45
INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	46 47
THIEREW(WIND-HIT_KDDHEDD_WIND) KUNKI_OL_DIDLEKOEHENID(*)	47
	10

1 2	MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
3	IERROR) INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
4	INTEGER (KIND=MPI_ADDRESS_KIND) STRIDE
5	
6	MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
7	OLDTYPE, NEWTYPE, IERROR)
8	INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
9	NEWTYPE, IERROR
10	MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
11	INTEGER OLDTYPE, NEWTYPE, IERROR
12	INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
13 14	MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
14	ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
16	INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
17	IERROR
18	<pre>INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)</pre>
19	MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
20	ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
21	INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
22	ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
23	
24	MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR
25	
26 27	MPI_TYPE_FREE(DATATYPE, IERROR)
28	INTEGER DATATYPE, IERROR
29	MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
30	ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
31	IERROR)
32	INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
33	ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
34	INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
35	MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
36	COMBINER, IERROR)
37	INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
38 39	IERROR
40	MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
40	INTEGER DATATYPE, IERROR
42	INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
43	
44	MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
45	INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT
46	INIEGEN(KIND-MII_GOUNI_KIND) ED, EXIENI
47	MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
48	INTEGER DATATYPE, IERROR

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1001
INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT	
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT	
<pre>MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR</pre>	
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, SIZE, IERROR	
MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) SIZE	
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, IERROR) <type> INBUF(*), OUTBUF(*) INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR</type>	
<pre>MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, IERROR) CHARACTER*(*) DATAREP <type> INBUF(*), OUTBUF(*) INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION INTEGER OUTCOUNT, DATATYPE, IERROR</type></pre>	
A.5.4 Collective Communication Fortran Bindings	
<pre>MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR</type></pre>	Ε,
<pre>MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPL RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, CON IERROR</type></pre>	
<pre>MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COM INFO, REQUEST, IERROR</type></pre>	ММ,

```
1002
```

1MPI\_ALLGATHER\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,  $\mathbf{2}$ RECVTYPE, COMM, INFO, REQUEST, IERROR) 3 <type> SENDBUF(\*), RECVBUF(\*) 4 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 5IERROR 6 MPI ALLREDUCE (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 7 <type> SENDBUF(\*), RECVBUF(\*) 8 INTEGER COUNT, DATATYPE, OP, COMM, IERROR 9 10 MPI\_ALLREDUCE\_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, 11 REQUEST, IERROR) 12<type> SENDBUF(\*), RECVBUF(\*) 13INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 14MPI\_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 15COMM, IERROR) 16<type> SENDBUF(\*), RECVBUF(\*) 17INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 18 19 MPI\_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, 20RDISPLS, RECVTYPE, COMM, IERROR) 21<type> SENDBUF(\*), RECVBUF(\*) 22INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPE, RECVCOUNTS(\*), RDISPLS(\*), 23RECVTYPE, COMM, IERROR 24MPI\_ALLTOALLV\_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 25RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 26<type> SENDBUF(\*), RECVBUF(\*) 27INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPE, RECVCOUNTS(\*), RDISPLS(\*), 28RECVTYPE, COMM, INFO, REQUEST, IERROR 29 30 MPI\_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, 31RDISPLS, RECVTYPES, COMM, IERROR) 32 <type> SENDBUF(\*), RECVBUF(\*) 33 INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPES(\*), RECVCOUNTS(\*), 34 RDISPLS(\*), RECVTYPES(\*), COMM, IERROR 35 MPI\_ALLTOALLW\_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 36 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR) 37 <type> SENDBUF(\*), RECVBUF(\*) 38 INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPES(\*), RECVCOUNTS(\*), 39 RDISPLS(\*), RECVTYPES(\*), COMM, INFO, REQUEST, IERROR 4041 MPI\_ALLTOALL\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 42RECVTYPE, COMM, INFO, REQUEST, IERROR) 43 <type> SENDBUF(\*), RECVBUF(\*) 44INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 45IERROR 46MPI\_BARRIER(COMM, IERROR) 47INTEGER COMM, IERROR 48

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE 1	.003
MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR) INTEGER COMM, INFO, REQUEST, IERROR	1 2
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) <type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR</type>	3 4 5 6
MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERRON <type> BUFFER(*) INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR</type>	8 9
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, IERROR</type>	10 11 12 13
<pre>MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUES IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR</type></pre>	ST, <sup>14</sup> 15 16 17 18
<pre>MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR</type></pre>	18 19 20 21 22
<pre>MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,</pre>	27 28 LS, 29 30 31
<pre>MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYL ROOT, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR</type></pre>	PE, 34 35 36 37 38 39
<pre>MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERD</type></pre>	<b>E</b> , 40 41 42
MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPL RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>	44 45 46 47 48

1	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR
3 4 5	MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
6 7	<type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR</type>
8 9 10 11	<pre>MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR</type></pre>
12 13 14 15 16 17	<pre>MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,</pre>
18 19 20 21 22	<pre>MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>
23 24 25	MPI_IBARRIER(COMM, REQUEST, IERROR) INTEGER COMM, REQUEST, IERROR
26 27 28 29	<pre>MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)</pre>
30 31 32	<pre>MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)</pre>
33 34 35 36 37 38	<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>
39 40 41 42 43	<pre>MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,</pre>
44 45 46 47 48	<pre>MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR</type></pre>

MPI\_IREDUCE\_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 1 REQUEST, IERROR) 2 <type> SENDBUF(\*), RECVBUF(\*) INTEGER RECVCOUNTS(\*), DATATYPE, OP, COMM, REQUEST, IERROR MPI\_IREDUCE\_SCATTER\_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 6 REQUEST, IERROR) <type> SENDBUF(\*), RECVBUF(\*) INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR 9 10 MPI\_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 11 <type> SENDBUF(\*), RECVBUF(\*) INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 1213 MPI\_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 14ROOT, COMM, REQUEST, IERROR) 15<type> SENDBUF(\*), RECVBUF(\*) 16INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, 17IERROR 18 19 MPI\_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 20RECVTYPE, ROOT, COMM, REQUEST, IERROR) 21<type> SENDBUF(\*), RECVBUF(\*) 22 INTEGER SENDCOUNTS(\*), DISPLS(\*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 23COMM, REQUEST, IERROR 24MPI\_OP\_COMMUTATIVE(OP, COMMUTE, IERROR) 25INTEGER OP, IERROR 26LOGICAL COMMUTE 2728 MPI\_OP\_CREATE(USER\_FN, COMMUTE, OP, IERROR) 29 EXTERNAL USER\_FN 30 LOGICAL COMMUTE 31INTEGER OP, IERROR 32 MPI\_OP\_FREE(OP, IERROR) 33 INTEGER OP, IERROR 34 35 MPI\_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR) 36 <type> SENDBUF(\*), RECVBUF(\*) 37 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR 38 MPI\_REDUCE\_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO, 39 REQUEST, IERROR) 40 <type> SENDBUF(\*), RECVBUF(\*) 41 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR 4243 MPI\_REDUCE\_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) 44<type> INBUF(\*), INOUTBUF(\*) 45INTEGER COUNT, DATATYPE, OP, IERROR 46MPI\_REDUCE\_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 47IERROR) 48

1 <type> SENDBUF(\*), RECVBUF(\*)  $\mathbf{2}$ INTEGER RECVCOUNTS(\*), DATATYPE, OP, COMM, IERROR 3 MPI\_REDUCE\_SCATTER\_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 4 IERROR) 5<type> SENDBUF(\*), RECVBUF(\*) 6 INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR 7 8 MPI\_REDUCE\_SCATTER\_BLOCK\_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, 9 COMM, INFO, REQUEST, IERROR) 10 <type> SENDBUF(\*), RECVBUF(\*) 11 INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 12MPI\_REDUCE\_SCATTER\_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 13 INFO, REQUEST, IERROR) 14 <type> SENDBUF(\*), RECVBUF(\*) 15INTEGER RECVCOUNTS(\*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR 1617MPI\_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 18 <type> SENDBUF(\*), RECVBUF(\*) 19 INTEGER COUNT, DATATYPE, OP, COMM, IERROR 20MPI\_SCAN\_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, 21IERROR) 22 <type> SENDBUF(\*), RECVBUF(\*) 23INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 2425MPI\_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 26ROOT, COMM, IERROR) 27<type> SENDBUF(\*), RECVBUF(\*) 28INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR 29 MPI\_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 30 RECVTYPE, ROOT, COMM, IERROR) 31<type> SENDBUF(\*), RECVBUF(\*) 32 INTEGER SENDCOUNTS(\*), DISPLS(\*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 33 COMM, IERROR 34 35 MPI\_SCATTERV\_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, 36 RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) 37 <type> SENDBUF(\*), RECVBUF(\*) 38 INTEGER SENDCOUNTS(\*), DISPLS(\*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 39 COMM, INFO, REQUEST, IERROR 40MPI\_SCATTER\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 41 RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) 42<type> SENDBUF(\*), RECVBUF(\*) 43 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, 44 REQUEST, IERROR 4546 47 48

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE 10	007
A.5.5 Groups, Contexts, Communicators, and Caching Fortran Bindings	1
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)	2
INTEGER COMM1, COMM2, RESULT, IERROR	3 4
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)	5
INTEGER COMM, GROUP, NEWCOMM, IERROR	6
MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM,	7
IERROR)	9
INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR CHARACTER*(*) STRINGTAG	10
	11 12
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR	13
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVA	
EXTRA_STATE, IERROR)	16
EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN INTEGER COMM_KEYVAL, IERROR	17
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	18 19
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)	20
INTEGER COMM, COMM_KEYVAL, IERROR	21
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)	22
INTEGER COMM, NEWCOMM, IERROR	23 24
MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	25
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	26
INTEGER OLDCOMM, COMM_KEYVAL, IERROR	27
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	28 29
ATTRIBUTE_VAL_OUT LOGICAL FLAG	30
	31
MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)	32
INTEGER COMM, INFO, NEWCOMM, IERROR	33 34
MPI_COMM_FREE(COMM, IERROR)	34
INTEGER COMM, IERROR	36
MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)	37
INTEGER COMM_KEYVAL, IERROR	38
MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	39 40
INTEGER COMM, COMM_KEYVAL, IERROR	40
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	42
LOGICAL FLAG	43
MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)	44
INTEGER COMM, INFO_USED, IERROR	45 46
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)	40 47
INTEGER COMM, RESULTLEN, IERROR	48

1	CHARACTER*(*) COMM_NAME
2 3 4	MPI_COMM_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR
5 6 7	MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR) INTEGER COMM, NEWCOMM, REQUEST, IERROR
8 9	MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR) INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR
10 11 12 13	<pre>MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>
14 15 16	ATTRIBUTE_VAL_OUT LOGICAL FLAG
17 18 19 20	MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
21 22 23	MPI_COMM_RANK(COMM, RANK, IERROR) INTEGER COMM, RANK, IERROR
24 25	MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR
26 27 28	MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR
29 30 31	MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
32 33 34	MPI_COMM_SET_INFO(COMM, INFO, IERROR) INTEGER COMM, INFO, IERROR
35 36 37	MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR) INTEGER COMM, IERROR CHARACTER*(*) COMM_NAME
38 39 40	MPI_COMM_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR
41 42	MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR) INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
43 44 45	MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR) INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
46 47 48	MPI_COMM_TEST_INTER(COMM, FLAG, IERROR) INTEGER COMM, IERROR LOGICAL FLAG

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1009
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR) INTEGER GROUP1, GROUP2, RESULT, IERROR	1 2
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	3 4 5
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR) INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	6 7 8
MPI_GROUP_FREE(GROUP, IERROR) INTEGER GROUP, IERROR	9 10
MPI_GROUP_FROM_SESSION_PSET(SESSION, PSET_NAME, NEWGROUP, IERROR) INTEGER SESSION, NEWGROUP, IERROR CHARACTER*(*) PSET_NAME	11 12 13 14
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR) INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	14 15 16
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	17 18 19
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR) INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR	20 21
<pre>MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR) INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR</pre>	22 23 24
MPI_GROUP_RANK(GROUP, RANK, IERROR) INTEGER GROUP, RANK, IERROR	25 26
MPI_GROUP_SIZE(GROUP, SIZE, IERROR) INTEGER GROUP, SIZE, IERROR	27 28 29
<pre>MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR) INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR</pre>	30 31
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	32 33 34
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADE TAG, NEWINTERCOMM, IERROR) INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG, NEWINTERCOMM, IERROR	36 37 38
MPI_INTERCOMM_CREATE_FROM_GROUPS(LOCAL_GROUP, LOCAL_LEADER, REMOTE_GRO REMOTE_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM, IERROR)	40 41 42
INTEGER LOCAL_GROUP, LOCAL_LEADER, REMOTE_GROUP, REMOTE_LEADER, IN ERRHANDLER, NEWINTERCOMM, IERROR CHARACTER*(*) STRINGTAG	
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) INTEGER INTERCOMM, NEWINTRACOMM, IERROR LOGICAL HIGH	46 47 48

1MPI\_TYPE\_CREATE\_KEYVAL(TYPE\_COPY\_ATTR\_FN, TYPE\_DELETE\_ATTR\_FN, TYPE\_KEYVAL,  $\mathbf{2}$ EXTRA\_STATE, IERROR) 3 EXTERNAL TYPE\_COPY\_ATTR\_FN, TYPE\_DELETE\_ATTR\_FN 4 INTEGER TYPE\_KEYVAL, IERROR 5INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTRA\_STATE 6 MPI TYPE DELETE ATTR(DATATYPE, TYPE KEYVAL, IERROR) 7 INTEGER DATATYPE, TYPE\_KEYVAL, IERROR 8 9 MPI\_TYPE\_DUP\_FN(OLDTYPE, TYPE\_KEYVAL, EXTRA\_STATE, ATTRIBUTE\_VAL\_IN, 10ATTRIBUTE\_VAL\_OUT, FLAG, IERROR) 11 INTEGER OLDTYPE, TYPE\_KEYVAL, IERROR 12INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTRA\_STATE, ATTRIBUTE\_VAL\_IN, 13ATTRIBUTE\_VAL\_OUT 14LOGICAL FLAG 15MPI\_TYPE\_FREE\_KEYVAL(TYPE\_KEYVAL, IERROR) 16INTEGER TYPE\_KEYVAL, IERROR 17 $^{18}$ MPI\_TYPE\_GET\_ATTR(DATATYPE, TYPE\_KEYVAL, ATTRIBUTE\_VAL, FLAG, IERROR) 19INTEGER DATATYPE, TYPE\_KEYVAL, IERROR 20INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL 21LOGICAL FLAG 22MPI\_TYPE\_GET\_NAME(DATATYPE, TYPE\_NAME, RESULTLEN, IERROR) 23INTEGER DATATYPE, RESULTLEN, IERROR 24CHARACTER\*(\*) TYPE\_NAME 2526MPI\_TYPE\_NULL\_COPY\_FN(OLDTYPE, TYPE\_KEYVAL, EXTRA\_STATE, ATTRIBUTE\_VAL\_IN, 27ATTRIBUTE\_VAL\_OUT, FLAG, IERROR) 28 INTEGER OLDTYPE, TYPE\_KEYVAL, IERROR 29INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTRA\_STATE, ATTRIBUTE\_VAL\_IN, 30 ATTRIBUTE\_VAL\_OUT 31LOGICAL FLAG 32 MPI\_TYPE\_NULL\_DELETE\_FN(DATATYPE, TYPE\_KEYVAL, ATTRIBUTE\_VAL, EXTRA\_STATE, 33 IERROR) 34 INTEGER DATATYPE, TYPE\_KEYVAL, IERROR 35INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL, EXTRA\_STATE 36 37 MPI\_TYPE\_SET\_ATTR(DATATYPE, TYPE\_KEYVAL, ATTRIBUTE\_VAL, IERROR) 38 INTEGER DATATYPE, TYPE\_KEYVAL, IERROR 39INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL 40MPI\_TYPE\_SET\_NAME(DATATYPE, TYPE\_NAME, IERROR) 41 INTEGER DATATYPE, IERROR 42CHARACTER\*(\*) TYPE\_NAME 43 44MPI\_WIN\_CREATE\_KEYVAL(WIN\_COPY\_ATTR\_FN, WIN\_DELETE\_ATTR\_FN, WIN\_KEYVAL, 45EXTRA\_STATE, IERROR) 46EXTERNAL WIN\_COPY\_ATTR\_FN, WIN\_DELETE\_ATTR\_FN 47INTEGER WIN\_KEYVAL, IERROR 48

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1011
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	1
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)	2
INTEGER WIN, WIN_KEYVAL, IERROR	4
MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	5
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	6 7
INTEGER OLDWIN, WIN_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	8
ATTRIBUTE_VAL_OUT	9
LOGICAL FLAG	10
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)	11 12
INTEGER WIN_KEYVAL, IERROR	13
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	14
INTEGER WIN, WIN_KEYVAL, IERROR	15 16
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL LOGICAL FLAG	10
	18
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR) INTEGER WIN, RESULTLEN, IERROR	19
CHARACTER*(*) WIN_NAME	20 21
MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN	
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	<b>,</b> 23
INTEGER OLDWIN, WIN_KEYVAL, IERROR	24
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	25 26
ATTRIBUTE_VAL_OUT LOGICAL FLAG	27
	28
MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IE INTEGER WIN, WIN_KEYVAL, IERROR	<b>KKUR)</b> 29 30
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	30
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)	32
INTEGER WIN, WIN_KEYVAL, IERROR	33
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	34 35
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)	36
INTEGER WIN, IERROR	37
CHARACTER*(*) WIN_NAME	38
	39 40
A.5.6 Process Topologies Fortran Bindings	41
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)	42
INTEGER COMM, NDIMS, IERROR	43 44
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)	44 45
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR	46
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IE	
	48

1	
1 2	INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR LOGICAL PERIODS(*), REORDER
3	
4	MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
5	INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
6	LOGICAL PERIODS(*)
7	MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
8	INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
9	LOGICAL PERIODS(*)
10	MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
11	INTEGER COMM, COORDS(*), RANK, IERROR
12	
13	MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
14 15	INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
15	MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
17	INTEGER COMM, NEWCOMM, IERROR
18	LOGICAL REMAIN_DIMS(*)
19	MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
20	INTEGER NNODES, NDIMS, DIMS, TERROR
21	
22	MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
23	INFO, REORDER, COMM_DIST_GRAPH, IERROR)
24	INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
25	WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR LOGICAL REORDER
26	LUGICAL REORDER
27 28	MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
29	OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
30	COMM_DIST_GRAPH, IERROR)
31	<pre>INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH,</pre>
32	IERROR
33	LOGICAL REORDER
34	
35	MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
36	MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
37	INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
38	<pre>DESTINATIONS(*), DESTWEIGHTS(*), IERROR</pre>
39	MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
40 41	INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
41	LOGICAL WEIGHTED
43	MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
44	INTEGER COMM, NNODES, NEDGES, IERROR
45	
46	MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
47	IERROR) INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
48	INIEGEN CUTTLUED, NNUDES, INDER(*), EDGES(*), CUTTLGRAFT, IERRUR

LOGICAL REORDER

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR) INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR	2 3 4
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR	5
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR) INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR	7 8 9
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR) INTEGER COMM, RANK, NNEIGHBORS, IERROR	10 11
<pre>MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR</type></pre>	12 13 14 15 16
<pre>MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,</pre>	17 18 19 20 21
<pre>MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR</type></pre>	22 23 24 25 26
<pre>MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,</pre>	27 28 29 30 31 32
<pre>MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,</pre>	32 33 34 35 36 37 38
MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR</type>	39 40 41 42 43
<pre>MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, IERROR</type></pre>	43 44 45 46 47 48

Unofficial Draft for Comment Only

1MPI\_NEIGHBOR\_ALLGATHERV\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,  $\mathbf{2}$ RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 3 <type> SENDBUF(\*), RECVBUF(\*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 4 5INFO, REQUEST, IERROR 6 MPI NEIGHBOR ALLGATHER INIT (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 7 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) 8 <type> SENDBUF(\*), RECVBUF(\*) 9 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 10 IERROR 11 12MPI\_NEIGHBOR\_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 13RECVTYPE, COMM, IERROR) 14<type> SENDBUF(\*), RECVBUF(\*) 15INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 16MPI\_NEIGHBOR\_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 17RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR) 18 <type> SENDBUF(\*), RECVBUF(\*) 19 INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPE, RECVCOUNTS(\*), RDISPLS(\*), 20RECVTYPE, COMM, IERROR 2122MPI\_NEIGHBOR\_ALLTOALLV\_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, 23RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, 24IERROR) 25<type> SENDBUF(\*), RECVBUF(\*) 26INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPE, RECVCOUNTS(\*), RDISPLS(\*), 27RECVTYPE, COMM, INFO, REQUEST, IERROR 28MPI\_NEIGHBOR\_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 29 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR) 30 <type> SENDBUF(\*), RECVBUF(\*) 31INTEGER SENDCOUNTS(\*), SENDTYPES(\*), RECVCOUNTS(\*), RECVTYPES(\*), COMM, 32 IERROR 33 INTEGER(KIND=MPI\_ADDRESS\_KIND) SDISPLS(\*), RDISPLS(\*) 34 35 MPI\_NEIGHBOR\_ALLTOALLW\_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, 36 RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, 37 IERROR) 38 <type> SENDBUF(\*), RECVBUF(\*) 39 INTEGER SENDCOUNTS(\*), SENDTYPES(\*), RECVCOUNTS(\*), RECVTYPES(\*), COMM, 40INFO, REQUEST, IERROR 41 INTEGER(KIND=MPI\_ADDRESS\_KIND) SDISPLS(\*), RDISPLS(\*) 42MPI\_NEIGHBOR\_ALLTOALL\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 43 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) 44 <tvpe> SENDBUF(\*), RECVBUF(\*) 45 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 46 IERROR 47 48

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1015
MPI_TOPO_TEST(COMM, STATUS, IERROR) INTEGER COMM, STATUS, IERROR	1 2 3
A.5.7 MPI Environmental Management Fortran Bindings	4 5
DOUBLE PRECISION MPI_WTICK()	6
DOUBLE PRECISION MPI_WTIME()	7 8
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR) INTEGER ERRORCLASS, IERROR	9 10 11
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR) INTEGER ERRORCLASS, ERRORCODE, IERROR	11 12 13
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR) INTEGER ERRORCODE, IERROR CHARACTER*(*) STRING	14 15 16 17
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR) INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR INTEGER INFO, IERROR	18 19 20
If the Fortran compiler provides $\texttt{TYPE(C_PTR)}$ , then overloaded by:	21
INTERFACE MPI_ALLOC_MEM	22 23
SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR) IMPORT :: MPI_ADDRESS_KIND	24
INTEGER :: INFO, IERROR	25
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR	26 27
END SUBROUTINE	28
SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	29
IMPORT :: MPI_ADDRESS_KIND	30
INTEGER :: INFO, IERROR	31
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	32
TYPE(C_PTR) :: BASEPTR	33 34
END SUBROUTINE	35
END INTERFACE	36
MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)	37
INTEGER COMM, ERRORCODE, IERROR	38
MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)	39
EXTERNAL COMM_ERRHANDLER_FN	40
INTEGER ERRHANDLER, IERROR	41 42
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)	43
INTEGER COMM, ERRHANDLER, IERROR	44
	45
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR	46
	47
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)	48

```
1
         INTEGER ERRHANDLER, IERROR
\mathbf{2}
     MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
3
         INTEGER ERRORCODE, ERRORCLASS, IERROR
4
\mathbf{5}
     MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
6
         INTEGER ERRORCODE, RESULTLEN, IERROR
7
         CHARACTER*(*) STRING
8
     MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
9
         INTEGER FH, ERRORCODE, IERROR
10
^{11}
     MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
12
         EXTERNAL FILE_ERRHANDLER_FN
13
         INTEGER ERRHANDLER, IERROR
14
     MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
15
         INTEGER FILE, ERRHANDLER, IERROR
16
17
     MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
18
         INTEGER FILE, ERRHANDLER, IERROR
19
     MPI_FREE_MEM(BASE, IERROR)
20
         <type> BASE(*)
21
         INTEGER IERROR
22
23
     MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
^{24}
         CHARACTER*(*) VERSION
25
         INTEGER RESULTLEN, IERROR
26
     MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
27
         CHARACTER*(*) NAME
28
         INTEGER RESULTLEN, IERROR
29
30
     MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
^{31}
         INTEGER VERSION, SUBVERSION, IERROR
32
     MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR)
33
34
         INTEGER SESSION, ERRORCODE, IERROR
35
     MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)
36
         EXTERNAL SESSION_ERRHANDLER_FN
37
         INTEGER ERRHANDLER, IERROR
38
     MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
39
         INTEGER SESSION, ERRHANDLER, IERROR
40
41
     MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
42
         INTEGER SESSION, ERRHANDLER, IERROR
43
44
     MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
45
         INTEGER WIN, ERRORCODE, IERROR
46
     MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
47
         EXTERNAL WIN_ERRHANDLER_FN
48
```

INTEGER ERRHANDLER, IERROR MPI\_WIN\_GET\_ERRHANDLER(WIN, ERRHANDLER, IERROR) INTEGER WIN, ERRHANDLER, IERROR MPI\_WIN\_SET\_ERRHANDLER(WIN, ERRHANDLER, IERROR) INTEGER WIN, ERRHANDLER, IERROR A.5.8 The Info Object Fortran Bindings 10 MPI\_INFO\_CREATE(INFO, IERROR) 11 INTEGER INFO, IERROR 12MPI\_INFO\_CREATE\_ENV(INFO, IERROR) 13 INTEGER INFO, IERROR 1415MPI\_INFO\_DELETE(INFO, KEY, IERROR) 16INTEGER INFO, IERROR 17 CHARACTER\*(\*) KEY 18 MPI\_INFO\_DUP(INFO, NEWINFO, IERROR) 19 INTEGER INFO, NEWINFO, IERROR 2021MPI\_INFO\_FREE(INFO, IERROR) 22 INTEGER INFO, IERROR 23MPI\_INFO\_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)  $^{24}$ INTEGER INFO, VALUELEN, IERROR 25CHARACTER\*(\*) KEY, VALUE 26LOGICAL FLAG 2728MPI\_INFO\_GET\_NKEYS(INFO, NKEYS, IERROR) 29 INTEGER INFO, NKEYS, IERROR 30 MPI\_INFO\_GET\_NTHKEY(INFO, N, KEY, IERROR) 31INTEGER INFO, N, IERROR 32 33 CHARACTER\*(\*) KEY 34 MPI\_INFO\_GET\_STRING(INFO, KEY, BUFLEN, VALUE, FLAG, IERROR) 35 INTEGER INFO, BUFLEN, IERROR 36 CHARACTER\*(\*) KEY, VALUE 37 LOGICAL FLAG 38 39 MPI\_INFO\_GET\_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR) 40 INTEGER INFO, VALUELEN, IERROR 41 CHARACTER\*(\*) KEY 42LOGICAL FLAG 43 MPI\_INFO\_SET(INFO, KEY, VALUE, IERROR) 44INTEGER INFO, IERROR 45CHARACTER\*(\*) KEY, VALUE 4647

1 2

3

4 5

6

9

```
1
     A.5.9 Process Creation and Management Fortran Bindings
\mathbf{2}
     MPI_ABORT(COMM, ERRORCODE, IERROR)
3
         INTEGER COMM, ERRORCODE, IERROR
4
\mathbf{5}
     MPI_CLOSE_PORT(PORT_NAME, IERROR)
6
         CHARACTER*(*) PORT_NAME
7
         INTEGER IERROR
8
     MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
9
         CHARACTER*(*) PORT_NAME
10
         INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
11
12
     MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
13
         CHARACTER*(*) PORT_NAME
14
         INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
15
     MPI_COMM_DISCONNECT(COMM, IERROR)
16
         INTEGER COMM, IERROR
17
18
     MPI_COMM_GET_PARENT(PARENT, IERROR)
19
         INTEGER PARENT, IERROR
20
     MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
21
         INTEGER FD, INTERCOMM, IERROR
22
23
     MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
^{24}
                   ARRAY_OF_ERRCODES, IERROR)
25
         CHARACTER*(*) COMMAND, ARGV(*)
26
         INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
27
                    IERROR
28
     MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
29
                   ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
30
                   ARRAY_OF_ERRCODES, IERROR)
31
         INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,
32
                    INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
33
         CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
34
35
     MPI_FINALIZE(IERROR)
36
         INTEGER IERROR
37
     MPI_FINALIZED(FLAG, IERROR)
38
         LOGICAL FLAG
39
         INTEGER IERROR
40
41
     MPI_INIT(IERROR)
42
         INTEGER IERROR
43
     MPI_INITIALIZED(FLAG, IERROR)
44
45
         LOGICAL FLAG
46
         INTEGER IERROR
47
     MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
48
```

INTEGER REQUIRED, PROVIDED, IERROR 1 2 MPI\_IS\_THREAD\_MAIN(FLAG, IERROR) LOGICAL FLAG INTEGER IERROR MPI\_LOOKUP\_NAME(SERVICE\_NAME, INFO, PORT\_NAME, IERROR) CHARACTER\*(\*) SERVICE\_NAME, PORT\_NAME INTEGER INFO, IERROR MPI\_OPEN\_PORT(INFO, PORT\_NAME, IERROR) 10 INTEGER INFO, IERROR 11 CHARACTER\*(\*) PORT\_NAME 1213 MPI\_PUBLISH\_NAME(SERVICE\_NAME, INFO, PORT\_NAME, IERROR) 14CHARACTER\*(\*) SERVICE\_NAME, PORT\_NAME 15INTEGER INFO, IERROR 16MPI QUERY THREAD (PROVIDED, IERROR) 17 INTEGER PROVIDED, IERROR 18 19 MPI\_SESSION\_FINALIZE(SESSION, IERROR) 20INTEGER SESSION, IERROR 21MPI\_SESSION\_GET\_INFO(SESSION, INFO\_USED, IERROR) 22 INTEGER SESSION, INFO\_USED, IERROR 23 $^{24}$ MPI\_SESSION\_GET\_NTH\_PSET(SESSION, INFO, N, PSET\_LEN, PSET\_NAME, IERROR) 25INTEGER SESSION, INFO, N, PSET\_LEN, IERROR 26CHARACTER\*(\*) PSET\_NAME 27MPI\_SESSION\_GET\_NUM\_PSETS(SESSION, INFO, NPSET\_NAMES, IERROR) 28INTEGER SESSION, INFO, NPSET\_NAMES, IERROR 29 30 MPI\_SESSION\_GET\_PSET\_INFO(SESSION, PSET\_NAME, INFO, IERROR) 31INTEGER SESSION, INFO, IERROR 32 CHARACTER\*(\*) PSET\_NAME 33 MPI\_SESSION\_INIT(INFO, ERRHANDLER, SESSION, IERROR) 34 INTEGER INFO, ERRHANDLER, SESSION, IERROR 3536 MPI\_UNPUBLISH\_NAME(SERVICE\_NAME, INFO, PORT\_NAME, IERROR) 37 CHARACTER\*(\*) SERVICE\_NAME, PORT\_NAME 38 INTEGER INFO, IERROR 39 40 41 A.5.10 One-Sided Communications Fortran Bindings 42MPI\_ACCUMULATE(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, 43 TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, IERROR) 44 <type> ORIGIN\_ADDR(\*) 45INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 46TARGET\_DATATYPE, OP, WIN, IERROR 47INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 48

1 2 3 4 5	<pre>MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK, TARGET_DISP, WIN, IERROR) <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*) INTEGER DATATYPE, TARGET_RANK, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP</type></pre>
6 7 8 9 10 11	MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK, TARGET_DISP, OP, WIN, IERROR) <type> ORIGIN_ADDR(*), RESULT_ADDR(*) INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP</type>
12 13 14 15 16 17	<pre>MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>
18 19 20 21 22 23 24 25	<pre>MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,</pre>
26 27 28 29 30 31	<pre>MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>
32 33 34 35 36 37 38 39	<pre>MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>
40 41 42 43 44 45 46 47 48	<pre>MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>

1 RESULT\_ADDR, RESULT\_COUNT, RESULT\_DATATYPE, TARGET\_RANK, TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, REQUEST,  $\mathbf{2}$ IERROR) <type> ORIGIN\_ADDR(\*), RESULT\_ADDR(\*) 4 INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, RESULT\_COUNT, RESULT\_DATATYPE, 5TARGET\_RANK, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, REQUEST, 6 7 IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 9 MPI\_RPUT(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, 10 TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, REQUEST, 11 IERROR) 12<type> ORIGIN\_ADDR(\*) 13 INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 14TARGET\_DATATYPE, WIN, REQUEST, IERROR 15INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 1617MPI\_WIN\_ALLOCATE(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 18 INTEGER(KIND=MPI\_ADDRESS\_KIND) SIZE, BASEPTR INTEGER DISP\_UNIT, INFO, COMM, WIN, IERROR 19 20If the Fortran compiler provides TYPE(C\_PTR), then overloaded by: 21INTERFACE MPI\_WIN\_ALLOCATE 22 SUBROUTINE MPI\_WIN\_ALLOCATE(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, & 23WIN, IERROR) 24IMPORT :: MPI\_ADDRESS\_KIND 25INTEGER :: DISP\_UNIT, INFO, COMM, WIN, IERROR 26INTEGER(KIND=MPI\_ADDRESS\_KIND) :: SIZE, BASEPTR 27END 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 31IMPORT :: MPI\_ADDRESS\_KIND 32 INTEGER :: DISP\_UNIT, INFO, COMM, WIN, IERROR 33 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: SIZE 34 TYPE(C\_PTR) :: BASEPTR 35 END SUBROUTINE 36 END INTERFACE 37 38 MPI\_WIN\_ALLOCATE\_SHARED(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 39 INTEGER(KIND=MPI\_ADDRESS\_KIND) SIZE, BASEPTR 40 INTEGER DISP\_UNIT, INFO, COMM, WIN, IERROR 41 If the Fortran compiler provides TYPE(C\_PTR), then overloaded by: 42INTERFACE MPI\_WIN\_ALLOCATE\_SHARED 43 SUBROUTINE MPI\_WIN\_ALLOCATE\_SHARED(SIZE, DISP\_UNIT, INFO, COMM, & 44BASEPTR, WIN, IERROR) 45IMPORT :: MPI\_ADDRESS\_KIND 46INTEGER :: DISP\_UNIT, INFO, COMM, WIN, IERROR 47INTEGER(KIND=MPI\_ADDRESS\_KIND) :: SIZE, BASEPTR 48

```
1
         END SUBROUTINE
2
         SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
3
               BASEPTR, WIN, IERROR)
4
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
5
           IMPORT :: MPI_ADDRESS_KIND
6
           INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
7
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
8
           TYPE(C_PTR) :: BASEPTR
9
         END SUBROUTINE
10
       END INTERFACE
11
     MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
12
         INTEGER WIN, IERROR
13
         <type> BASE(*)
14
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
15
16
     MPI_WIN_COMPLETE(WIN, IERROR)
17
         INTEGER WIN, IERROR
18
     MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
19
         <type> BASE(*)
20
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
21
         INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
22
23
     MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
^{24}
         INTEGER INFO, COMM, WIN, IERROR
25
     MPI_WIN_DETACH(WIN, BASE, IERROR)
26
         INTEGER WIN, IERROR
27
         <type> BASE(*)
28
29
     MPI_WIN_FENCE(ASSERT, WIN, IERROR)
30
         INTEGER ASSERT, WIN, IERROR
^{31}
     MPI_WIN_FLUSH(RANK, WIN, IERROR)
32
         INTEGER RANK, WIN, IERROR
33
34
     MPI_WIN_FLUSH_ALL(WIN, IERROR)
35
         INTEGER WIN, IERROR
36
37
     MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
         INTEGER RANK, WIN, IERROR
38
39
     MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
40
         INTEGER WIN, IERROR
41
42
     MPI_WIN_FREE(WIN, IERROR)
         INTEGER WIN, IERROR
43
44
     MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
45
         INTEGER WIN, GROUP, IERROR
46
47
     MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
48
         INTEGER WIN, INFO_USED, IERROR
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1023
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR) INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR	1 2
MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR) INTEGER ASSERT, WIN, IERROR	3 4 5
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR	6 7 8
MPI_WIN_SET_INFO(WIN, INFO, IERROR) INTEGER WIN, INFO, IERROR	9 10
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR) INTEGER WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	11 12 13
If the Fortran compiler provides TYPE(C_PTR), then overloaded by: INTERFACE MPI_WIN_SHARED_QUERY	14 15 16
SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, & BASEPTR, IERROR)	17 18
IMPORT :: MPI_ADDRESS_KIND INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR	19 20
END SUBROUTINE SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &	21 22 23
BASEPTR, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND	24 25
INTEGER :: WIN, RANK, DISP_UNIT, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	26 27 28
TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE	29 30 31
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR	32 33
MPI_WIN_SYNC(WIN, IERROR) INTEGER WIN, IERROR	34 35 36
MPI_WIN_TEST(WIN, FLAG, IERROR) INTEGER WIN, IERROR LOGICAL FLAG	37 38 39
MPI_WIN_UNLOCK(RANK, WIN, IERROR) INTEGER RANK, WIN, IERROR	40 41 42
MPI_WIN_UNLOCK_ALL(WIN, IERROR) INTEGER WIN, IERROR	43 44
MPI_WIN_WAIT(WIN, IERROR) INTEGER WIN, IERROR	45 46 47
INTEGER WIN, IERROR	47 48

```
1
     A.5.11 External Interfaces Fortran Bindings
\mathbf{2}
     MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
3
         INTEGER REQUEST, IERROR
4
\mathbf{5}
     MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
6
                   IERROR)
7
         EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
8
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
9
         INTEGER REQUEST, IERROR
10
     MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
11
         INTEGER STATUS(MPI_STATUS_SIZE), IERROR
12
         LOGICAL FLAG
13
14
     MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
15
         INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
16
     MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
17
         INTEGER STATUS (MPI_STATUS_SIZE), DATATYPE, IERROR
18
         INTEGER(KIND=MPI_COUNT_KIND) COUNT
19
20
21
     A.5.12 I/O Fortran Bindings
22
     MPI_CONVERSION_FN_NULL(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION,
23
                   EXTRA_STATE, IERROR)
24
         <TYPE> USERBUF(*), FILEBUF(*)
25
         INTEGER DATATYPE, COUNT, IERROR
26
         INTEGER(KIND=MPI_OFFSET_KIND) POSITION
27
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
28
29
     MPI_FILE_CLOSE(FH, IERROR)
30
         INTEGER FH, IERROR
^{31}
     MPI_FILE_DELETE(FILENAME, INFO, IERROR)
32
33
         CHARACTER*(*) FILENAME
34
         INTEGER INFO, IERROR
35
     MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
36
         INTEGER FH, AMODE, IERROR
37
38
     MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
39
         INTEGER FH, IERROR
40
         LOGICAL FLAG
41
     MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
42
         INTEGER FH, IERROR
43
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
44
45
     MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
46
         INTEGER FH, GROUP, IERROR
47
     MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
48
```

INTEGER FH, INFO_USED, IERROR	1
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)	2 3
INTEGER FH, IERROR	4
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	5
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)	6
INTEGER FH, IERROR	7
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	8
	9
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)	10
INTEGER FH, IERROR	11
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	12
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)	13
INTEGER FH, DATATYPE, IERROR	14
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT	15 16
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)	10
INTEGER FH, ETYPE, FILETYPE, IERROR	18
INTEGER(KIND=MPI_OFFSET_KIND) DISP	19
CHARACTER*(*) DATAREP	20
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	21
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	22
<type> BUF(*)</type>	23
	24
MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	25
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	26
<type> BUF(*)</type>	27
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	28 29
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	30
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	31
<type> BUF(*)</type>	32
MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	33
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	34
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	35
<type> BUF(*)</type>	36
MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	37
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	38
<type> BUF(*)</type>	39
MOT ETLE TUDITE (EU DUE COUNT DATATVOE DECUERT TEDDOD)	40
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	41 42
<pre><pre><pre><pre><pre><pre><pre><pre></pre></pre></pre></pre></pre></pre></pre></pre>	43
	43
MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	45
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	46
<type> BUF(*)</type>	47
MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	48

```
1
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
\mathbf{2}
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
3
         <type> BUF(*)
4
     MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
5
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
6
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
7
         <type> BUF(*)
8
9
     MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
10
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
11
         <type> BUF(*)
12
     MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
13
         INTEGER COMM, AMODE, INFO, FH, IERROR
14
         CHARACTER*(*) FILENAME
15
16
     MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
17
         INTEGER FH, IERROR
18
         INTEGER(KIND=MPI_OFFSET_KIND) SIZE
19
    MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
20
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
21
         <type> BUF(*)
22
23
     MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
24
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
25
         <type> BUF(*)
26
     MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
27
         INTEGER FH, COUNT, DATATYPE, IERROR
28
         <type> BUF(*)
29
30
     MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
31
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
32
         <type> BUF(*)
33
     MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
36
         <type> BUF(*)
37
38
     MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
39
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
40
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
41
         <type> BUF(*)
42
     MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
43
         INTEGER FH, COUNT, DATATYPE, IERROR
44
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45
         <type> BUF(*)
46
47
     MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
48
```

INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type>	1 2
	3
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	4
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	5
<type> BUF(*)</type>	6
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	7
INTEGER FH, COUNT, DATATYPE, IERROR	8
<type> BUF(*)</type>	9
lojpor bor (l)	10
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)	11
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	12
<type> BUF(*)</type>	13
	14
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	15
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	16
<type> BUF(*)</type>	17
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)	18
INTEGER FH, WHENCE, IERROR	19
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	
	20
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)	21
INTEGER FH, WHENCE, IERROR	22
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	23
MDT ETTE GET ATOMICITY/EU ELAC TERROR)	24
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)	25
INTEGER FH, IERROR	26
LOGICAL FLAG	26 27
LOGICAL FLAG	
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR)	27
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR	27 28
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR)	27 28 29
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR	27 28 29 30
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR)	27 28 29 30 31
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE	27 28 29 30 31 32
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)	27 28 29 30 31 32 33
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR	27 28 29 30 31 32 33 34
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER (KIND=MPI_OFFSET_KIND) DISP	27 28 29 30 31 32 33 34 35
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR	27 28 29 30 31 32 33 34 35 36
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER (KIND=MPI_OFFSET_KIND) DISP	27 28 29 30 31 32 33 34 35 36 37
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER(KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP	27 28 29 30 31 32 33 34 35 36 37 38
<pre>LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER(KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR</pre>	27 28 29 30 31 32 33 34 35 36 37 38 39
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER (KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	27 28 29 30 31 32 33 34 35 36 37 38 39 40
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER (KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS_SIZE), IERROR	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER (KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
<pre>LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER(KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS_SIZE), IERROR </pre>	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
<pre>LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER (KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER (KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS_SIZE), IERROR </pre>	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
<pre>LOGICAL FLAG MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR MPI_FILE_SET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR) INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER(KIND=MPI_OFFSET_KIND) DISP CHARACTER*(*) DATAREP MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS_SIZE), IERROR </pre>	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43

1 2 3	MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR <type> BUF(*)</type>
4 5 6 7	MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type>
8 9 10 11	<pre>MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>
12 13 14 15 16	<pre>MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>
17 18 19 20	<pre>MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>
21 22 23 24	<pre>MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>
25 26 27 28	<pre>MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>
29 30 31	<pre>MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR <type> BUF(*)</type></pre>
32 33 34 35	<pre>MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>
36 37 38	<pre>MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>
<ol> <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>	<pre>MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR) CHARACTER*(*) DATAREP EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE INTEGER IERROR</pre>

A.5.13 Language Bindings Fortran Bindings	<sup>1</sup> #-error!
The following procedure is not available with mpif.h: MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR) TYPE(MPI_Status) :: F08_STATUS INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR	2
The following procedure is not available with mpif.h: MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR) INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR TYPE(MPI_Status) :: F08_STATUS	7 8 9 10
MPI_F_SYNC_REG(BUF) <type> BUF(*)</type>	11 12 13
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR	14 15
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR) INTEGER R, NEWTYPE, IERROR	16 17 18
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR	19 20
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR) INTEGER TYPECLASS, SIZE, DATATYPE, IERROR	21 22 23
A.5.14 Tools / Profiling Interface Fortran Bindings	24 25 26
MPI_PCONTROL(LEVEL) INTEGER LEVEL	27 28 29
A.5.15 Deprecated Fortran Bindings	30 31
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR) INTEGER COMM, KEYVAL, IERROR	32 33
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR LOGICAL FLAG	34 35 36 37
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR	38 39
MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR) INTECEP OLDCOMM KEYVAL EXTRA STATE ATTRIBUTE VAL IN	40 41 42
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR	43 44 45
LOGICAL FLAG	

```
1
         CHARACTER*(*) KEY, VALUE
\mathbf{2}
         LOGICAL FLAG
3
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
4
         INTEGER INFO, VALUELEN, IERROR
5
         CHARACTER*(*) KEY
6
         LOGICAL FLAG
7
8
     MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
9
         EXTERNAL COPY_FN, DELETE_FN
10
         INTEGER KEYVAL, EXTRA_STATE, IERROR
11
     MPI_KEYVAL_FREE(KEYVAL, IERROR)
12
         INTEGER KEYVAL, IERROR
13
14
     MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
15
                    ATTRIBUTE_VAL_OUT, FLAG, IERR)
16
         INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
17
                     ATTRIBUTE_VAL_OUT, IERR
18
         LOGICAL FLAG
19
     MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
20
         INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR
21
22
     MPI_SIZEOF(X, SIZE, IERROR)
23
         <type> X
^{24}
         INTEGER SIZE, IERROR
25
26
27
28
29
30
^{31}
32
33
34
35
36
37
38
39
40
^{41}
42
43
44
45
46
47
48
```

	#-359 Ar	9 # all changes #-nnn by issue nnn #-PRnnn by pull request Non problematic fomatting-changes and typo-correct Embiggening: only new _c versions are marked, all New or changed content Problematic fomatting-changes are listed #-error Errors in RC 40 - also marked with #-error B.m.n: i refers to Change-Log section B.m.n, Item	other is not listed <sup>1</sup> 2 3 4	
	$\mathbf{C}$	hange-Log	8 9 10 11	)
Move item B.2.1: 1. to here	sion featu the u forma the so this v corre B.1 B.1.1 1.		tions and new <sup>15</sup> aries or change <sup>16</sup> modifications, <sup>17</sup> therwise noted, <sup>18</sup> cunctionality in <sup>16</sup> roduced in the <sup>20</sup> <sup>21</sup> <sup>22</sup> <sup>23</sup> <sup>24</sup> Items should be sorted b their first related page <sup>27</sup> utines and the <sup>28</sup> f.h. <sup>29</sup> PI-3.1 Sections <sup>31</sup> <sup>32</sup>	Done #-error #-359+389 #-331 #-error! in whole B.1 Done in PR427 #-error Done in PR427
	B.1.2	<ul> <li>MPI_F_ERROR.</li> <li>Section 19.3.5 on page 833, and MPI-3.1 Section 17.2.5 on page 658.</li> <li>Added missing const to IN parameters for MPI_STATUS_F2F08 and MPI_STATUS_F082F.</li> <li>Changes in MPI-4.0</li> <li>Section 2.4, 3.4, 3.7.2, 3.7.3, 3.8.1, 3.8.2, 6.13, 14.4.5, and Annex A.2 o 60, 67, 80, 83, 271, 675, and 869.</li> </ul>	34 35 36 37 38 38 40	
#-error #-335 #-358 + 35 2.2. p10 18.1 p.777 19.1.5 p78 B.1.2 p.103	9 3. 9 1ff	The semantic terms were updated. Remark: 16.3 and 16.4 are now one section ==> here all okay PR434 MPI_SIZEOF was deprecated Chapter 4 on page 99. A new chapter on partitioned communication was added. Sections 2.2, 18.1., and 19.1.5 on pages 9, 777, and 786. The limit for the maximum length of MPI identifiers was removed. This change. Section 19.1.5 on page 786.	47	Done in PR461 #-update8
#-update	2	An execption for the specific Fortran names in the case of TS 29113 interfaction of the specific Fortran names in the case of TS 29113 interfaction of the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of TS 29113 interfactors and the specific Fortran names in the case of the specific Fortran names in the specific Fortran names in the case of the specific Fortran names in the specific		

Done in

Done

PR434

in

PR434

New MPI procedures are MPI\_SESSION\_{INIT | FINALIZE}, MPI\_SESSION\_GET\_{...}, MPI\_SESSION\_{...}ERRHANDLER, MPI\_GROUP\_FROM\_SESSION \_PSET, MPI\_COMM\_CREATE\_FROM\_GROUP, MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS, and new conversion functions are MPI\_SESSION\_{C2F | F2C}. New declarations are MPI\_Session in C and TYPE(MPI\_Session) together with the related overloaded operators .EQ., .NE., == and /= in the Fortran mpi\_f08 and mpi modules, and the callback function prototype MPI\_Session\_errhandler\_function. New constants are MPI\_SESSION\_NULL, MPI\_ERR\_SESSION, MPI\_MAX\_PSET\_NAME\_LEN, MPI\_MAX\_STRINGTAG\_LEN, MPI\_T\_BIND\_MPI\_SESSION and the predefined info key "mpi\_size".

Done in	1 2 3	4		Section 7.4.2 on page 321. MPI_COMM_TYPE_HW_UNGUIDED was added as a new possible value for the split_type parameter of the MPI_COMM_SPLIT_TYPE function.
PR434 #-update1	4	F		Section 15.3.8. on page 745 with the MPI_T_{SOURCE   EVENT}{} and MPI_T_CATEGORY_{GET   GET_NUM}_EVENTS routines
#-update1 #-update8	$\frac{5}{6}$	و		A callback-driven event interface was added to the MPI tool information interface.
Done in PR461	7 8 9 10			Section 7.4.2 on page 321. MPI_COMM_TYPE_HW_GUIDED was added as a new possible value for the split_type parameter of the MPI_COMM_SPLIT_TYPE function, as well as a new info key "mpi_hw_resource_type". A specific value associated with this new info key is also
#-update9	in			defined: "mpi_shared_memory". There seems to be absolutely
#-error!!!	<sup>11</sup> PR4	461	7	Ino change for Sessions!!!!           Chapter 11, Sections 28, 3.2.3, 7.2.4, 7.3.2, 7.4.2, 7.6.2, 8.4, 8.5.1, 8.5.3, 8.5.4, 9.1.1,
#- <u>103</u>	14			9.1.2, 9.3, 9.3.4, 9.5, 11.6, 14.2.1, 14.2.7, 14.7, 15.3.4, 19.3.4, 19.3.6, and Annex A on
Done	15 16			pages 479, <b>24</b> , 33, 309, 312, 321, 352, 385, 386, 388, 390, 443, 445, 450, 458, 462, 508,
#-update8	10			633, 639, 706, 722, 829, 834, and 845 The Sessions Model was added to the standard.
Done	18	s	2	Throughout the entire document.
in PR461	19 20	C		New large count functions and callbacks were introduced to accomodate large buffers
	20 21			and/or datatypes.
	22			Clarifications were added to the behavior of INOUT/OUT parameters that cannot
	23 24			represent the value to be returned for the MPI_BUFFER_DETACH and MPI_FILE_GET_TYPE_EXTENT functions.
	25			A new error class MPI_ERR_VALUE_TOO_LARGE was introduced.
	26	C		New large count functions MPI_{}_c in C and
	27 28	e U		Section 11.2.1 on page 480. through function overloading in the A new function MPI_INFO_CREATE_ENV was added. Fortran mpi_f08 module, (with the exception of the
	29			explicit Fortran procedures
	30 21	B.2	2	Changes from Version 3.1 to Version 3.2 MPI_Op_create_c and MPI_Register_datarep_c) and the new large count callbacks
Done	31 32	B.2	1	Eixes to Errata in Previous Versions of MPI MPI_User_function_c and MPI_Datarep conversion function c were
	33			
	34 25	1		Sections 8.6.1, 8.6.2 and 8.9 on pages 410, 415 and 437, and MPI-3.1 Sections 7.6.1, 7.6.2 and 7.8 on pages 315, 318 and 329.
	e this i re B.1			MPI_NEIGHBOR_ALLTOALL{ V W} and MPI_NEIGHBOR_ALLGATHER{ V} for Car-
Delo	37			tesian virtual grids were clarified. An advice to implementors was added to illustrate
	38 39			a correct implementation for the case of $periods[d] == 1$ or .TRUE. and $dims[d] == 1$ or 2 in a direction d.
	40			
Done	41	B.2	.2	Changes in MIPI-3.2 Sort the combined list by their first related page
	42 43	1		Section 3.7 on page 58.
	44			The introduction of MPI nonblocking communication was changed to describe cor- rectness and performance reasons for the use of nonblocking communication.
# orror!	45	-		
#-error!	46 47	2		Section 3.8.4 on page 88. Sections 3.8.4 and 16.3 on pages 88 and 772. Cancelling a send request by calling MPI_CANCEL has been deprecated and may be
	48			removed in a future version of the MPI specification.

<i>B.2.</i>	CHANGES FROM VERSION 3.1 TO VERSION 3.2 1033	Done in	Ð
	MPI_{ALLGATHER }_INIT MPI_NEIGHBOR_{ALLGATHER }_INIT	PR4	61
3.	Persistent collective communication and persistent neighborhood communication were added to the standard.	2 3 Dor	odate8
4.	Section 7.4.2 on page 321, and MPI 3.1 Section 6.4.2 on page 237. done only for errata! The functions MPI_COMM_DUP and MPI_COMM_IDUP were updated to no longer propagate info hints. This change may affect backward compatibility.	4 PR4 5 #-er	
5.	MPI 3.1 Sections 6.4.4, 11.2.7, and 14.2.8 on pages 359, 557, and 641, 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.	9 10 11 12 13	
6.	Section 7.4.4 on page 339. 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	14 15 16 17	
7.	The MPI_COMM_IDUP_WITH_INFO function was added.	18 19 20	
	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. Section 15.3.10, Table 15.7 and Section 16.3 on pages 766, 768, and 772. Section 15.3.10 and Table 15.7 on pages 766 and 768. MPI_T_ERR_INVALID_ITEM is deprecated. MPI routines should return	21 22 23 24 25 26 27	
10.	Section 8.5. Section 8.5.2 on page 387. MPI_DIMS_CREATE is now guaranteed to return MPI_SUCCESS if the number of di-	28 29 30 31	
11.	Sections 2.8, 9.3, 9.5, and 11.2.1 on pages 24, 450, 462, and 480. 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 37	
12.	Sections 9.2, 12.2.2, and 12.2.3 on pages 447, 546, and 548. Introduced alignment requirements for memory allocated through MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, and MPI_WIN_ALLOCATE_SHARED and added a new info key	38 39 40 41	
13.	Section 9.4 on page 463. The error class MPI ERR PROC ABORTED has been added.	42 43 44	
14.	The mpi_f08 binding incorrectly had the dummy parameter flag in the MPI F08 binding for MPI_STATUS_SET_CANCELLED marked as INTENT(OUT). It has been	45 46 47 48	

#-error!	Missing ch	ange-log	for 1-sided passiv synchronization is now also allowed	for memory	<pre>v allocated with MPI_Win_allocate_shared</pre>
12.5.3 p595:9-13	109/		Section~\ref{sec:1sided-lock} on page~\pageref{sec:1sided-lock} on page~\pageref{sec:1sided-lock} on page~\pageref{sec:1sided}		
#-update2			\mpifunc{MPI\_WIN\_ALLOCATE\_SHARED}.		
in PR434	<sup>1</sup> 15.		s $9.3$ and $9.4$ on pages $450$ and $461$ . In definition of errors to say that MPI should c	ontinue w	henever possible and
	3		he user to recover from errors.	ononiue w	henever possible and
	4	anow u			
#-error!	<sub>5</sub> 16.	Section	10. on page 471.		
Done	6	Added	a new function MPI_INFO_GET_STRING that	takes a bu	iffer length argument
	7	for retu	urning info value strings. This function returns	the requi	ired buffer length for
	8	the req	uested string and guarantees null termination	for C strip	ngs where buffer size
	9	is great	ter than 0.	Remark: 1	6.3 and 16.4
	10	~ .	3	are now o	ne section
#-error	17.		10 on page 471 and Section 16. on page 772.		
Done	12	MPI_IN	IFO_GET and MPI_INFO_GET_VALUELEN wer	e depreca	ted.
	13 18	Section	9.4 on page 461.		
	14		text to clarify what is implied about the status	of MPI ar	nd user visible buffers
	15		API functions return MPI_SUCCESS or other error		id user visible bullets
J	16	whom w		Si coucs.	
#-error!!!	17 19.	Section	11.2.1 on page $480$ Section $11.10.4$ on page $53$	36.	
#-update1	18	Clarifie	d the semantic of failure and error reporting	before (ar	nd during) MPI_INIT
Done	19	and aft	er MPI_FINALIZE.		
	20 20	<b>a</b>			
#-error!!!	20. 21		11.8.4 on page 521. Section 11.8.4 on page 521		1 1
#-update1	22		the "mpi_initial_errhandler" reserved info key with		
Done	23	-	rors_abort", "mpi_errors_are_fatal", and "mpi_error		0
	24	MPI_C	OMM_SPAWN, MPI_COMM_SPAWN_MULTIPL	E, and m	piexec
#-error!!!	<sup>25</sup> 21.	Section	3.7 and 3.9 60 and 90 3.9 and 3.7 on pages 90 and 60.		
#-update1	26		on of MPI_ISENDRECV and MPI_ISENDRECV_I	REPI ACE	
Dana	27				Missing Change-log entries or lines will be
# update2	<sup>28</sup> D 2	CL			added if new constants, types, callbacks
#-update2	<sup>20</sup> B.3	Char	nges from Version 3.0 to Version 3.1		and functions are added to MPI-4.0 and not
#-TODO	<sup>30</sup> B.3.		to Errata in Previous Versions of MPI		listed above.
	31 D.J.	I TIXES	to Effata III Trevious versions of IVII I		
	32 1.	. Chapte	rs 3–19, Annex A.4 on page 908, and Example	6.21 on p	age $234$ , and MPI-3.0
	33	Chapte	rs 3–17, Annex A.3 on page 707, and Example	5.21 on p	age 187.
	34	Within	the mpi_f08 Fortran support method, BIND(C)	) was rem	oved from all
	35	SUBROU	TINE, FUNCTION, and ABSTRACT INTERFACE def	initions.	
	36 0	C	2.9.5 m man 26 and MDI 2.0 Station 2.9.5 m		
	37 Z.		3.2.5 on page 36, and MPI-3.0 Section 3.2.5 on	* 0	of the Fortner derived
	38		ree public fields MPI_SOURCE, MPI_TAG, and MP		of the Fortran derived
	39	type i	YPE(MPI_Status) must be of type INTEGER.		
	<sup>40</sup> 3.	Section	3.8.2 on page 83, and MPI-3.0 Section 3.8.2 on	n page 67.	
	41		g arguments of the Fortran interfaces of MPI_IN		were originally incor-
	42		lefined as INTEGER (instead as LOGICAL).		- ·
	43	Ū.			
	44 4.		7.4.2 on page 321, and MPI-3.0 Section 6.4.2 o		
	45		<pre>mpi_f08 binding of MPI_COMM_IDUP, the o</pre>	utput arg	ument
	46	newcom	nm is declared as ASYNCHRONOUS.		
	47				
	48				

5.	Section 7.4.4 on page 339, 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.	1 2 3
6.	Section 8.6 on page 410, 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).	4 5 6 7 8 9
7.	Section 9.1.1 on page 443, 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.	10 11 12 13
8.	Sections 9.2 (MPI_ALLOC_MEM and MPI_ALLOC_MEM_CPTR), 12.2.2 (MPI_WIN_ALLOCATE and MPI_WIN_ALLOCATE_CPTR), 12.2.3 (MPI_WIN_ALLOCATE_SHARED and MPI_WIN_ALLOCATE_SHARED_CPTR), 12.2.3 (MPI_WIN_SHARED_QUERY and MPI_WIN_SHARED_QUERY_CPTR), 15.2.1 and 15.2.7 (Profiling interface), and corresponding sections in MPI-3.0. The linker name concept was substituted by defining specific procedure names.	14 15 16 17 18 19
9.	Section 12.2.1 on page 543, 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.	20 21 22 23 24
10.	Section 12.3.4 on page 566, 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.	25 26 27 28
11.	Section 12.3.4 on page 566, 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.	29 30 31
12.	Section 15.3 on page 719, 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.	32 33 34 35
13.	Section 15.3.3 on page 721, and MPI-3.0 Section 14.3.3 on page 563. New paragraph clarifying variable name equivalence in the tools information interface.	36 37 38
14.	Sections 15.3.6, 15.3.7, and 15.3.9 on pages 726, 733, and 761, 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.	39 40 41 42 43 44
15.	Section 15.3.7 on page 733, and MPI-3.0 Section 14.3.7 on page 573. Clarify return code of MPI_T_PVAR_{START,STOP,RESET} routines.	45 46 47 48

1 2	16. Section 15.3.7 on page 733, and MPI-3.0 Section 14.3.7 Clarify the return code when bad handle is passed to an	1 O /
3 4 5 6	17. Section 19.1.4 on page 785, and MPI-3.0 Section 17.1.4 on The advice to implementors at the end of the section we the following section.	
7 8 9	18. Section 19.1.5 on page 786, and MPI-3.0 Section 17.1.5 on the section was fully rewritten. The linker name concept specific procedure names.	
10 11 12	19. Section 19.1.6 on page 791, and MPI-3.0 Section 17.1.6 on The requirements on BIND(C) procedure interfaces were	· · · · · · · · · · · · · · · · · · ·
13 14 15 16 17	20. Annexes A.3, A.4, and A.5 on pages 870, 908, and 995, MPI-3.0 Annexes A.2, A.3, and A.4 on pages 685, 707, a The predefined callback MPI_CONVERSION_FN_NULL nexes.	and 756.
18 19 20 21	21. Annex A.4.5 on page 948, and MPI-3.0 Annex A.3.4 on In the mpi_f08 binding of MPI_{COMM TYPE WIN}_{DUP NULL_COPY NULL_D information was removed.	
22 23	3.3.2 Changes in MPI-3.1	
24 25 26 27	<ol> <li>Sections 2.6.4 and 5.1.5 on pages 24 and 137. The use of the intrinsic operators "+" and "-" for absorby MPI_AINT_ADD and MPI_AINT_DIFF. In C, they can</li> </ol>	
28 29 30 31 32 33	2. Sections 9.1.1, 11.2.1, and 11.6 on pages 443, 480, and 5 The routines MPI_INITIALIZED, MPI_FINALIZED, MPI MPI_IS_THREAD_MAIN, MPI_GET_VERSION, and M are callable from threads without restriction (in the sense irrespective of the actual level of thread support provide plementation supports threads.	_QUERY_THREAD, PI_GET_LIBRARY_VERSION of MPI_THREAD_MULTIPLE),
34 35 36	<ol> <li>Section 12.2.1 on page 543. The "same_disp_unit" info key was added for use in RMA</li> </ol>	window creation routines.
37 38 39	4. Sections 14.4.2 and 14.4.3 on pages 650 and 657. Added MPI_FILE_IREAD_AT_ALL, MPI_FILE_IWRITE_ MPI_FILE_IREAD_ALL, and MPI_FILE_IWRITE_ALL	_AT_ALL,
40 41 42 43	5. Sections 15.3.6, 15.3.7, and 15.3.9 on pages 726, 733, an Clarified that NULL parameters can be provided in MPI_T_{CVAR PVAR CATEGORY}_GET_INFO routines	
44 45 46 47 48	6. Sections 15.3.6, 15.3.7, 15.3.9, and 15.3.10 on pages 726 New routines MPI_T_CVAR_GET_INDEX, MPI_T_PVA MPI_T_CATEGORY_GET_INDEX, were added to support ables and categories. The error codes MPI_T_ERR_INV MPI_T_ERR_INVALID_NAME were added to indicate invalue	R_GET_INDEX, ort retrieving indices of vari- VALID and

<i>B.4</i> .	CHANGES FROM VERSION 2.2 TO VERSION 3.0	1037
B.4	Changes from Version 2.2 to Version 3.0	1
B.4.1	Fixes to Errata in Previous Versions of MPI	2 3
1.	Sections 2.6.2 and 2.6.3 on pages 22 and 23, and MPI-2.2 Section 2.6.2 on page lines 41–42, Section 2.6.3 on page 18, lines 15–16, and Section 2.6.4 on page lines 40–41. This is an MPI-2 erratum: The scope for the reserved prefix MPI_ and the C namespace MPI is now any name as originally intended in MPI-1.	18, <sup>5</sup> 6
	Sections 3.2.2, 6.9.2, 14.5.2 Table 14.2, and Annex A.1.1 on pages 31, 222, 689, 845, and MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, Annex A.1.1 on pages 27, 164, 433, 472 and 513 This is an MPI-2.2 erratum: New named predefined datatypes MPI_CXX_BOOL, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran correspondin the C++ types bool, std::complex <float>, std::complex<double>, and std::complex<long double="">. These datatypes also correspond to the deprect C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non-stand C++ types Complex&lt;&gt; were substituted by the standard types std::complex&lt;&gt;.</long></double></float>	and 10 and 11 12 13 14 g to 15 16 ated 17 and 18 dard 19 20 21 22
J.	This is an MPI-2.2 erratum: MPI_C_COMPLEX was added to the "Complex" reduce group.	23 24 25
4.	Section 8.5.5 on page 397, and MPI-2.2, Section 7.5.5 on page 257, C++ interface page 264, line 3. This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree now defined as int& indegree and int& outdegree in the C++ interface of MPI_DIST_GRAPH_NEIGHBORS_COUNT.	27
5.	Section 14.5.2, Table 14.2 on page 689, and MPI-2.2, Section 13.5.3, Table 13.2 page 433. This was an MPI-2.2 erratum: The MPI_C_BOOL "external32" representation is rected to a 1-byte size.	33
6.	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.	36 37 38 39
7.	Annex A.1.1 on page 845, 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.	40 41 42 43 44 45 46 47 48

1038	
B.4.2	Cha

#### nges in MPI-3.0

- 1. Section 2.6.1 on page 22, Section 17.2 on page 776 and all other chapters. The C++ bindings were removed from the standard. See errata in Section B.4.1 on page 1037 for the latest changes to the MPI C++ binding defined in MPI-2.2. This change may affect backward compatibility. 2. Section 2.6.1 on page 22, Section 16.1 on page 769 and Section 17.1 on page 775.
- 8 The deprecated functions MPI\_TYPE\_HVECTOR, MPI\_TYPE\_HINDEXED, 9 MPI\_TYPE\_STRUCT, MPI\_ADDRESS, MPI\_TYPE\_EXTENT, MPI\_TYPE\_LB, 10 MPI\_TYPE\_UB, MPI\_ERRHANDLER\_CREATE (and its callback function prototype 11 MPI\_Handler\_function), MPI\_ERRHANDLER\_SET, MPI\_ERRHANDLER\_GET, the dep-12recated special datatype handles MPI\_LB, MPI\_UB, and the constants 13 MPI\_COMBINER\_HINDEXED\_INTEGER, MPI\_COMBINER\_HVECTOR\_INTEGER, 14MPI\_COMBINER\_STRUCT\_INTEGER were removed from the standard. 15
  - This change may affect backward compatibility.
  - 3. Section 2.3 on page 10.
    - Clarified parameter usage for IN parameters. C bindings are now const-correct where backward compatibility is preserved.
    - 4. Section 2.5.4 on page 18 and Section 8.5.4 on page 390. The recommended C implementation value for MPI\_UNWEIGHTED changed from NULL to non-NULL. An additional weight array constant (MPI\_WEIGHTS\_EMPTY) was introduced.
- 255. Section 2.5.4 on page 18 and Section 9.1.1 on page 443. 26Added the new routine MPI\_GET\_LIBRARY\_VERSION to query library specific ver-27sions, and the new constant MPI\_MAX\_LIBRARY\_VERSION\_STRING.
  - 6. Sections 2.5.8, 3.2.2, 3.3, 6.9.2, on pages 20, 31, 33, 222, Sections 5.1, 5.1.7, 5.1.8, 5.1.11, 13.3 on pages 115, 142, 144, 148, 628, and Annex A.1.1 on page 845.
    - 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.
  - 7. Chapter 3 on page 29 through Chapter 19 on page 779. In the C language bindings, the array-arguments' interfaces were modified to consistently use use [] instead of \*.
    - Exceptions are MPI\_INIT, which continues to use char **\*\*\*argv** (correct because of subtle rules regarding the use of the & operator with char \*argv[]), and MPI\_INIT\_THREAD, which is changed to be consistent with MPI\_INIT.
    - 8. Sections 3.2.5, 5.1.5, 5.1.11, 5.2 on pages 36, 137, 148, 170.
      - The functions MPI\_GET\_COUNT and MPI\_GET\_ELEMENTS were defined to set the

1

2

3

4

5

6 7

16

17

18

19 20

21

22

23

24

28

29

30

31

32

33 34

35

36

37

38 39

40

41

42

43

44

4546

47

48

#-...

count argument to MPI\_UNDEFINED when that argument would overflow. The functions MPI\_PACK\_SIZE and MPI\_TYPE\_SIZE were defined to set the size argument to MPI\_UNDEFINED when that argument would overflow. In all other MPI-2.2 routines, the type and semantics of the count arguments remain unchanged, i.e., int or INTEGER.

- Section 3.2.6 on page 39, and Section 3.8 on page 80. MPI\_STATUS\_IGNORE can be also used in MPI\_IPROBE, MPI\_PROBE, MPI\_IMPROBE, and MPI\_MPROBE.
- Section 3.8 on page 80 and Section 3.10 on page 96. 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, 19.3.4, A.1.1 on pages 83, 85, 829, 845.

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.

- 2312. Section 5.1.2 on page 117 and Section 5.1.13 on page 153.  $^{24}$ The routine MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK and constant 25MPI\_COMBINER\_HINDEXED\_BLOCK were added. 2613. Chapter 6 on page 183 and Section 6.12 on page 246. 27Added nonblocking interfaces to all collective operations. 28 2914. Sections 7.4.2, 7.4.4, 12.2.7, on pages 321, 339, 557. 30 The new routines MPI\_COMM\_DUP\_WITH\_INFO, MPI\_COMM\_SET\_INFO, 31MPI\_COMM\_GET\_INFO. MPI\_WIN\_SET\_INFO, and MPI\_WIN\_GET\_INFO were 32 added. The routine MPI\_COMM\_DUP must also duplicate info hints. 33 34 15. Section 7.4.2 on page 321. 35Added MPI\_COMM\_IDUP. 36
- 16. Section 7.4.2 on page 321. 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.
- Section 7.4.2 on page 321.
   Added the MPI\_COMM\_SPLIT\_TYPE routine and the communicator split type constant MPI\_COMM\_TYPE\_SHARED.
- Section 7.6.2 on page 352.
   In MPI-2.2, communication involved in an MPI\_INTERCOMM\_CREATE operation could interfere with point-to-point communication on the parent communicator with

#### Unofficial Draft for Comment Only

the same tag or MPI\_ANY\_TAG. This interference has been removed in MPI-3.0.

1

2 3

4

5 6

7

8

9 10

11

12

13

14

15

16

17

18

19

20

21

22

37

38

39

40 41

42

43

44

45 46

47

1 2 3	19.	Section 7.8 on page 375. Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type and window names.
4 5 6	20.	Section 8.5.8 on page 408. MPI_CART_MAP can also be used for a zero-dimensional topologies.
7 8 9 10 11 12 13 14 15 16 17 18	21.	Section 8.6 on page 410 and Section 8.7 on page 422. The following neighborhood collective communication routines were added to support sparse communication on virtual topology grids: MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, MPI_NEIGHBOR_ALLTOALLW and the nonblocking variants MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALL, 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.
19 20 21 22	22.	Section 11.2.1 on page 480 and Section 11.2.1 on page 483. 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.
23 24 25	23.	Section 11.2.1 on page 480. Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.
26 27 28 29 30	24.	Chapter 12 on page 541. 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.
31 32 33	25.	Section 15.3 on page 719. A new MPI Tool Information Interface was added. The following changes are related to the Fortran language support.
34 35 36	26.	Section 2.3 on page 10, and Sections 19.1.1, 19.1.2, 19.1.7 on pages 779, 780, and 795. The new mpi_08 Fortran module was introduced.
37 38 39 40 41 42	27.	Section 2.5.1 on page 16, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 780, 783, and 795. 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.
43 44 45 46 47 48	28.	Sections 2.5.4, 2.5.5 on pages 18, 19, Sections 19.1.1, 19.1.10, 19.1.11, 19.1.12, 19.1.13 on pages 779, 805, 806, 807, 810, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 780, 783, 795. Within the mpi_08 Fortran module, choice buffers were defined as assumed-type and assumed-rank according to Fortran 2008 TS 29113 [46], and the compile-time constant MPI_SUBARRAYS_SUPPORTED was set to .TRUE With this, Fortran subscript triplets

can be used in nonblocking MPI operations; vector subscripts are not supported in nonblocking operations. If the compiler does not support this Fortran TS 29113 feature, the constant is set to .FALSE..

- 29. Section 2.6.2 on page 22, Section 19.1.2 on page 780, and Section 19.1.7 on page 795. The ierror dummy arguments are OPTIONAL within the mpi\_08 Fortran module.
- 30. Section 3.2.5 on page 36, Sections 19.1.2, 19.1.3, 19.1.7, on pages 780, 783, 795, and Section 19.3.5 on page 831. 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 55. 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 5.1 on page 115, Section 5.1.6 on page 140, and Section 19.1.15 on page 811. 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 5.1.10, 6.9.5, 6.9.7, 7.7.4, 7.8, 9.3.1, 9.3.2, 9.3.3, 16.1, 19.1.9 on pages 148, 29229, 236, 369, 375, 453, 455, 457, 769, and 797. In some routines, the dummy ar-30 gument names were changed because they were identical to the Fortran keywords TYPE and FUNCTION. The new dummy argument names must be used because the mpi and mpi\_08 modules guarantee keyword-based actual argument lists. The argument name type was changed in MPI\_TYPE\_DUP, the Fortran 34 USER\_FUNCTION of MPI\_OP\_CREATE, MPI\_TYPE\_SET\_ATTR, 35MPI\_TYPE\_GET\_ATTR, MPI\_TYPE\_DELETE\_ATTR, MPI\_TYPE\_SET\_NAME, 36 MPI\_TYPE\_GET\_NAME, MPI\_TYPE\_MATCH\_SIZE, the callback prototype defini-37 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-41 tency reasons, INOUBUF was changed to INOUTBUF in MPI\_REDUCE\_LOCAL, and intracomm to newintracomm in MPI\_INTERCOMM\_MERGE.
- 34. Section 7.7.2 on page 360. 44It was clarified that in Fortran, the flag values returned by a comm\_copy\_attr\_fn 45callback, including MPI\_COMM\_NULL\_COPY\_FN and MPI\_COMM\_DUP\_FN, are 46.FALSE. and .TRUE.; see MPI\_COMM\_CREATE\_KEYVAL. 4748

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13

14

15

1617

18

19 20

21

22

23

 $^{24}$ 

25

26

2728

31

32

33

38

39

40

42

1 2 3 4	35.	Section 9.2 on page 447. 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.
5 6 7 8	36.	Section 19.1.15 on page 811, and Section 19.1.7 on page 795. Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers in MPI operations.
9 10 11 12 13 14 15 16 17	37.	Section 19.1.16 on page 812 to Section 19.1.19 on page 821, Section 19.1.7 on page 795, and Section 19.1.8 on page 796. 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 19.1.7.
18 19 20	38.	Section 19.1.2 on page 780. Within the mpi_08 Fortran module, dummy arguments are now declared with INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces.
21 22 23	39.	Section 19.1.3 on page 783, and Section 19.1.7 on page 795. The existing mpi Fortran module must implement compile-time argument checking.
24 25 26	40.	Section 19.1.4 on page 785. The use of the mpif.h Fortran include file is now strongly discouraged.
27 28 29 30 31	41.	Section A.1.1, Table " <i>Predefined functions</i> " on page 853, Section A.1.3 on page 860, and Section A.4.5 on page 948. Within the new mpi_f08 module, all callback prototype definitions are now defined with explicit interfaces PROCEDURE(MPI) that have the BIND(C) attribute; user-written callbacks must be modified if the mpi_f08 module is used.
32 33 34 35 36	42.	Section A.1.3 on page 860. In some routines, the Fortran callback prototype names were changed from $\dots$ _FN to $\dots$ _FUNCTION to be consistent with the other language bindings.
37 38	B.5	Changes from Version 2.1 to Version 2.2
<ol> <li>39</li> <li>40</li> <li>41</li> <li>42</li> </ol>	1.	Section 2.5.4 on page 18. It is now guaranteed that predefined named constant handles (as other constants) can be used in initialization expressions or assignments, i.e., also before the call to MPI_INIT.
43 44 45 46	2.	Section 2.6 on page 21, and Section 17.2 on page 776. The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.
48 47 48	3.	Section 3.2.2 on page 31. MPI_CHAR for printable characters is now defined for C type char (instead of signed

char). This change should not have any impact on applications nor on MPI libraries (except some comment lines), because printable characters could and can be stored in any of the C types char, signed char, and unsigned char, and MPI\_CHAR is not allowed for predefined reduction operations.

		۲
4.	Section $3.2.2$ on page $31$ .	5 6
	MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL,	7
	MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and	8
	$MPI\_C\_LONG\_DOUBLE\_COMPLEX$ are now valid predefined $MPI$ data types.	9
5	Section 3.4 on page 46, Section 3.7.2 on page 60, Section 3.9 on page 90, and Section 6.1	10
э.	section 5.4 on page 40, section $5.7.2$ on page 00, section $5.9$ on page 90, and section $0.1$ on page 183.	11
	The read access restriction on the send buffer for blocking, non blocking and collective	12
	API has been lifted. It is permitted to access for read the send buffer while the	13
	operation is in progress.	14
		15
6.	Section 3.7 on page 58.	16
	The Advice to users for IBSEND and IRSEND was slightly changed.	17
7.	Section 3.7.3 on page 67.	18
	The advice to free an active request was removed in the Advice to users for	19
	MPI_REQUEST_FREE.	20 21
		21
8.	Section 3.7.6 on page 79.	22
	MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.	24
9.	Section 6.8 on page 213.	25
	"In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and	26
	MPI_ALLTOALLW for intra-communicators.	27
		28
10.	Section 6.9.2 on page 222.	29
	Predefined parameterized datatypes (e.g., returned by	30
	MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g.	31
	MPI_REAL8) have been added to the list of valid datatypes in reduction operations.	32
11.	Section $6.9.2$ on page $222$ .	33
	MPI_(U)INT{8,16,32,64}_T are all considered C integer types for the purposes of the	34
	predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran	35
	integer types. MPI_C_BOOL is considered a Logical type.	36
	MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and	37
	MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types.	38
19	Section $6.9.7$ on page $236$ .	39
12.	The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been	40 41
	added.	41
		43
13.	Section $6.10.1$ on page $238$ .	43
	The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan-	45
	dard.	46
14.	Section $6.11.2$ on page $243$ .	47
•	Added in place argument to MPI_EXSCAN.	48

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

1 2 3 4 5 6	15.	Section 7.4.2 on page 321, and Section 7.6 on page 348. Implementations that did not implement MPI_COMM_CREATE on inter-communi- cators will need to add that functionality. As the standard described the behav- ior of this operation on inter-communicators, 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.
7 8 9 10 11	16.	Section 7.4.2 on page 321. MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm is an intra-communicator. If comm is an inter-communicator it was clarified that all processes in the same local group of comm must specify the same value for group.
12 13 14 15 16 17	17.	Section 8.5.4 on page 390. 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.
18 19 20 21 22	18.	Section 8.5.5 on page 397. 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.
22 23 24 25	19.	Section 8.5.5 on page 397. Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT.
26 27 28	20.	Section 9.1.1 on page 443. The subversion number changed from 1 to 2.
29 30 31	21.	Section 9.3 on page 450, Section 16.2 on page 772, and Annex A.1.3 on page 860. Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names.
32 33 34 35 36	22.	Section 11.2.4 on page 490. Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Implementors must now also register all implementation-internal attribute deletion callbacks on MPI_COMM_SELF before returning from MPI_INIT/MPI_INIT_THREAD.
37 38 39 40 41 42 43 44	23.	Section 12.3.4 on page 566. The restriction added in MPI 2.1 that the operation MPI_REPLACE in MPI_ACCUMULATE can be used only with predefined datatypes has been removed. MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0. Also, a clarification has been made that MPI_REPLACE can be used only in MPI_ACCUMULATE, not in collective operations that do reductions, such as MPI_REDUCE and others.
45 46 47 48	24.	Section 13.2 on page 621. Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for MPI::Grequest::Start() for consistency with the rest of MPI functions that take function pointer arguments.

Done in PR449

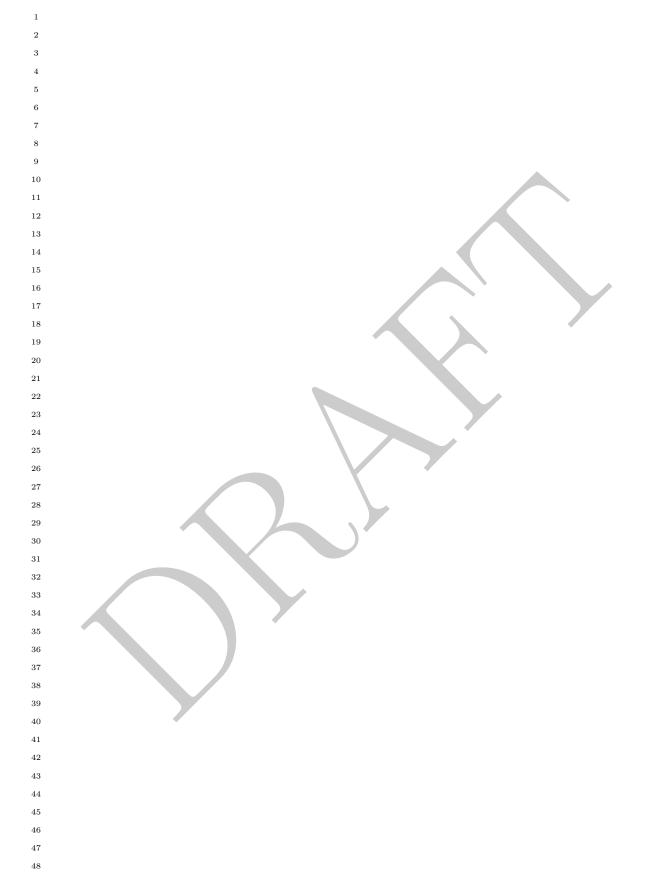
25.	Section 14.5.2 on page 688, and Table 14.2 on page 689. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes in the "interval20" perpresentation	1 2 3 4 5
26.	in the "external32" representation. Section 19.3.7 on page 837. The description was modified that it only describes how an MPI implementation be- haves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example 16.17 was replaced with three new examples 19.13, 19.14, and 19.15 on pages 837–839 explicitly detailing cross-language attribute behavior. Implementations that matched the behavior of the old example will need to be updated.	6 7 8 9 10 11 12
27.	Annex A.1.1 on page 845. Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 859).	13 14 15
28.	Annex A.1.1 on page 845. Table Named Predefined Datatypes. Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_FLOAT_COMPLEX, MPI_C_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes.	16 17 18 19 20
B.6	Changes from Version 2.0 to Version 2.1	21 22
1.	Section 3.2.2 on page 31, and Annex A.1 on page 845. In addition, the MPI_LONG_LONG should be added as an optional type; it is a synonym for MPI_LONG_LONG_INT.	23 24 25 26
2.	Section 3.2.2 on page 31, and Annex A.1 on page 845. MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings.	27 28 29 30
3.	Section 3.2.5 on page 36. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned.	31 32 33 34 35 36
4.	Section 5.1 on page 115. 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.	37 38 39 40 41 42
5.	Section 5.3 on page 178. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.	43 44 45
6.	Section 6.9.6 on page 234. If comm is an inter-communicator in MPI_ALLREDUCE, then both groups should	46 47 48

1 2		provide <b>count</b> and <b>datatype</b> arguments that specify the same type signature (i.e., it is not necessary that both groups provide the same <b>count</b> value).
3		not necessary that both groups provide the same count value).
4	7.	Section $7.3.1$ on page $310$ .
5		MPI_GROUP_TRANSLATE_RANKS and MPI_PROC_NULL: MPI_PROC_NULL is a valid
6		rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL
7		as the translated rank.
8	0	
9	8.	Section 7.7 on page 358.
10		About the attribute caching functions:
11		Advice to implementors. High-quality implementations should raise an er-
12		ror when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL
13		is used with an object of the wrong type with a call to
14		MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or
15		MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each key-
16		val, information on the type of the associated user function. ( <i>End of advice to</i>
17		implementors.)
18		
19	9.	Section 7.8 on page 375.
20		In MPI_COMM_GET_NAME: In C, a null character is additionally stored at
20		name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For-
21		tran, name is padded on the right with blank characters. resultlen cannot be larger
22		then MPI_MAX_OBJECT_NAME.
23	10	Cratical 8.4 and and 200
25	10.	Section 8.4 on page 385.
26		About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must
27		have identical values on all processes of the group of comm_old.
28	11.	Section 8.5.1 on page 386.
29		In MPI_CART_CREATE: If ndims is zero then a zero-dimensional Cartesian topology
30		is created. The call is erroneous if it specifies a grid that is larger than the group size
31		or if ndims is negative.
32		
33	12.	Section 8.5.3 on page 388.
34		In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$ , then
35		MPI_COMM_NULL is returned in all processes.
36	13	Section 8.5.3 on page 388.
37	10.	In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times
38		in the list of neighbors of a process (i.e., there may be multiple edges between two
39		processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the
40		graph). The adjacency matrix is allowed to be non-symmetric.
40		oraphi, the adjaceney matrix is anoned to be non symmetric.
42		Advice to users. Performance implications of using multiple edges or a non-
43		symmetric adjacency matrix are not defined. The definition of a node-neighbor
43		edge does not imply a direction of the communication. (End of advice to users.)
45	11	Section 8.5.5 on page 307
46	14.	Section 8.5.5 on page 397. In MPI_CARTDIM_GET and MPI_CART_GET: If comm is associated with a zero-
47		dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and
48		MPI_CART_GET will keep all output arguments unchanged.
		wing Chitiger win keep an output arguments unchanged.

15.	Section 8.5.5 on page 397. 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.	1 2 3
16.	Section 8.5.5 on page 397. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.	4 5 6 7
17.	Section 8.5.6 on page 405. 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.	8 9 10 11 12 13
18.	Section 8.5.7 on page 407. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associ- ated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology.	14 15 16 17
18.1.	Section 9.1.1 on page 443. The subversion number changed from 0 to 1.	18 19 20
19.	Section 9.1.2 on page 445. 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.	21 22 23 24 25
20.	Section 9.3 on page 450. 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.	26 27 28 29 30 31 32 33
21.	Section 11.2.1 on page 480, 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 11.10.4 on page 536.	34 35 36 37 38
22.	Section 11.2.1 on page 480. About MPI_ABORT:	39 40 41
	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> )	42 43 44 45
	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.)	46 47 48

$ \begin{array}{c} 1 \\ 2 \\ 4 \\ - error \\ 6 \\ \hline                                $	
10 24 11 12 13	. Section 12.3 on page 559. 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.
14 25 15 16 17 18	. Section 12.3 on page 559. 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.
19 26 20 21 22	. Section 12.3.4 on page 566. MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.
	. Section 14.2.8 on page 641. 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.
<sup>28</sup> 28 29 30 31	. Section 14.2.8 on page 641. 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.
32 29 33 34 35	. Section 14.3 on page 644. If a file does not have the mode MPI_MODE_SEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.
	. Section 14.5.2 on page 688. The bias of 16 byte doubles was defined with 10383. The correct value is 16383.
$38 \\ 39 \\ 40$	. 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.
42	. Section 19.1.9 on page 797. About MPI_TYPE_CREATE_F90_XXX:
43 44 45 46 47 48	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). ( <i>End of advice to implementors.</i> )	1 2 3 4 5
33. Section A.1.1 on page 845. MPI_BOTTOM is defined as void * const MPI::BOTTOM.	6 7 8
	9 10 11
	12 13 14
	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



# Bibliography

		10
[1]	Reverse domain name notation convention. https://docs.oracle.com/javase/ tutorial/java/package/namingpkgs.html. 7.4.2	11 12
[2]	V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and com- munication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2	13 14 15 16
[3]	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	17 18 19 20
[4]	Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In OON-SKI '94, page in press, 1994. 7.1	20 21 22
[5]	A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. <i>IEEE Computer</i> , 26(6):88–95, June 1993. 1.2	23 24 25 26
[6]	Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. <i>Parallel Computing</i> , 15:119–132, 1990. 1.2	27 28 29 30
[7]	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. 12.7	31 32
[8]	Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and eval- uation of primitives for parallel I/O. In <i>Proceedings of Supercomputing '93</i> , pages 452–461, 1993. 14.1	33 34 35 36
[9]	R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992. $1.2$	37 38 20
[10]	Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. <i>Parallel Computing</i> , 20(4):547–564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. 1.2	39 40 41 42 43
[11]	Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. <i>Parallel Computing</i> , 20(4):615–632, April 1994. 1.2	44 45 46 47 48

1  $\mathbf{2}$ 3 4  $\mathbf{5}$ 

1 2 3	[12]	S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole- routed networks for interprocessor communication in multicomputers. In <i>Proceedings</i> of the 1990 Supercomputing Conference, pages 647–656, 1990. 8.1
4 5 6 7	[13]	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. 8.1
8 9 10	[14]	Parasoft Corporation. Express version 1.0: A communication environment for parallel computers, 1988. 1.2
11 12 13 14	[15]	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. 17, 40
15 16 17 18 19	[16]	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. 14.1
20 21 22	[17]	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. [15], pages 282–291. 7.4.2
23 24 25 26	[18]	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
27 28 29 30	[19]	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
31 32	[20]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Concepts, June 1991. 1.2
33 34 35	[21]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992. 1.2
36 37 38 39	[22]	D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991. 7.1.2
40 41 42	[23]	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
43 44 45	[24]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org. 1.3
46 47 48	[25]	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</i> <i>Parallel Computing.</i> MIT Press, 1994. 11.1

- [26] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990. 1.2
- [27] Brice Goglin, Emmanuel Jeannot, Farouk Mansouri, and Guillaume Mercier. Hardware topology management in MPI applications through hierarchical communicators. *Parallel Computing*, 76:70–90, 2018. 7.4.2
- [28] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned multithreaded MPI communication. In ISC High Performance Conference (ISC), 2019. 4.1
- [29] Ryan E. Grant, Anthony Skjellum, and Purushotham V. Bangalore. Lightweight threading with MPI using persistent communications semantics. In Workshop on Exascale MPI (ExaMPI). Held in conjunction with the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), 2015. 4.1
- [30] 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
- [31] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. 1.2
- [32] 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. 19.1.3
- [33] 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
- [34] 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. 6.12
- [35] 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. 8.6
- [36] 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. 6.12
- [37] 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. 6.12

1 2

3

4

5

6

7 8

9

10

11 12

13

14

15 16

17

18 19

20

21

22

23

24 25

26

27

28

29 30

 $^{31}$ 

32

33

34

35

36

37 38

39

40

 $41 \\ 42$ 

43

44

45

1 2 3 4	[38]	T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimization for medical image reconstruction algorithms. In <i>Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting</i> , volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 6.12
5 6 7 8	[39]	T. Hoefler and J. L. Träff. Sparse collective operations for MPI. In <i>Proceedings of the 23rd IEEE International Parallel &amp; Distributed Processing Symposium, HIPS'09 Workshop</i> , May 2009. 8.6
9 10 11	[40]	Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI — common practice, issues, and extensions. In Cotronis et al. [15], pages 345–355. 7.4.2
12 13 14 15	[41]	Daniel J. Holmes, Bradley Morgan, Anthony Skjellum, Purushotham V. Bangalore, and Srinivas Sridharan. Planning for performance: Enhancing achievable performance for MPI through persistent collective operations. <i>Parallel Computing</i> , 81:32 – 57, 2019. 6.13
16 17 18	[42]	Institute of Electrical and Electronics Engineers, New York. <i>IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-2008</i> , 2008. 14.5.2
19 20 21 22	[43]	International Organization for Standardization, Geneva. ISO 8859-1:1987: Informa- tion processing — 8-bit single-byte coded graphic character sets — Part 1: Latin al- phabet No. 1, February 1987. 14.5.2
23 24 25	[44]	International Organization for Standardization, Geneva. ISO/IEC 9945-1:1996: Infor- mation technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language], December 1996. 11.6, 14.2.1
26 27 28 29	[45]	International Organization for Standardization, Geneva. ISO/IEC 1539-1:2010: Information technology — Programming languages — Fortran — Part 1: Base language, November 2010. 19.1.1, 19.1.2
30 31 32 33	[46]	International Organization for Standardization, Geneva. ISO/IEC TS 29113:2012: Information technology — Further interoperability of Fortran with C, December 2012. 19.1.1, 19.1.1, 19.1.2, 19.1.7, 28
34 35	[47]	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. 5.1.4
36 37 38 39	[48]	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. 14.1
40 41 42	[49]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. 8.1
43 44 45 46 47 48	[50]	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. Available online: https://docs.freebsd.org/44doc/psd/21.ipc/paper.pdf. 11.10.5

[51]	Message Passing Interface Forum. Summary of the semantics of all operation-related MPI procedures, 2020. Available online: https://www.mpi-forum.org/docs. A.2	1 2
[52]	Bradley Morgan, Daniel J. Holmes, Anthony Skjellum, Purushotham Bangalore, and Srinivas Sridharan. Planning for performance: Persistent collective operations for MPI. In <i>Proceedings of the 24th European MPI Users' Group Meeting</i> , EuroMPI '17, pages 4:1–4:11, New York, NY, USA, 2017. ACM. 6.13	3 4 5 6 7
[53]	nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2	8 9
[54]	Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 14.1	10 11
[55]	William J. Nitzberg. <i>Collective Parallel I/O</i> . PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 14.1	12 13 14
[56]	4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. 11.10.5	15 16 17
[57]	<ul> <li>Paul Pierce. The NX/2 operating system. In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pages 384–390. ACM Press, 1988.</li> <li>1.2</li> </ul>	18 19 20
[58]	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. 15.2.8	21 22 23 24
[59]	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. 14.1	25 26 27
[60]	A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, <i>Proceedings of the Fifth Distributed Memory Concurrent Computing Conference</i> , pages 767–776. IEEE Press, 1990. 1.2, 7.1.2	28 29 30 31
[61]	<ul> <li>A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992.</li> <li>1.2</li> </ul>	32 33 34 35
[62]	Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, <i>Proceedings of the Scalable Parallel Libraries Conference</i> , pages 166–173. IEEE Computer Society Press, October 1993. 7.1	36 37 38 39 40
[63]	Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator ex- tensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University — Dept. of Computer Science, August 1994. Archived at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6283. 6.2.2	41 42 43 44
[64]	Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. <i>Parallel Computing</i> , 20(4):565–596, April 1994. 7.1.2, 7.5.6	45 46 47 48

[65] The Internet Society. XDR: External Data Representation Standard, May 2006. http:  $\mathbf{2}$ //www.rfc-editor.org/pdfrfc/rfc4506.txt.pdf. 14.5.2 [66] 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. 14.1 [67] Jesper Larsson Träff. SMP-aware message passing programming. In Eighth Inter-national Workshop on High-level Parallel Programming Models and Supportive Envi-ronments (HIPS), 17th International Parallel and Distributed Processing Symposium (*IPDPS*), pages 56–65, 2003. 7.4.2 [68] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. 14.5.2 [69] D. Walker. Standards for message passing in a distributed memory environment. Tech-nical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2  $^{24}$  $^{31}$ 

# General Index

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.

absolute addresses, 20, <u>137</u>, 815 access epoch, 583action in function names,  $\underline{10}$ active, 67, 380 active target communication, 583 addresses, 152 absolute, 20, 21, 137, 815 correct use, 152 relative displacement, 20, 137 alignment, 448, 547, 550, 1033 all-reduce, 234 nonblocking, 265 persistent, 290 all-to-all, 213 nonblocking, 258 persistent, 283 array arguments, 18 assertions, 598 ASYNCHRONOUS Fortran attribute, 816 attribute, 307, 358, 837 caching,  $\underline{306}$ backward incompatibilies, 777 barrier synchronization, 189 nonblocking, 248 persistent, 272 blocking, <u>46</u>, 50, 647 I/O, 648 Blocking operation, 12 Blocking procedure,  $\underline{14}$ bounds of datatypes, 142 broadcast, 190 nonblocking, 248 persistent, 273 buffer allocation, 55 buffered, <u>47</u>, 59, <u>60</u> nonblocking, 59buffered send, 51

	language binding, <b>23</b>	12
	caching, <b>305</b> , <u>306</u> , <b>358</b>	13
	callback functions	14
	language interoperability, 836	15
	prototype definitions, 860	16
	deprecated, 867	17
	cancel, $25, 00, \underline{00}, 00, 400, 772, 1052$	
	canonical pack and unpack, 178	18
	Cartesian	19
	topology, <u>385</u> , <b>386</b>	20
	Chameleon, 2	21
	change-log, <b>1031</b>	22
	Chimp, 2	23
	choice, 19	24
	class	25
	in function names, $\underline{10}$	26
	clock synchronization, <b>446</b>	
	collective, 647	27
	split, <b>696</b>	28
1	collective communication, 183	29
	correctness, <b>296</b>	30
	file data access operations, 671	31
	neighborhood, <b>410</b>	32
	nonblocking, <b>246</b>	33
	Collective operation, $\underline{12}$	34
	Collective procedure, $\underline{14}$	
	commit, <u>146</u> COMMON blocks, <b>810</b>	35
	COMMON blocks, <b>819</b> communication, 541	36
	collective, <b>183</b>	37
	modes, $46$	38
	one-sided, 541	39
	overlap with computation, 58	40
	partitioned point-to-point, 99	41
	point-to-point, <b>29</b>	42
	RMA, <u>541</u>	43
	communicator, $34$ , $305$ , $306$	
	hidden, <b>186</b> , <b>298</b> , <b>591</b>	44
	complete operation, $\underline{12}$	45
	Completing procedure, $\underline{14}$	46
	completion, $67$	47
		48

#### General Index

1 multiple, 72  $\mathbf{2}$ completion stage,  $\underline{12}$ connected, 5363 constants, 18, 841, 845 4 context, **305**, 306, 308 5control variables 6 tools interface, 726 7 conversion, 45 8 counts, 20 9 create 10 in function names, 10 11 data, 31 12data conversion, 45 13 datatype 14derived, 15 15equivalent, 1516named, 15 portable, 1517predefined, 15 18 unnamed,  $\underline{15}$ 19datatypes, 115, 834 20deadlock 21avoid, nonblocking communication, 58 22default file error, 706 23delete in function names, 10 24deprecated interfaces, 22, 769 25derived datatype, 15, 115, 811 26disconnected, 537 27displacement, 631, 645 28distributed graph 29 topology, 385, 390 30 Dynamic Process Model, 479  $^{31}$ dynamically attached memory, 552 32 elementary datatype, <u>631</u>, 646 33 empty, 67 34 end of file, 63335 envelope, 29, 33 36 environmental inquiries, 445 37 equivalent datatypes, 15error handling, 24, 450 38 default file error handler, 634, 637, 706 39 error codes and classes, 461, 462 40 error handlers, 453, 462, 836 41 finalize, 25, 488, 537 42I/O, 706, 707 43 initial error handler, 25, 451, 481, 488, 521 44one-sided communication, 599 process failure, 25, 537 45program error, 25 46 resource error, 25 47startup, 25, 481, 521 48

transmission failure,  $\underline{25}$ establishing communication, **523** etype, <u>631</u>, 646 events tools interface, **745** exclusive scan nonblocking, **270** persistent, **295** explicit offsets, <u>648</u>, **650** exposure epoch, <u>583</u> extent of datatypes, <u>116</u>, <u>141</u>, **142** true extent, <u>144</u> external32 file data representation, <u>685</u> extra-state, **841** 

fairness, 52, <u>77</u>

file, 631 data access, 647 collective operations, 671 explicit offsets, 650 individual file pointers, 657 seek, 674 shared file pointers, 667 split collective, 675 end of file, 633 filetype, 632handle, 633 interoperability, 684 manipulation, 633 offset, 20, 632 pointer, 633 size, 633 view, 631, 632, 644 file size, 701 finalize, 486 finished, 489 Fortran language binding, 22, 779 Fortran support, 779 Freeing procedure, 14 freeing stage, 12 gather, **191** nonblocking, 250 persistent, 274

nonblocking, 250 persistent, 274 gather-to-all, 209 nonblocking, 256 persistent, 280 general datatype, <u>115</u> generalized requests, 621, <u>621</u> get in function names, <u>10</u> graph

topology, <u>385</u>, **388** group, 305, <u>306</u>, <u>308</u>, 350 group objects, 308handles, <u>16</u>, **829** hardware resource type, <u>333</u>, <u>335</u> host rank, 445 immediate, <u>14</u>, <u>60</u> inactive, 67 inclusive scan, 241 nonblocking, 269 persistent, 294 incomplete,  $\underline{14}$ ,  $\underline{60}$ Incomplete procedure, 14 independent,  $\underline{537}$ individual file pointers, 648, 657 info object, 471 file info, 641 keys, 867 values, 868 initial error handler, 25Initialization procedure,  $\underline{13}$ initialization stage, 11initiation, <u>11</u>, <u>60</u> Initiation procedure, 14inter-communication, 307, 349 inter-communicator, 307, 349 collective operations, 187, 188 interlanguage communication, 842 internal file data representation,  $\underline{685}$ interoperability, 684 intra-communication, 307, 348 intra-communicator, <u>306</u>, <u>348</u> collective operations, 186 intra-communicator objects,  $\underline{309}$ I/O, 631 IO rank, **446** is in function names, 10language binding, 21, 779 interoperability, 828 summary, 845 large count, 21, 827 displacement, 827 large count, 21lb\_marker, 130, 131, 135, <u>140</u>, 141, 146 erased, 144 local, 47local group, 322 Local procedure, 13 loosely synchronous model, 380

	lower bound, $\underline{141}$	1
	lower-bound markers, <b>140</b>	2
	magnag 94	3
	macros, <b>24</b> main thread, <u>509</u>	4
	matched receives, 85	5
	matching	6
	type, <b>42</b> , <u>149</u> , <b>700</b>	7
	matching probe, $83$	8
	memory	9
	alignment, 448, 547, 550, 1033	10
	allocation, 447, 546, 548	11
	system, 16	
	memory model, <u>542</u> , <u>582</u>	12
	separate, 542, 550	13
	unified, 542, 550	14
	message, 29	15
	data, <b>31</b>	16
	envelope, <b>33</b>	17
	modes, <b>46</b>	18
	module variables, 819	19
	mpi module	20
	Fortran support, <b>783</b>	21
	MPI operation, $\underline{11}$	22
	MPI process initialization, <u>479</u>	23
	Dynamic Process Model, <b>479</b> Sessions Model, <b>309</b> , <b>312</b> , <b>445</b> , <b>479</b> , <b>509</b> ,	23
	634	
	World Model, <b>309</b> , <b>312</b> , <b>445</b> , <b>479</b> , <b>509</b> ,	25
	<b>634</b> , <b>707</b>	26
	mpi_f08 module	27
	Fortran support, 780	28
1	MPI_SIZEOF and storage_size(), 23, 773,	29
	803-805	30
	mpiexec, 482, 505, <u>506</u>	31
	mpif.h include file	32
	Fortran support, <b>785</b>	33
	mpirun, 506	34
	multiple completions, <b>72</b>	35
	named datatype, 15	36
	names, $524$	37
	name publishing, <b>529</b>	38
	naming objects, <b>375</b>	39
	native	
	file data representation, $685$	40
	neighborhood collective communication, 410	41
	nonblocking, <b>422</b>	42
	periodic and dims==1 or 2, $1032$	43
	non-local, 47	44
	Non-local procedure, $\underline{13}$	45
	nonblocking, <b>58</b> , 559, 647	46
	communication, $\underline{58}$	47
	completion, 67	48

#### 1060

1 Fortran problems, 813  $\mathbf{2}$ I/O, 648 initiation, 60 3 persistent 4 partitioned completion, 106 5request objects, **60** 6 Nonblocking operation, 12 7 Nonblocking procedure, 148 Noncollective operation, 12 9 null handle, 67 10 null processes, 96 11 offset, 20, 632 12one-sided communication, 541 13 Fortran problems, 815 14opaque objects, 16, 834 15operation 16blocking, 12 collective, 1217complete, 1218 nonblocking,  $\underline{12}$ 19 noncollective, 1220operation-related, 13 21partitioned receive, 12 22 partitioned send, 12 23persistent, 12origin, 542 $^{24}$ 25pack, 170 26canonical, 178 27packing unit, 173 28parallel procedure, 38029partitioned completion, 106 partitioned point-to-point communication, 99 30 passive target communication, 583  $^{31}$ performance variables 32 tools interface, 733 33 persistent communication requests, 90 34 collective persistent, **271**, **429** 35 Fortran problems, 815 36 Persistent operation, 1237 PICL, 2 PMPI\_, 713 38 point-to-point communication, 29 39 portable datatype, 1540 ports, **524** 41 POSIX 42environment, 469 43 FORTRAN, 17 44 I/O, 634, 635, 648, 684, 697 model, 631 45 predefined datatype, 15 46 predefined reduction operations, 222 47private window copy, 581 48

probe, 80 probe, matching, 83 procedure blocking, 14 collective, 14 completing, 14freeing, 14 incomplete,  $\underline{14}$ initialization, 13 initiation, 14 local, 13non-local, 13nonblocking, 14 operation-related, 13starting, 13 process creation, 479 process failures, 25 process group, 34processes, 24 processor name, 446 profiling interface, 713 program error, 25 progress, 247 prototype definitions, 860 deprecated, 867 public window copy, 581 PVM, 2 rank, <u>308</u> ready, 47, 59, 60 nonblocking, 59 ready send, 51 receive, 29, 30, 34 buffer, 30complete, 58 context, 350start call, 58 reduce, **220** nonblocking, 264 persistent, 289 reduce-scatter, 238 nonblocking, 266, 268 persistent, 291, 293 reduction operations, 219, 836 predefined, 222 process-local, 236 scan, 241 user-defined, 229 related, 173 relative displacement, 20, 137 remote group, 322 Remote Memory Access see RMA, 541 removed interfaces, 22, 775

request complete I/O, 648 request objects, 60 resource error, 25RMA, 541 communication calls, 559 request-based, 574 memory model, 581 synchronization calls, 583 scan, **241** inclusive, **241** scatter, 202nonblocking, 253 persistent, 277 seek, 674 semantics, 1031 collective communications, 184 file collective, 700 collective access, 671 conflicting access, 697 consistency, 696 explicit offsets, 650 nonblocking collective, 700 shared file pointer, 667 split collective, 696 inter-communicator, 322 MPI\_COMM\_IDUP, 324 nonblocking communications, 71 nonblocking partitioned communications, 107 partitioned point-to-point communication, 100 point-to-point communication, 51 process failure, 537 semantics and correctness one-sided communication, 600 send, <u>29</u>, **30** buffer, 29complete, 58 context, 350start, 58send-receive, 40nonblocking, 65, 66 separate memory model, 542, 550, 582 sequential storage,  $\underline{152}$ serialization, 58 Sessions Model, 309, 312, 445, 479, 509, 634 set in function names,  $\underline{10}$ shared file pointers, 648, 667 shared memory allocation, 548 signals, 27

singleton init, <b>535</b>	1
size changing	2
I/O, <b>701</b>	3
source, 350	4
split collective, 647, 675	5
stage	
completion, $\underline{12}$	6
freeing, $\underline{12}$	7
initialization, $11$	8
starting, $11$	9
standard 46 59	10
nonblocking, <u>59</u>	11
standard send, 51	*
started, 489	12
Starting procedure, $\underline{13}$	13
starting processes, <b>512</b> , <b>513</b>	14
starting stage, $\underline{11}$	15
startup, <b>480</b>	16
portable, <b>506</b>	17
state, 18	18
status, <b>36</b> , <b>831</b>	
associating information, <b>628</b>	19
ignore, <b>39</b>	20
test, <b>79</b>	21
strong synchronization, $584$	22
synchronization, 541, 559	23
synchronization, 641, 665	24
RMA, 583	25
synchronous, $47$ , 59, $60$ , $67$	
nonblocking, $59$	26
synchronous send, 51	27
system memory, <u>16</u>	28
system memory, <u>ro</u>	29
tag values, 445	30
target, $542$	31
thread compliant, $\underline{484}$ , $\underline{508}$	32
threads, <b>508</b>	
thread-safe, 2, 444, 461, 462	<sup>33</sup> 2 749
timers and synchronization, 469	34
tool information interface, <b>719</b>	35
tool support, <b>713</b>	36
topologies, <b>383</b>	37
topology	38
Cartesian, <u>385</u> , <b>386</b>	39
distributed graph, <u>385</u> , <b>390</b>	40
graph, <u>385</u> , <b>388</b>	
virtual, <b>384</b>	41
transmission failures, 25	42
true extent of datatypes, 144	43
TS 29113, 19, 22, 779–784, 786–77	89 792-796 44
805, 806, 812, 816, 817, 8	
1040, 1041	620, 621,
type map, $116$	
type map, $\underline{110}$ type matching, $42$ , $\underline{149}$	47
$y_{\rm PC}$ matching, $42$ , $149$	48

```
1
       type signature, \underline{116}
\mathbf{2}
       types, 859
3
       ub_marker, 130, 131, 135, <u>140</u>, 141, 146
4
            erased, 144
\mathbf{5}
       unified memory model, 542, 550, <u>582</u>
6
       universe size, 534
7
       unnamed datatype, \underline{15}
8
       unpack, 170
            canonical, 178
9
       upper bound, \underline{141}
10
       upper-bound markers, 140
11
       user functions at process termination, 490
12
       user-defined data representations, 691
13
       user-defined reduction operations, 229
14
       verbosity levels
15
            tools interface, 720
16
       version inquiries, 443
17
       view, 631, 632, 644
18
       virtual topology, 306, 307, 384
19
20
       weak synchronization, 585
21
       window
            allocation, 546
22
            creation, 543
23
            dynamically attached memory, 552
24
            shared memory allocation, 548
25
       World Model, 309, 312, 445, 479, 509, 634,
26
                 707
27
       XDR, 46
28
29
       Zipcode, 2
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, 616, 823, 825 Attributes between languages, 837 Blocking/Nonblocking collectives do not match, 301C/Fortran handle conversion, 831 Cartesian virtual topologies, 437 Client-server code, 78 with probe, 82with probe, wrong, 82Datatype 3D array, 162 absolute addresses, 167 array of structures, 164 elaborate example, 175, 176 matching type, 149 matrix transpose, 163 union, 168Datatypes matching, 43 not matching, 43 untyped, 43Deadlock if not buffered, 54 with MPI\_Bcast, 296, 297 wrong message exchange, 53False matching of collective operations, 299 Fortran 90 copying and sequence problem, 807, 809 derived types, 811 heterogeneous communication (unsafe), 804 invalid KIND, 800 MPI\_TYPE\_MATCH\_SIZE implementation, 803 overlapping communication and computation, 823, 824, 826 register optimization, 814–816

		12
	Independence of nonblocking operations, 303	13
	Inter-communicator, 332	14
	Intercommunicator, 326	15
	Interlanguage communication, 842	16
	Intertwined matching pairs, 52	17
	Margana auchanga (ning nang) 52	18
	Message exchange (ping-pong), 53 Mixing blocking and nonblocking collective	
	operations, 299	19
	Mixing collective and point-to-point requests,	20
	301	21
	MPI_ACCUMULATE, 568	22
	MPI_Accumulate, 614, 615, 618	23
	MPI_Aint, 164	24
	MPI_Aint_add, 618	25
	MPI_Allgather, 212	26
Ν	MPI_ALLOC_MEM, 449, 450	27
	MPI_Alloc_mem, 450, 618	28
	MPI_ALLREDUCE, 235	29
	MPI_Alltoall, 301	30
	MPI_ASYNC_PROTECTS_NONBLOCKING,	31
	616	32
	MPI_BARRIER, 502	33
	MPI_Barrier, 488, 605–608, 613–616	
	MPI_Bcast, 191, 296–299	34
	MPI_BSEND, 52	35
	MPI_Buffer_attach, 56, 488 MPI_Buffer_detach, 56	36
	MPI_BYTE, 43	37
	MPI_Cancel, 488	38
	MPI_CART_COORDS, 406	39
	MPI_CART_GET, 437	40
	MPI_CART_RANK, 406	41
	MPI_CART_SHIFT, 406, 417, 437	42
	MPI_CART_SUB, 407	43
	MPI_CARTDIM_GET, 417	44
	MPI_CHARACTER, 44	45
	MPI_Comm_accept, 532, 533	40
	MPI_Comm_connect, 532, 533	
	MPI_Comm_create, 326, 343, 344, 346	47
		48

9

10

#### 1064

1	MPI_COMM_CREATE_FROM_GROUP, 498,
2	502
3	MPI_Comm_create_keyval, 373
4	MPI_Comm_dup, 345
5	MPI_Comm_get_attr, 373
6	MPI_Comm_get_parent, 522
7	MPI_Comm_group, 326, 346, 373
8	MPI_Comm_remote_size, 332
9	MPI_Comm_set_attr, 373
	MPI_COMM_SPAWN, 515
10	MPI_Comm_spawn, 515, 522 MPI_COMM_SPAWN_MULTIPLE, 520
11	MPI_Comm_spawn_multiple, 520
12	MPI_Comm_split, 332, 356, 357
13	MPI_Comm_split_type, 334, 336
14	MPI_Compare_and_swap, 616, 618
15	MPI_DIMS_CREATE, 388, 437
16	MPI_DIST_GRAPH_CREATE, 395
17	MPI_Dist_graph_create, 396
18	MPI_DIST_GRAPH_CREATE_ADJACENT,
19	395
20	MPI_F_sync_reg, 616
	MPI_FILE_CLOSE, 659, 663
21	MPI_FILE_GET_AMODE, 640
22	MPI_FILE_IREAD, 663
23	MPI_FILE_OPEN, 659, 663
24	MPI_FILE_READ, 659
25	MPI_FILE_SET_ATOMICITY, 702
26	MPI_FILE_SET_VIEW, 659, 663
27	MPI_FILE_SYNC, 703
28	MPI_Finalize, 487–489
29	MPI_FREE_MEM, 449, 450
30	MPI_Free_mem, 618 MPI_Gather, 176, 196, 197, 201
31	MPI_Gathery, 176, 197, 201 MPI_Gathery, 176, 197–201
32	MPI_GET, 564, 566
33	MPI_Get, 605–607, 612, 613
	MPI_Get_accumulate, 614, 615, 618
34	MPI_GET_ADDRESS, 138, 811, 812, 834
35	MPI_Get_address, 164, 167, 168, 175
36	MPI_GET_COUNT, 151
37	MPI_GET_ELEMENTS, 151
38	MPI_GRAPH_CREATE, 389, 402
39	MPI_GRAPH_NEIGHBORS, 402
40	MPI_GRAPH_NEIGHBORS_COUNT, 402
41	MPI_Grequest_complete, 626
42	
	MPI_Grequest_start, 626
43	MPI_Group_excl, 343
43 44	MPI_Group_excl, 343 MPI_Group_free, 326, 343, 344
44	MPI_Group_excl, 343 MPI_Group_free, 326, 343, 344 MPI_GROUP_FROM_SESSION_PSET, 498,
44 45	MPI_Group_excl, 343 MPI_Group_free, 326, 343, 344 MPI_GROUP_FROM_SESSION_PSET, 498, 500, 502
44 45 46	MPI_Group_excl, 343 MPI_Group_free, 326, 343, 344 MPI_GROUP_FROM_SESSION_PSET, 498, 500, 502 MPI_Group_incl, 326, 344, 346
44 45	MPI_Group_excl, 343 MPI_Group_free, 326, 343, 344 MPI_GROUP_FROM_SESSION_PSET, 498, 500, 502

MPI\_Ibarrier, 299-301 MPI\_Ibcast, 249, 302, 303 MPI\_INFO\_CREATE, 498 MPI\_INFO\_ENV, 482 MPI\_INFO\_SET, 498 MPI\_Intercomm\_create, 356, 357 MPI\_Iprobe, 488 MPI\_IRECV, 69-71, 78 MPI\_Irecv, 301 MPI\_ISEND, 69-71, 78 MPI\_NEIGHBOR\_ALLGATHER, 413 MPI\_NEIGHBOR\_ALLTOALL, 417 MPI\_Op\_create, 233, 234, 244 MPI\_Open\_port, 532, 533 MPI\_Pack, 175, 176 MPI\_Pack\_size, 176 MPI\_Parrived, 111 MPI\_Pready, 100, 108, 109, 111 MPI\_Precv\_init, 100, 108, 109, 111 MPI\_PROBE, 82 MPI\_Psend\_init, 100, 108, 109, 111 MPI\_Publish\_name, 532 MPI\_Put, 589, 595, 606, 608, 612, 613 MPI\_RECV, 43, 44, 52–54, 71, 82, 149 MPI\_Recv, 300 MPI\_REDUCE, 224, 227 MPI\_Reduce, 227, 228, 233, 234 MPI\_REQUEST\_FREE, 70 MPI\_Request\_free, 487 MPI\_Rget, 617 MPI\_Rput, 617 MPI\_Scan, 244 MPI\_Scatter, 206 MPI\_Scatterv, 206, 207 MPI\_SEND, 43, 44, 53, 54, 71, 82, 149 MPI\_Send, 164, 167, 168, 175, 300, 301 MPI\_SENDRECV, 162, 163, 417 MPI\_SENDRECV\_REPLACE, 406 MPI\_SESSION\_FINALIZE, 498, 500, 502 MPI\_SESSION\_GET\_NTH\_PSET, 500, 502 MPI\_SESSION\_GET\_NUM\_PSETS, 500, 502 MPI\_SESSION\_INIT, 498, 500, 502 MPI\_SSEND, 52, 71 MPI\_Test\_cancelled, 488 MPI\_TYPE\_COMMIT, 147, 162, 163, 564, 811, 812 MPI\_Type\_commit, 164, 167, 168, 175, 196-201, 207, 244 MPI\_TYPE\_CONTIGUOUS, 118, 140, 149, 151MPI\_Type\_contiguous, 196 MPI\_TYPE\_CREATE\_DARRAY, 136 MPI\_TYPE\_CREATE\_HVECTOR, 162, 163 MPI\_Type\_create\_hvector, 164, 167

MPI\_TYPE\_CREATE\_INDEXED\_BLOCK, 564MPI\_TYPE\_CREATE\_RESIZED, 811, 812 MPI\_TYPE\_CREATE\_STRUCT, 128, 140, 163, 811, 812 MPI\_Type\_create\_struct, 164, 167, 168, 175, 199, 201, 244 MPI\_TYPE\_CREATE\_SUBARRAY, 709 MPI\_TYPE\_EXTENT, 564 MPI\_TYPE\_FREE, 564 MPI\_Type\_get\_contents, 169 MPI\_Type\_get\_envelope, 169 MPI\_TYPE\_GET\_EXTENT, 162, 163, 566, 568MPI\_Type\_get\_extent, 164 MPI\_TYPE\_INDEXED, 123, 162 MPI\_Type\_indexed, 164, 167 MPI\_TYPE\_VECTOR, 119, 162, 163 MPI\_Type\_vector, 197, 198, 200, 207 MPI\_Unpack, 175, 176 MPI\_Unpublish\_name, 532 MPI\_User\_function, 234 MPI\_WAIT, 69-71, 78, 663 MPI\_Wait, 299-301 MPI\_Waitall, 301, 302, 617 MPI\_WAITANY, 78 MPI\_Waitany, 617 MPI\_WAITSOME, 78 MPI\_Win\_attach, 618 MPI\_Win\_complete, 589, 607, 608, 613 MPI\_WIN\_CREATE, 564, 566, 568 MPI\_Win\_create\_dynamic, 618 MPI\_Win\_detach, 618 MPI\_WIN\_FENCE, 564, 566, 568 MPI\_Win\_fence, 612 MPI\_Win\_flush, 606, 614-616, 618 MPI\_Win\_flush\_all, 615 MPI\_Win\_flush\_local, 605 MPI\_WIN\_FREE, 566, 568 MPI\_Win\_lock, 595, 605-608 MPI\_Win\_lock\_all, 616–618 MPI\_Win\_post, 607, 608, 613 MPI\_Win\_start, 589, 607, 608, 613 MPI\_Win\_sync, 605, 606, 614-616 shared memory windows, 616 MPI\_Win\_unlock, 595, 605-608 MPI\_Win\_unlock\_all, 617, 618 MPI\_Win\_wait, 607, 608, 613 mpiexec, 482, 507, 508 Neighborhood collective communication, 437 Non-deterministic program with MPI\_Bcast,

298

Non-overtaking messages, 52

Nonblocking operations, 69, 70	1
message ordering, 71	2
progress, 71	3
Overlapping Communicators, 302	4
	5
Partitioned Communication	6
Equal send/recv partitioning, 108	7
Partial completion notification, 111 Send with tasks, 109	8
Simple example, 100	9
Pipelining nonblocking collective operations,	10
302	11
Profiling interface	12
implementation using the C macro	13
preprocessor, 717	14
implementation using weak symbols, 716 measurement wrapper, 716	15
Progression of nonblocking collective	16
operations, 300	17 18
	18
Shared memory windows	20
MPI_Win_sync, 616	20
Threads and MPI, 509	22
Tool information interface	23
basic usage of performance variables, 743	24
listing names of all control variables, 729	25
reading the value of a control variable, 732 Topologies, 437	26
Typemap, 117–119, 123, 128, 136	27
	28
Virtual topologies, 437	29
	30
	31
	32
	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

# MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles, including info keys and values. Underlined page numbers give the location of the primary definition or use of the indexed term.

14"access\_style", 643 15"accumulate\_ops", 544 16"accumulate\_ordering", 544, 609 "alloc\_shared\_noncontig", 549, 550 17"appnum", 536 18 "arch", 482, 521 19 "argv", 482 2021"cb\_block\_size", 643 22"cb\_buffer\_size", 643 "cb\_nodes", 643 23"chunked", 64324"chunked\_item", 643 25"chunked\_size", 643 26"collective\_buffering", 643 27"command", 482 28"external32", 178, 685, 686, 688-690, 695, 29 799-801, 804, 1037, 1045 30  $^{31}$ "false", 339, 340, 643 32 "file", 482, 521 33 "file\_perm", 643 34 "filename", 643 35 "host", 482, 521 36 "hwloc://L3Cache", 494 37 38 "internal", 685, 686, 696 39 "io\_node\_list", 644 "ip\_address", 532 40 "ip\_port", 531 41 42"maxprocs", 482, 483 43 "mpi://", 494 44"mpi://SELF", 494, 495, 634 45"mpi://WORLD", 494, 495, 500 MPI::\_LONG\_LONG, 1037 46 MPI::BOOL, 1037 47MPI::COMPLEX, 1037 48

5 6

12

13

MPI::DOUBLE\_COMPLEX, 1037 MPI::F\_COMPLEX16, 1037 MPI::F\_COMPLEX32, 1037 MPI::F\_COMPLEX4, 1037 MPI::F\_COMPLEX8, 1037 MPI::INTEGER16, 1037 MPI::LONG\_DOUBLE\_COMPLEX, 1037 MPI::LONG\_LONG, 1037 MPI::REAL16, 1037 MPI\_2DOUBLE\_PRECISION, 226, 851 MPI 2INT, 226, 851 MPI\_2INTEGER, 226, 851 MPI\_2REAL, 226, 851 MPI\_ADDRESS\_KIND, 19, 20, 33, 359, 806, 837, 848 MPI\_AINT, 33, 117, 223, 553, 689, 827, 849, 850, 1043, 1045 MPI\_ANY\_SOURCE, 35, 36, 40, 41, 51, 64-67, 80-85, 94, 339, 381, 446, 847 MPI\_ANY\_TAG, 18, 35, 36, 38, 40, 41, 64-67, 80, 81, 83-85, 87, 88, 94, 96, 97, 339, 847.1039 MPI\_APPNUM, 536, 854 MPI\_ARGV\_NULL, 19, 515, 516, 520, 806, 856 MPI\_ARGVS\_NULL, 19, 519, 806, 856 "mpi\_assert\_allow\_overtaking", 340, 1033 "mpi\_assert\_exact\_length", 340, 1033 "mpi\_assert\_no\_any\_source", 339, 1033 "mpi\_assert\_no\_any\_tag", 339, 1033 MPI\_ASYNC\_PROTECTS\_NONBLOCKING, 19, 616, 780, 781, 783, 785, 788, 795, 797, 817, 848, 1042 MPI\_BAND, 222, 224, 852 MPI\_BOR, 222, 224, 852 MPI\_BOTTOM, 10, 19, 20, 39, 107, 137, 152, 153, 186, 392, 394, 517, 553, 557, 782, 784, 791, 806, 810, 813-819, 822, 834-836, 841, 847, 1049 MPI\_BSEND\_OVERHEAD, 58, 847

MPI\_BXOR, 222, 224, 852 MPI\_BYTE, <u>31</u>, 32, 43–45, 178, 223, 632, 685, 686, 689, 700, 842, 849, 850, 1045 MPI\_C\_BOOL, <u>32</u>, 223, 689, 849, 1037, 1043, 1045 MPI\_C\_COMPLEX, <u>32</u>, 223, 689, 849, 1037, 1043, 1045 MPI\_C\_DOUBLE\_COMPLEX, <u>32</u>, 223, 689, 849, 1043, 1045 MPI\_C\_FLOAT\_COMPLEX, 32, 223, 689, 849, 1043, 1045 MPI\_C\_LONG\_DOUBLE\_COMPLEX, <u>32</u>, 223, 689, 849, 1043, 1045 MPI\_CART, 397, 853 MPI\_CHAR, <u>32</u>, 45, 128, 225, 689, 724, 849, 1042, 1043 MPI\_CHARACTER, <u>31</u>, 44, 45, 225, 689, 850 MPI\_COMBINER\_CONTIGUOUS, 155, 159, 855 MPI\_COMBINER\_DARRAY, 155, 161, 855 MPI\_COMBINER\_DUP, 155, 159, 855 MPI\_COMBINER\_F90\_COMPLEX, 155, 161, 855 MPI\_COMBINER\_F90\_INTEGER, 155, 161, 855 MPI\_COMBINER\_F90\_REAL, 155, 161, 855 MPI\_COMBINER\_HINDEXED, 23, 155, 160, 855 MPI\_COMBINER\_HINDEXED\_BLOCK, 155, 160.855.1039 MPI\_COMBINER\_HINDEXED\_INTEGER, 23, 776, 1038 MPI\_COMBINER\_HVECTOR, 23, 155, 160, 855 MPI\_COMBINER\_HVECTOR\_INTEGER, 23, 776, 1038 MPI\_COMBINER\_INDEXED, 155, 160, 855 MPI\_COMBINER\_INDEXED\_BLOCK, 155, 160, 855 MPI\_COMBINER\_NAMED, 154, 155, 159, 855 MPI\_COMBINER\_RESIZED, 155, 162, 855 MPI\_COMBINER\_STRUCT, 23, 155, 160, 855 MPI\_COMBINER\_STRUCT\_INTEGER, 23, 776, 1038MPI\_COMBINER\_SUBARRAY, 155, 161, 855 MPI\_COMBINER\_VECTOR, 155, 159, 855 MPI\_COMM\_DUP\_FN, 23, 362, 853, 1041 MPI\_COMM\_NULL, <u>309</u>, <u>325</u>, <u>326</u>, <u>328–330</u>, 333-335, 338, 339, 355, 377, 387, 389, 517, 518, 538, 539, 830, 852, 1046 MPI\_COMM\_NULL\_COPY\_FN, 23, 362, 782, 836, 853, 1041 MPI\_COMM\_NULL\_DELETE\_FN, 23, 362, 853 MPI\_COMM\_PARENT, 377 MPI\_COMM\_SELF, 25, 309, 328, 341, 359, 377, 451, 480, 481, 486, 488, 490, 538, 634, 777, 851, 1033, 1044

$MPI_COMMI_I PPE_HW_GUIDED, \frac{333}{334}, $	1
851,1032	2
MPI_COMM_TYPE_HW_UNGUIDED, <u>335</u> ,	3
337,851,1032	4
MPI_COMM_TYPE_SHARED, <u>333</u> , 334, 851,	5
1039	6
MPI_COMM_WORLD, 18, 26, 34, 307,	
309-312, 320, 321, 334, 336, 337, 341,	7
344, 352, 377, 387, 445, 446, 454, 465,	8
479-483, 485-489, 491, 505, 506,	9
512–514, 518, 520, 534–538, 684, 706,	10
707, 730, 739, 777, 829, 841, 851,	11
1033, 1047	12
MPI_COMPLEX, <u>31</u> , 223, 688, 689, 798, 850	
MPI_COMPLEX16, <u>33</u> , 223, 690, 850	13
MPI_COMPLEX32, <u>33</u> , 223, 690, 850	14
MPI_COMPLEX4, <u>33</u> , 223, 690, 850	15
MPI_COMPLEX8, <u>33</u> , 223, 690, 850	16
MPI_CONGRUENT, <u>321</u> , 351, 851	17
MPI_CONVERSION_FN_NULL, <u>695</u> , 853	18
MPI_COUNT, <u>33</u> , 117, 223, 231, 689, 724, 827,	
849, 850, 1038	19
MPI_COUNT_KIND, 19, <u>20</u> , 33, 848	20
MPI_CXX_BOOL, <u>33</u> , 223, 689, 690, 850, 1037	21
MPI_CXX_DOUBLE_COMPLEX, <u>33</u> , 223, 689,	22
690, 850, 1037	23
MPI_CXX_FLOAT_COMPLEX, <u>33</u> , 223, 689,	24
690, 850, 1037	25
MPI_CXX_LONG_DOUBLE_COMPLEX, <u>33</u> ,	
223, 689, 690, 850, 1037	26
MPI_DATATYPE_NULL, 147, 852	27
	28
MPI_DISPLACEMENT_CURRENT, 645, 856,	29
1048 MDL DIST CDADU 207 252 1044	30
MPI_DIST_GRAPH, 397, 853, 1044	31
MPI_DISTRIBUTE_BLOCK, 133, 134, 856	
MPI_DISTRIBUTE_CYCLIC, 133, 134, 856	32
MPI_DISTRIBUTE_DFLT_DARG, 133, 134, 856	33
MPI_DISTRIBUTE_NONE, 133, 134, 856	34
MPI_DOUBLE, <u>32</u> , 223, 689, 724, 734, 735,	35
797, 849	36
MPI_DOUBLE_COMPLEX, <u>32</u> , 223, 688, 689,	37
798,850	
MPI_DOUBLE_INT, 226, 227, 851	38
MPI_DOUBLE_PRECISION, <u>31</u> , 223, 689, 798,	39
850	40
MPI_DUP_FN, 23, 362, <u>770</u> , 854	41
MPI_ERR_ACCESS, 463, 637, 707, 846	42
MPI_ERR_AMODE, 463, 635, 707, 846	43
MPI_ERR_ARG, 463, 845	
MPI_ERR_ASSERT, 463, 600, 846	44
MPI_ERR_BAD_FILE, 463, 707, 846	45
MPI_ERR_BASE, 449, 463, 600, 846	46
MPI_ERR_BUFFER, 463, 845	47
MPI_ERR_COMM, 463, 845	48
_ , ,	

NADE CONNA TYDE LINA CLUDED 200 201

	1	MPI_ERR_CONVERSION, 463, 695, 707, 846	N
	2	MPI_ERR_COUNT, 463, 845	
	3	MPI_ERR_DIMS, 463, 845	N
	4	MPI_ERR_DISP, 463, 600, 846	
	5	MPI_ERR_DUP_DATAREP, 463, 692, 707, 846	N
	6	MPI_ERR_FILE, 463, 707, 846	N
	7	MPI_ERR_FILE_EXISTS, 463, 707, 846	N
	8	MPI_ERR_FILE_IN_USE, 463, 637, 707, 846	N
		MPI_ERR_GROUP, 463, 845	
	9	MPI_ERR_IN_STATUS, 37, 39, 67, 74, 76, 453,	
	10	461, 463, 625, 650, 846 MPI_ERR_INFO, 463, 846	
	11	MPI_ERR_INFO_KEY, 463, 473, 846	N
	12	MPI_ERR_INFO_NOKEY, 463, 473, 846	"
	13	MPI_ERR_INFO_VALUE, 463, 473, 846	N
	14	MPI_ERR_INTERN, 452, 463, 845	IV
	15	MPI_ERR_IO, 463, 707, 846	"
	16	MPI_ERR_KEYVAL, 373, 463, 846	N
	17	MPI_ERR_LASTCODE, 462, 464–466, 766, 847	N
	18	MPI_ERR_LOCKTYPE, 463, 600, 846	N
	19	MPI_ERR_NAME, 463, 531, 846	N
	20	MPI_ERR_NO_MEM, 448, 463, 846	N
		MPI_ERR_NO_SPACE, 463, 707, 846	N
	21	MPI_ERR_NO_SUCH_FILE, 463, 637, 707, 846	N
	22	MPI_ERR_NOT_SAME, 463, 707, 846	N
	23	MPI_ERR_OP, 464, 600, 845	N
	24	MPI_ERR_OTHER, 462, 464, 845	N
	25	MPI_ERR_PENDING, 74, 464, 845	N
	26	MPI_ERR_PORT, 464, 528, 846	N
	27	MPI_ERR_PROC_ABORTED, 464, 505, 846,	N
	28		
	29	MPI_ERR_QUOTA, 464, 707, 846	N
	30	MPI_ERR_RANK, 464, 600, 845	-¶∕ "'
	31	MPI_ERR_READ_ONLY, 464, 707, 846 MPI_ERR_REQUEST, 464, 845	N
	32	MPI_ERR_RMA_ATTACH, 464, 600, 846	N
#-other	83	MPI_ERR_RMA_CONFLICT, 464, 600, 846	N
#-error!		MPI_ERR_RMA_FLAVOR, 464, 551, 600, 846	N
		MPI_ERR_RMA_RANGE, 464, 600, 846	N
Done	35	MPI_ERR_RMA_SHARED, 464, 600, 846	
	36	MPI_ERR_RMA_SYNC, 464, 600, 846	"
	37	MPI_ERR_ROOT, 464, 845	N
	38	MPI_ERR_SERVICE, 464, 531, 846	
	39	MPI_ERR_SESSION, 464	N
	40	MPI_ERR_SIZE, 464, 600, 846	N
	41	MPI_ERR_SPAWN, 464, 516, 517, 846	
	42	MPI_ERR_TAG, 464, 845	N
	43	MPI_ERR_TOPOLOGY, 464, 845	_
	44	MPI_ERR_TRUNCATE, 464, 845	N
	45	MPI_ERR_TYPE, 154, 157, 464, 707, 845	N
	46	MPI_ERR_UNKNOWN, 462, 464, 845	
	40 47	MPI_ERR_UNSUPPORTED_DATAREP, 464,	N
		707, 846	N N
	48		IV

MPI\_ERR\_UNSUPPORTED\_OPERATION, 464, 707,846 MPI\_ERR\_VALUE\_TOO\_LARGE, 464, 693, 1032 MPI\_ERR\_WIN, 464, 600, 846 MPI\_ERRCODES\_IGNORE, 19, 517, 806, 856 MPI\_ERRHANDLER\_NULL, 460, 852 MPI\_ERROR, <u>37</u>, 67, 246, 574, 832, 848, 1034, 1041 MPI\_ERRORS\_ABORT, 451, 481, 521, 1033 "mpi\_errors\_abort", <u>521</u>, <u>1034</u> MPI\_ERRORS\_ARE\_FATAL, 451, 452, 468, 521, 599, 706, 848, 1033 "mpi\_errors\_are\_fatal", 521, 1034 MPI\_ERRORS\_RETURN, 451, 452, 468, 505, 521, 706, 841, 848 'mpi\_errors\_return", 521, 1034 MPI\_F08\_STATUS\_IGNORE, 833, 857, 1041 MPI\_F08\_STATUSES\_IGNORE, 833, 857, 1041 MPI\_F\_ERROR, <u>832</u>, 848, 1031 MPI\_F\_SOURCE, 832, 848, 1031 MPI\_F\_STATUS\_IGNORE, 832, 857 MPI\_F\_STATUS\_SIZE, 19, <u>832</u>, 848, 1031 MPI\_F\_STATUSES\_IGNORE, 832, 857 MPI\_F\_TAG, <u>832</u>, 848, 1031 MPI\_FILE\_NULL, 636, 706, 707, 852 MPI\_FLOAT, <u>32</u>, 128, 221, 223, 687, 689, 849 MPI\_FLOAT\_INT, 15, 226, 227, 851 MPI\_GRAPH, 397, 853 MPI\_GROUP\_EMPTY, <u>308</u>, 314, 315, 325, 326, 328, 338, 355, 852 MPI\_GROUP\_NULL, <u>308</u>, 318, 319, 852 MPI\_HOST, 445, 851 'mpi\_hw\_resource\_type", 334, 335, 1032 MPI\_IDENT, <u>312</u>, 321, 851 MPI\_IN\_PLACE, 19, 186, 216, 785, 806, 847 MPI\_INFO, 272 \mpicdecl{MPI\\_Info} MPI\_INFO\_ENV, 478, 482, 483, 491, 851, 1040 MPI\_INFO\_NULL, 395, 448, 477, 516, 526, 635, 637, 646, 852 'mpi\_initial\_errhandler", 481, 482, 521, 1034 MPI\_INT, 15, <u>32</u>, 116, 222, 687–689, 723, 724, 727, 733, 737, 797, 841, 843, 849 MPI\_INT16\_T, 32, 223, 689, 849, 1043, 1045 MPI\_INT32\_T, <u>32</u>, 223, 689, 724, 849, 1043, 1045MPI\_INT64\_T, <u>32</u>, 223, 689, 724, 849, 1043, 1045 MPI\_INT8\_T, <u>32</u>, 223, 689, 849, 1043, 1045 MPI\_INTEGER, <u>31</u>, 42, 43, 223, 689, 797, 798, 843, 850 MPI\_INTEGER1, <u>32</u>, 223, 690, 850 MPI\_INTEGER16, 223, 690, 850 MPI\_INTEGER2, <u>32</u>, 223, 688, 690, 850

MPI\_INTEGER4, <u>32</u>, 223, 690, 850 MPI\_INTEGER8, 33, 223, 690, 802, 850 MPI\_INTEGER\_KIND, 19, 848 MPI\_IO, 445, 446, 851 MPI\_KEYVAL\_INVALID, 363, 364, 365, 847 MPI\_LAND, 222, 223, 852 MPI\_LASTUSEDCODE, 465, 854 MPI\_LB, 23, 776, 1038 MPI\_LOCK\_EXCLUSIVE, 592, 847 MPI\_LOCK\_SHARED, 592, 593, 847 MPI\_LOGICAL, 31, 223, 689, 850 MPI\_LONG, 32, 222, 689, 849 MPI\_LONG\_DOUBLE, 32, 223, 689, 849 MPI\_LONG\_DOUBLE\_INT, 226, 851 MPI\_LONG\_INT, 226, 227, 851 MPI\_LONG\_LONG, 32, 223, 849, 1045 MPI\_LONG\_LONG\_INT, 32, 223, 689, 849, 1045MPI\_LOR, 222, 223, 852 MPI\_LXOR, 222, 223, 852 MPI\_MAX, 221-223, 244, 852 MPI\_MAX\_DATAREP\_STRING, 19, 647, 692, 848 MPI\_MAX\_ERROR\_STRING, 19, 461, 466, 848 MPI\_MAX\_FROM\_GROUP\_STRINGTAG, 355 MPI\_MAX\_FROM\_GROUP\_TAG, 338, 848 MPI\_MAX\_INFO\_KEY, 19, 463, 471, 474, 475, 848 MPI\_MAX\_INFO\_VAL, 19, 463, 471, 848 MPI\_MAX\_LIBRARY\_VERSION\_STRING, 19, 444, 848, 1038 MPI\_MAX\_OBJECT\_NAME, 19, 376-379, 848, 1040, 1046 MPI\_MAX\_PORT\_NAME, 19, 526, 848 MPI\_MAX\_PROCESSOR\_NAME, 19, 447, 848, 1047 MPI\_MAX\_PSET\_NAME\_LEN, 497, 848 MPI\_MAXLOC, 222, 225, 226, 229, 852 MPI\_MESSAGE\_NO\_PROC, 84, 87, 88, 97, 847, 1039 MPI\_MESSAGE\_NULL, 84, 86-88, 852, 1039 MPI\_MESSAGE\_PROC\_NULL, 84 MPI\_MIN, 222, 223, 852 "mpi\_minimum\_memory\_alignment", 448, 547, 550.1033 MPI\_MINLOC, 222, 225, 226, 229, 852 MPI\_MODE\_APPEND, 634, 635, 855 MPI\_MODE\_CREATE, 634, 635, 644, 855 MPI\_MODE\_DELETE\_ON\_CLOSE, 634-636, 855 MPI\_MODE\_EXCL, 634, 635, 855 MPI\_MODE\_NOCHECK, 593, 598, 599, 855 MPI\_MODE\_NOPRECEDE, 588, 598, 599, 855 MPI\_MODE\_NOPUT, 598, 599, 855

MPI\_MODE\_NOSTORE, 598, 599, 855 MPI\_MODE\_NOSUCCEED, 598, 599, 855  $\mathbf{2}$ MPI\_MODE\_RDONLY, 634, 635, 640, 855 3 MPI MODE RDWR, 634, 635, 855 4 MPI\_MODE\_SEQUENTIAL, 634, 635, 638, 645, 5650, 658, 674, 699, 855, 1048 6 MPI\_MODE\_UNIQUE\_OPEN, 634, 635, 855 7 MPI\_MODE\_WRONLY, 634, 635, 855 MPI\_NO\_OP, 544, 571–573, 852, 1035 MPI\_NULL\_COPY\_FN, 23, 362, 770, 854 9 MPI\_NULL\_DELETE\_FN, 23, 363, 770, 854 10 MPI\_OFFSET, 33, 223, 689, 849, 850, 1043, 11 104512MPI\_OFFSET\_KIND, 19, 20, 33, 701, 806, 848 13 MPI\_OP\_NULL, 233, 852 14MPI\_ORDER\_C, 18, 130, 133, 134, 856 15MPI\_ORDER\_FORTRAN, 18, 130, 133, 134, 16856 MPI\_PACKED, 15, 31, 32, 42, 43, 172-174, 178, 17688, 689, 842, 849, 850 18 MPI\_PROC\_NULL, 30, 34, 36, 81, 84, 87, 88, 1996, 97, 188, 191, 193, 195, 204, 206, 20222, 311, 406, 410, 445, 446, 550, 551, 21560, 847, 1039, 1046, 1048 22#-other MPI\_PROD, 222, 223, 852 23MPI\_REAL, 31, 42, 223, 688, 689, 797, 798, #-error! 804, 850  $^{24}$ #-update5 MPI\_REAL16, 32, 223, 690, 850 25MPI\_REAL2, <u>32</u>, 223, 690, 850 26#-TODO MPI\_REAL4, 32, 223, 690, 797, 802, 850 27Merge of MPI\_REAL8, 32, 223, 690, 797, 850, 1043 PR400 28MPI\_REPLACE, 568, 569, 571, 573, 616, 852, 291044, 1048 Is PR400 30 MPI\_REQUEST\_NULL, 67-70, 73-76, 624, 852 incomplete?  $^{31}$ MPI\_ROOT, 188, 847 MPI\_SEEK\_CUR, 666, 674, 856 32 MPI\_SEEK\_END, 666, 674, 856 33 MPI\_SEEK\_SET, 666, 667, 674, 856 34MPI\_SESSION\_NULL, 494, 852 35"mpi\_shared\_memory", 334, 1032 36 MPI\_SHORT, 32, 222, 689, 849 37 MPI\_SHORT\_INT, 226, 851 38 MPI\_SIGNED\_CHAR, 32, 223, 225, 689, 849, 39 1045 MPI\_SIMILAR, <u>312</u>, <u>321</u>, <u>351</u>, <u>851</u> 40 "mpi\_size", 494, 498, 502 41 MPI\_SOURCE, 37, 246, 832, 848, 1034, 1041 42MPI\_STATUS\_IGNORE, 10, 19, 39, 40, 623, 43 650, 784, 806, 832, 833, 841, 856, 857, 441039 45MPI\_STATUS\_SIZE, 19, 37, 786, 848, 1041 46MPI\_STATUSES\_IGNORE, 18, 19, 39, 40, 623, 625, 806, 832, 833, 856, 857 4748

1	MPI_SUBARRAYS_SUPPORTED, 19, 780, 781,
2	784-788, 792-795, 807-809, 848, 1040
3	MPI_SUBVERSION, 19, 444, 857
4	MPI_SUCCESS, 22, 24, 67, 74, 76, 362–364,
	366, 367, 370, 371, 388, 461–463,
5	467-469, 481, 517, 695, 720, 729, 738,
6	740-742, 748, 752, 753, 757, 764, 766,
7	768, 770, 845, 1033, 1034
8	MPI_SUM, 222, 223, 568, 841, 852
9	MPI_T_BIND_MPI_COMM, 721, 858
	MPI_T_BIND_MPI_COMM, <u>121</u> , 838 MPI_T_BIND_MPI_DATATYPE, <u>721</u> , 858
10	MPL_T_DIND_MPL_EDDUANDLED_701_000
11	MPI_T_BIND_MPI_ERRHANDLER, 721, 858
12	MPI_T_BIND_MPI_FILE, <u>721</u> , 858
13	MPI_T_BIND_MPI_GROUP, <u>721</u> , 858
14	MPI_T_BIND_MPI_INFO, <u>721</u> , 858
	MPI_T_BIND_MPI_MESSAGE, <u>721</u> , 858
15	MPI_T_BIND_MPI_OP, <u>721</u> , 858
16	MPI_T_BIND_MPI_REQUEST, 721, 858
17	MPI_T_BIND_MPI_SESSION, 721
18	MPI_T_BIND_MPI_WIN, <u>721</u> , 858
19	MPI_T_BIND_NO_OBJECT, <u>721</u> , 728, 730,
20	737, 739, 752, 753, 858
	MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE,
21	748, <u>749</u> , 859
22	MPI_T_CB_REQUIRE_MPI_RESTRICTED,
23	748, 748, 749, 859
24	MPI_T_CB_REQUIRE_NONE, 748, 859
25	MPI_T_CB_REQUIRE_THREAD_SAFE, 748,
26	749,859
27	MPI_T_CVAR_HANDLE_NULL, 731, 857
	MPI_T_ENUM_NULL, 727, 737, 752, 857
28	MPI_T_ERR_CANNOT_INIT, 768, 847
29	MPI_T_ERR_CVAR_SET_NEVER, 732, 768,
30	847
31	MPI_T_ERR_CVAR_SET_NOT_NOW, <u>732</u> ,
32	768, 847
33	MPI_T_ERR_INVALID, <u>768</u> , 847, 1036
34	MPI_T_ERR_INVALID_HANDLE, 739, 757,
	768, 847
35	MPI_T_ERR_INVALID_INDEX, 23, 748, 752,
36	<u>768</u> , 773, 847, 1033
37	MPI_T_ERR_INVALID_ITEM, 23, <u>768</u> , 773,
38	847, 1033
39	MPI_T_ERR_INVALID_NAME, 729, 738, 753,
40	764, <u>768</u> , 847, 1036
	MPI_T_ERR_INVALID_SESSION, <u>768</u> , 847
41	MPI_T_ERR_MEMORY, <u>768</u> , 847
42	
43	MPI_T_ERR_NOT_ACCESSIBLE, <u>749</u> , 750,
44	768, 847
45	MPI_T_ERR_NOT_INITIALIZED, 768, 847
46	MPI_T_ERR_NOT_SUPPORTED, <u>748</u> , 768,
47	MPI_T_ERR_OUT_OF_HANDLES, 768, 847
48	MPI_T_ERR_OUT_OF_SESSIONS, 768, 847

MPI_T_ERR_PVAR_NO_ATOMIC, <u>742</u> , 768,
847 MPI_T_ERR_PVAR_NO_STARTSTOP, <u>740</u> ,
741, 768, 847
MPI_T_ERR_PVAR_NO_WRITE, <u>742</u> , 768, 847
MPI_T_PVAR_ALL_HANDLES, <u>740</u> , 741–743,
858 MPI_T_PVAR_CLASS_AGGREGATE, <u>734</u> , 735,
MPI_T_PVAR_CLASS_COUNTER, 734, 858
MPI_T_PVAR_CLASS_GENERIC, 735, 858
MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>734</u> , 858
MPI_T_PVAR_CLASS_LEVEL, <u>733</u> , 858
MPI_T_PVAR_CLASS_LOWWATERMARK,
$\underline{734}, 858$
MPI_T_PVAR_CLASS_PERCENTAGE, <u>734</u> , 858
MPI_T_PVAR_CLASS_SIZE, <u>734</u> , 858
MPI_T_PVAR_CLASS_STATE, <u>733</u> , 858 MPI_T_PVAR_CLASS_TIMER, <u>735</u> , 858
MPI_T_PVAR_CLASS_TIMER, <u>735</u> , 858
MPI_T_PVAR_HANDLE_NULL, 740, 857
MPI_T_PVAR_SESSION_NULL, 738, 857
MPI_T_SCOPE_ALL, 728, 858
MPI_T_SCOPE_ALL_EQ, <u>728</u> , 732, 858
MPI_T_SCOPE_CONSTANT, 728, 858
MPL T SCOPE GROUP 728 858
MPI_T_SCOPE_GROUP_EQ, <u>728</u> , 732, 858 MPI_T_SCOPE_LOCAL, <u>728</u> , 858
MPI_T_SCOPE_LOCAL, 728, 858
MPL T SCOPE READONLY, 728, 858
MPI_T_SOURCE_ORDERED, 747, 859
MPI_T_SOURCE_UNORDERED, 747, 859
MPI_T_VERBOSITY_MPIDEV_ALL, 720, 721,
857
MPI_T_VERBOSITY_MPIDEV_BASIC, <u>721</u> ,
MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>721</u> , 857
MPI_T_VERBOSITY_TUNER_ALL, <u>721</u> , 857
MPI_T_VERBOSITY_TUNER_BASIC, 721, 857
MPI_T_VERBOSITY_TUNER_DETAIL, 721,
857
MPI_T_VERBOSITY_USER_ALL, <u>721</u> , 857
MPI_T_VERBOSITY_USER_BASIC, 720, 721,
857
MPI_T_VERBOSITY_USER_DETAIL, 720, 721,
857 MDI TAC 27 246 222 242 1024 1041
MPI_TAG, <u>37</u> , 246, 832, 848, 1034, 1041
MPI_TAG_UB, 34, 445, 837, 840, 851
MPI_THREAD_FUNNELED, 483, 855
"MPI_THREAD_FUNNELED", 493
MPI_THREAD_MULTIPLE, 483, 484, 486, 855, 1036
"MPI_THREAD_MULTIPLE", 493
MPI_THREAD_SERIALIZED, 483, 855

"MPI\_THREAD\_SERIALIZED", 493 MPI\_THREAD\_SINGLE, 483-485, 493, 855 "MPI\_THREAD\_SINGLE", 493 "mpi\_thread\_support\_level", 493 MPI\_TYPE\_DUP\_FN, 370, 853 MPI\_TYPE\_NULL\_COPY\_FN, 370, 853 MPI\_TYPE\_NULL\_DELETE\_FN, 370, 853, 1041 MPI\_TYPECLASS\_COMPLEX, 803, 856 MPI\_TYPECLASS\_INTEGER, 803, 856 MPI\_TYPECLASS\_REAL, 803, 856 MPI\_UB, 4, 23, 776, 1038 MPI\_UINT16\_T, 32, 223, 689, 849, 1043, 1045 MPI\_UINT32\_T, 32, 223, 689, 724, 849, 1043, 1045 MPI\_UINT64\_T, <u>32</u>, 223, 689, 724, 849, 1043, 1045MPI\_UINT8\_T, <u>32</u>, 223, 689, 849, 1043, 1045 MPI\_UNDEFINED, 21, 38, 39, 56, 73, 74, 76, 77, 140, 143, 146, 151, 175, 310, 311, 329, 330, 333, 397, 408, 409, 687, 693, 799, 847, 1039, 1045 MPI\_UNEQUAL, <u>312</u>, 321, 351, 851 MPI\_UNIVERSE\_SIZE, 512, 534, 535, 854 MPI\_UNSIGNED, <u>32</u>, 222, 689, 724, 733–735, 849 MPI\_UNSIGNED\_CHAR, 32, 223, 225, 689, 849 MPI\_UNSIGNED\_LONG, <u>32</u>, 223, 689, 724, 734, 735, 849 MPI\_UNSIGNED\_LONG\_LONG, <u>32</u>, 223, 689, 724, 734, 735, 849, 1045 MPI\_UNSIGNED\_SHORT, <u>32</u>, 222, 689, 849 MPI\_UNWEIGHTED, 19, 392, 394-396, 403, 405, 806, 856, 1038, 1044 MPI\_VAL, 16, 829 MPI\_VERSION, 19, 444, 857 MPI\_WCHAR, <u>32</u>, 225, 379, 688, 689, 849, 1045 MPI\_WEIGHTS\_EMPTY, 19, 392, 394, 806, 856, 1038 MPI\_WIN\_BASE, 556, 840, 854 MPI\_WIN\_CREATE\_FLAVOR, 556, 854 MPI\_WIN\_DISP\_UNIT, 556, 854 MPI\_WIN\_DUP\_FN, 366, 853 MPI\_WIN\_FLAVOR\_ALLOCATE, 557, 854 MPI\_WIN\_FLAVOR\_CREATE, 557, 854 MPI\_WIN\_FLAVOR\_DYNAMIC, 557, 854 MPI\_WIN\_FLAVOR\_SHARED, 551, 557, 854 MPI\_WIN\_MODEL, 556, 583, 854 MPI\_WIN\_NULL, 555, 852 MPI\_WIN\_NULL\_COPY\_FN, 366, 853 MPI\_WIN\_NULL\_DELETE\_FN, 367, 853 MPI\_WIN\_SEPARATE, 557, 582, 583, 602, 854 MPI\_WIN\_SIZE, 556, 854 MPI\_WIN\_UNIFIED, 557, 583, 603, 611, 854

'ED", 493	MPI_WTIME_IS_GLOBAL, 445, 446, 470, 837,	1	
83-485, 493, 855	851	2	
, 493	"mpix://UNIVERSE", 494	3	
", 493	"native", 685, 686, 695, 696	4	
, 853 _FN, <u>370,</u> 853	"nb_proc", 644	5	
E_FN, <u>370</u> , 853,	"no_locks", 544, 556	6	
<u> </u>	"none", 609	7	
LEX, 803, 856	"num_io_nodes", 644	8	
ER, 803, 856		9	
803, 856	"path", 521	10	
200 040 1049 1045	"random", 643	11	
589, 849, 1043, 1045	"rar", 609	12	
589, 724, 849, 1043,	"raw", 609	13	
589,724,849,1043,	"read_mostly", 643	14	
, 124, 045, 1045,	"read_once", 643	15	
89, 849, 1043, 1045	"reverse_sequential", 643	16	
39, 56, 73, 74, 76,	"same_disp_unit", 544, 1036	17	
,151,175,310,311,	"same_op", 544	18	
7, 408, 409, 687, 693,	"same_op_no_op", 544	19	
045	"same_size", 544, 1035	20	
351, 851	"sequential", 643	21	
2, 534, 535, 854	"soft", 482, 516, 521	22	
689, 724, 733–735,	"striping_factor", 644	23	
$\underline{2}, 223, 225, 689, 849$	"striping_unit", 644	24	
$\underline{2}, 223, 223, 089, 724, 734, $	Uthread Involut 199	25	
<u>12</u> , 220, 005, 124, 104,	"thread_level", 482 "true", 339, 340, 556, 643	26	
ONG, <u>32</u> , 223, 689,	100, 300, 340, 500, 045	20	
9, 1045	"war", 609		
<u>32</u> , 222, 689, 849	"waw", 609	28	
392, 394–396, 403,	"wdir", 482, 521	29	#-other
38, 1044	"write_mostly", 643	00	#-error!
	"write_once", 643	31	#-enoi:
57	Are these constants really new constants?		
9, 688, 689, 849, 1045 19, 392, 394, 806,	+ = referenced in a new change-log entry		
10, 002, 004, 000,	+ MPI_COMM_TYPE_HW_GUIDED		
, 854	+ MPI_COMM_TYPE_HW_UNGUIDED + MPI_ERRORS_ABORT		
OR, 556, 854	+ MPI_ERR_PROC_ABORTED		
6,854	- MPI_ERR_SESSION> Done in PR461		
853	+ MPI_ERR_VALUE_TOO_LARGE		#-TODO
CATE, 557, 854	+ MPI_F_ERROR		Merge of
TE, 557, 854	+ MPI_F_SOURCE + MPI_F_STATUS_SIZE		PR400
AMIC, 557, 854	+ MPI_F_STATUS_SIZE Done		ls PR400
ED, 551, 557, 854	(MPI_INFO WRONG !!!! and removed: DC	ONE)	incomplete?
83, 854	MPI_MAX_FROM_GROUP_STRINGTAG Thi		
N, <u>366</u> , 853	MPI_MAX_FROM_GROUP_TAG or t	hat on	e????
E_FN, <u>367</u> , 853	+ MPI_MAX_PSET_NAME_LEN - MPI_MESSAGE_PROC_NULL> use constskip	- Dor	e in PR449
7, 582, 583, 602, 854	- MPI_SESSION_NULL -> Done in PR461		5 III I IN <del>7</del> 70
., 302, 300, 002, 001	- MPI_T_BIND_MPI_SESSION		
583,603,611,854	- MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE		
,, <del></del> , <del>,</del> <del>,</del>	- MPI_T_CB_REQUIRE_MPI_RESTRICTED		
	- MPI_T_CB_REQUIRE_NONE		
Unomicial Draft for	C - MPI_T_CB_REQUIRE_THREAD_SAFE - MPI_T_ERR_NOT_ACCESSIBLE		
	- MPI_T_ERR_NOT_SUPPORTED		
	- MPI_T_SOURCE_ORDERED		
			I

MPI\_T\_SOURCE\_UNORDERED

# MPI Declarations Index

10

11 12 This index refers to declarations needed in C and Fortran, 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). Fortran defined types are shown as TYPE(name).

13	MPI_Aint, <u>20</u> , 20, 21, 33, <u>117</u> , 120, 123, 127,
14	137-139, 142-145, 155, 178-180, 543,
15	546, 548, 552, 553, 556, 560, 563, 567,
16	569, 572-574, 576, 577, 579, 687, 692,
17	747,806,837,849,859
18	MPI_Comm, 16, <u>30</u> , 312, 319, 320, 322–324,
19	$\underline{327}, 329, 332, 338, 340, 341, 350, 351, 350, 351, 351, 351, 351, 351, 351, 351, 351$
	353, 355, 361, 364, 365, 496, 851, 852,
20	859, 860 TYPE(MPI_Comm), 793, 851, 852, 860
21	MPI_Count, <u>20</u> , 21, 33, 746, 747, 849, 859, <b>103</b>
22	MPI_Count, <u>20</u> , 21, 35, 740, 747, 849, 859, 1058 MPI_Datatype, <u>117</u> , 370, 815, 849–852, 859, 860
23	TYPE(MPI_Datatype), 849–852, 860
24	MPI_Errhandler, <u>453</u> , 454–460, 830, 848, 852,
25	859, 860
26	TYPE(MPI_Errhandler), 848, 852, 860
27	MPI_F08_status, <u>832</u> , 857, 859, 1041
28	MPI_File, 457, 458, <u>633</u> , 636–639, 641, 642,
29	644,646,650656,658670,672674,
30	676-683,  687,  697,  698,  830,  852,  859,
	000
31	860
31 32	TYPE(MPI_File), 852, 860
32	TYPE(MPI_File), 852, 860 MPI_Fint, <u>829</u> , 857, 859, 1045
32 33	TYPE(MPI_File), 852, 860 MPI_Fint, 829, 857, 859, 1045 MPI_Group, <u>310</u> , 311–316, 318, 351, 496, 557,
32 33 34	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, } \underline{829, 857, 859, 1045} \\ \mbox{MPI_Group, } \underline{310, 311} \mbox{-} 316, 318, 351, 496, 557, \\ \mbox{588, 589, 639, 830, 852, 859, 860} \end{array}$
32 33 34 35	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, } \underline{829, 857, 859, 1045} \\ \mbox{MPI_Group, } \underline{310, 311-316, 318, 351, 496, 557, } \\ \mbox{588, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), 852, 860} \end{array}$
32 33 34 35 36	TYPE(MPI_File), 852, 860 MPI_Fint, <u>829</u> , 857, 859, 1045 MPI_Group, <u>310</u> , 311–316, 318, 351, 496, 557, 588, 589, 639, 830, 852, 859, 860 TYPE(MPI_Group), 852, 860 MPI_Info, 447, <u>471</u> , 472–477, 494, 498, 513,
32 33 34 35 36 37	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, 829, 857, 859, 1045} \\ \mbox{MPI_Group, 310, 311-316, 318, 351, 496, 557, } \\ \mbox{588, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), 852, 860} \\ \mbox{MPI_Info, 447, 471, 472-477, 494, 498, 513, } \\ \mbox{516, 518, 525, 529-531, 538, 558, 633, } \end{array}$
32 33 34 35 36 37 38	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, 829, 857, 859, 1045} \\ \mbox{MPI_Group, 310, 311-316, 318, 351, 496, 557, } \\ \mbox{588, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), 852, 860} \\ \mbox{MPI_Info, 447, 471, 472-477, 494, 498, 513, } \\ \mbox{516, 518, 525, 529-531, 538, 558, 633, } \\ \mbox{636, 641, 642, 644, 772, 830, 851, 852, } \end{array}$
32 33 34 35 36 37	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, 829, 857, 859, 1045} \\ \mbox{MPI_Group, 310, 311-316, 318, 351, 496, 557, } \\ \mbox{588, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), 852, 860} \\ \mbox{MPI_Info, 447, 471, 472-477, 494, 498, 513, } \\ \mbox{516, 518, 525, 529-531, 538, 558, 633, } \\ \mbox{636, 641, 642, 644, 772, 830, 851, 852, } \\ \mbox{859, 860, 1048} \end{array}$
32 33 34 35 36 37 38	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, 829, 857, 859, 1045} \\ \mbox{MPI_Group, 310, 311-316, 318, 351, 496, 557, } \\ \mbox{588, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), 852, 860} \\ \mbox{MPI_Info, 447, 471, 472-477, 494, 498, 513, } \\ \mbox{516, 518, 525, 529-531, 538, 558, 633, } \\ \mbox{636, 641, 642, 644, 772, 830, 851, 852, } \end{array}$
32 33 34 35 36 37 38 39	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, $\underline{829}, 857, 859, 1045} \\ \mbox{MPI_Group, $\underline{310}, 311-316, 318, 351, 496, 557, $588, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), $52, 860} \\ \mbox{MPI_Info, $447, $\underline{471}, $472-477, $494, 498, 513, $516, 518, 525, 529-531, 538, 558, 633, $636, 641, 642, 644, 772, 830, 851, 852, $859, 860, 1048} \\ \mbox{TYPE(MPI_Info), $851, 852, 860} \end{array}$
32 33 34 35 36 37 38 39 40	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, 829, 857, 859, 1045} \\ \mbox{MPI_Group, 310, 311-316, 318, 351, 496, 557, 588, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), 852, 860} \\ \mbox{MPI_Info, 447, 471, 472-477, 494, 498, 513, 516, 518, 525, 529-531, 538, 558, 633, 636, 641, 642, 644, 772, 830, 851, 852, 859, 860, 1048} \\ \mbox{TYPE(MPI_Info), 851, 852, 860} \\ \mbox{MPI_Message, 83, 830, 847, 852, 859, 860, 1039} \\ \mbox{TYPE(MPI_Message), 847, 852, 860} \\ \mbox{MPI_Offset, 20, 20, 21, 33, 637-639, 644, 646, } \end{array}$
32 33 34 35 36 37 38 39 40 41	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, $\underline{829}, 857, 859, 1045} \\ \mbox{MPI_Group, $\underline{310}, 311-316, 318, 351, 496, 557, $588, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), $52, 860} \\ \mbox{MPI_Info, $447, $\underline{471}, $472-477, $494, 498, 513, $516, 518, 525, 529-531, 538, 558, 633, $636, 641, 642, 644, 772, 830, 851, 852, $859, 860, 1048} \\ \mbox{TYPE(MPI_Info), $851, 852, 860} \\ \mbox{MPI_Message, $\underline{83}, 830, 847, 852, 859, 860, 1039} \\ \mbox{TYPE(MPI_Message), $847, 852, 860} \\ \mbox{MPI_Offset, $\underline{20}, 20, 21, 33, 637-639, 644, 646, $650-656, 665-667, 674, 676, 678, 693, $ \end{array}$
32 33 34 35 36 37 38 39 40 41 42	$\begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, $\underline{829}, 857, 859, 1045} \\ \mbox{MPI_Group, $\underline{310}, 311-316, 318, 351, 496, 557, $58, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), 852, 860} \\ \mbox{MPI_Info, $447, $\underline{471}, $472-477, $494, 498, 513, $516, 518, 525, 529-531, 538, 558, 633, $636, 641, 642, 644, 772, 830, 851, 852, $859, 860, 1048} \\ \mbox{TYPE(MPI_Info), $51, 852, 860} \\ \mbox{MPI_Message, $\underline{83}, 830, 847, 852, 859, 860, 1039} \\ \mbox{TYPE(MPI_Message), $847, 852, 859, 860, 1039} \\ \mbox{TYPE(MPI_Message), $847, 852, 860} \\ \mbox{MPI_Offset, $\underline{20}, 20, 21, 33, 637-639, 644, 646, $650-656, 665-667, 674, 676, 678, 693, $\underline{701}, 747, 829, 849, 859} \\ \end{array}$
32 33 34 35 36 37 38 39 40 41 42 43	$\label{eq:spectral_states} \begin{array}{l} \mbox{TYPE(MPI_File), 852, 860} \\ \mbox{MPI_Fint, $\underline{829}, 857, 859, 1045} \\ \mbox{MPI_Group, $\underline{310}, 311-316, 318, 351, 496, 557, $58, 589, 639, 830, 852, 859, 860} \\ \mbox{TYPE(MPI_Group), $52, 860} \\ \mbox{MPI_Info, $447, $\underline{471}, $472-477, $494, 498, 513, $516, 518, 525, 529-531, 538, 558, 633, $636, 641, 642, 644, 772, 830, 851, 852, $859, 860, 1048} \\ \mbox{TYPE(MPI_Info), $51, 852, 860} \\ \mbox{MPI_Message, $\underline{83}, 830, 847, 852, 859, 860, 1039} \\ \mbox{TYPE(MPI_Message), $847, 852, 859, 860, 1039} \\ \mbox{TYPE(MPI_Message), $847, 852, 860} \\ \mbox{MPI_Offset, $\underline{20}, 20, 21, 33, 637-639, 644, 646, $650-656, 665-667, 674, 676, 678, 693, $\underline{701}, 747, 829, 849, 859} \\ \mbox{MPI_Op, $220, $\underline{229}, 232, 234, 236-239, 241, 243, $459} \\ \end{array}$
32 33 34 35 36 37 38 39 40 41 42 43 44	$\begin{split} & TYPE(MPI\_File), 852, 860 \\ & MPI\_Fint, \underline{829}, 857, 859, 1045 \\ & MPI\_Group, \underline{310}, 311-316, 318, 351, 496, 557, \\ & 588, 589, 639, 830, 852, 859, 860 \\ & TYPE(MPI\_Group), 852, 860 \\ & MPI\_Info, 447, \underline{471}, 472-477, 494, 498, 513, \\ & 516, 518, 525, 529-531, 538, 558, 633, \\ & 636, 641, 642, 644, 772, 830, 851, 852, \\ & 859, 860, 1048 \\ & TYPE(MPI\_Info), 851, 852, 860 \\ & MPI\_Message, \underline{83}, 830, 847, 852, 859, 860, 1039 \\ & TYPE(MPI\_Message), 847, 852, 859, 860, 1039 \\ & TYPE(MPI\_Message), 847, 852, 860 \\ & MPI\_Offset, \underline{20}, 20, 21, 33, 637-639, 644, 646, \\ & 650-656, 665-667, 674, 676, 678, 693, \\ & \underline{701}, 747, 829, 849, 859 \\ & MPI\_Op, 220, \underline{229}, 232, 234, 236-239, 241, 243, \\ & 244, 264-266, 268-270, 289-291, \\ \end{split}$
32 33 34 35 36 37 38 39 40 41 42 43 44 45	$\begin{split} & TYPE(MPI\_File), 852, 860 \\ & MPI\_Fint, \underline{829}, 857, 859, 1045 \\ & MPI\_Group, \underline{310}, 311-316, 318, 351, 496, 557, \\ & 588, 589, 639, 830, 852, 859, 860 \\ & TYPE(MPI\_Group), 852, 860 \\ & MPI\_Info, 447, \underline{471}, 472-477, 494, 498, 513, \\ & 516, 518, 525, 529-531, 538, 558, 633, \\ & 636, 641, 642, 644, 772, 830, 851, 852, \\ & 859, 860, 1048 \\ & TYPE(MPI\_Info), 851, 852, 860 \\ & MPI\_Message, \underline{83}, 830, 847, 852, 859, 860, 1039 \\ & TYPE(MPI\_Message), 847, 852, 859, 860, 1039 \\ & TYPE(MPI\_Message), 847, 852, 860 \\ & MPI\_Offset, \underline{20}, 20, 21, 33, 637-639, 644, 646, \\ & 650-656, 665-667, 674, 676, 678, 693, \\ & \underline{701}, 747, 829, 849, 859 \\ & MPI\_Op, 220, \underline{229}, 232, 234, 236-239, 241, 243, \\ & 244, 264-266, 268-270, 289-291, \\ & 293-295, 567, 569, 572, 577, 579, 830, \\ \end{split}$
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	$\begin{split} & TYPE(MPI\_File), 852, 860 \\ & MPI\_Fint, \underline{829}, 857, 859, 1045 \\ & MPI\_Group, \underline{310}, 311-316, 318, 351, 496, 557, \\ & 588, 589, 639, 830, 852, 859, 860 \\ & TYPE(MPI\_Group), 852, 860 \\ & MPI\_Info, 447, \underline{471}, 472-477, 494, 498, 513, \\ & 516, 518, 525, 529-531, 538, 558, 633, \\ & 636, 641, 642, 644, 772, 830, 851, 852, \\ & 859, 860, 1048 \\ & TYPE(MPI\_Info), 851, 852, 860 \\ & MPI\_Message, \underline{83}, 830, 847, 852, 859, 860, 1039 \\ & TYPE(MPI\_Message), 847, 852, 859, 860, 1039 \\ & TYPE(MPI\_Message), 847, 852, 860 \\ & MPI\_Offset, \underline{20}, 20, 21, 33, 637-639, 644, 646, \\ & 650-656, 665-667, 674, 676, 678, 693, \\ & \underline{701}, 747, 829, 849, 859 \\ & MPI\_Op, 220, \underline{229}, 232, 234, 236-239, 241, 243, \\ & 244, 264-266, 268-270, 289-291, \\ \end{split}$

TYPE(MPI\_Op), 852, 860 MPI\_Request, 60-65, 67, 68, 69, 72-75, 77, 79, 88-95, 102-106, 461, 622, 625, 653-656, 662-665, 669, 670, 808, 830, 852, 859, 860 TYPE(MPI\_Request), 852, 860 MPI\_Session, 459, 830, 852, 859, 860 TYPE(MPI\_Session), 860 MPI\_Status, 34, 37, 39-41, 67, 68, 72-75, 77, 79-81, 83, 85, 89, 150, 623, 628, 629, 650-653, 658-661, 668, 669, 672, 673, 677, 679-683, 783, 831-834, 856, 859, 860, 1034, 1038, 1041 TYPE(MPI\_Status), 37, 793, 832-834, 856, 860, 1031, 1034, 1041 MPI\_T\_cb\_safety, 859 MPI\_T\_cvar\_handle, 730, 731, 857 MPI\_T\_enum, 724, 725, 726, 736, 750, 857 MPI\_T\_event\_instance, 756 MPI\_T\_event\_registration, 753 MPI\_T\_pvar\_handle, 739, 740-742, 857, 858 MPI\_T\_pvar\_session, 738, 739-742, 857 MPI\_T\_source\_order, 859 MPI\_Win, 367-369, 455, 456, 543, 546, 548, 552, 555, 557, 558, 560, 563, 567, 569, 572-574, 576, 577, 579, 586, 588-590, #-other 592-597, 830, 852, 859, 860 TYPE(MPI\_Win), 852, 860 #-error! #-update5 Are these type declarations really new types? YES + = referenced in a new change-log entry - MPI T cb safety - MPI\_T\_event\_instance - MPI\_T\_event\_registration

- MPI\_T\_source\_order

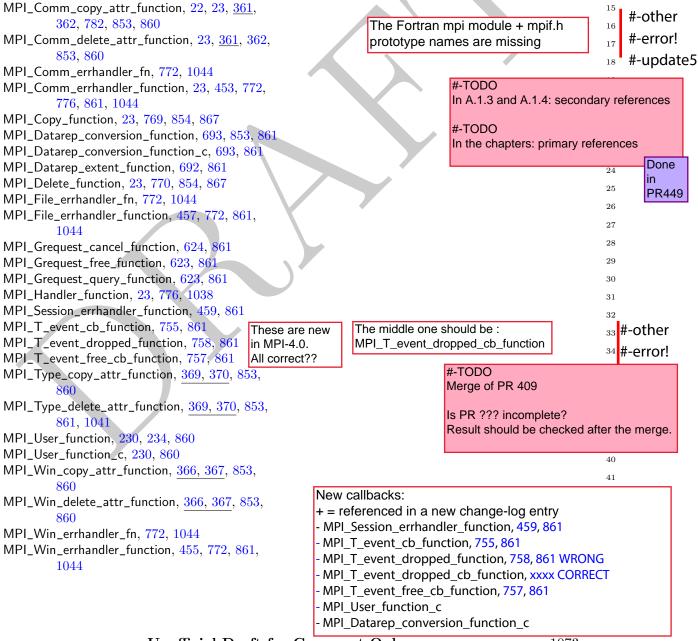
Do they all have correct primary and secondary references in

the Declaration Index?

	Merge of PR422	PR422	
		is now	
	Is PR422 incomplete?	meraed	
	Result should be checked	after the merg	e.
nent Only	1072		

# 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.



Unofficial Draft for Comment Only

# MPI Function Index

Wrong macro is used. Caused Function instead of Callback Index!

 $\mathbf{5}$ 

10

See page 23

<u>rne</u> underlined page numbers refer to the function definitions.

#-othe	er	<sup>11</sup> COMM_COPY_ATTR_FUNCTION, 23
#-erro	rl	<sup>12</sup> COMM_DELETE_ATTR_FUNCTION, 23
		13 COPY_FUNCTION, 23
Done		14
in	1	DELETE_FUNCTION, 23
PR438		<sup>16</sup> MPI_ABORT, 231, 451, 481, 487, <u>505</u> , 537,
		723, 829, 1047
		<sup>125</sup> , 625, 1047 <sup>18</sup> MPI_ACCUMULATE, 541, 559, <u>567</u> , 568, 572,
		$\begin{array}{c} & & 11112, 110000 \text{ min}(21,000,012,000,000$
		MPLADD ERROR CLASS 465
		MPL ADD EBROR CODE, 466
		MPI_ADD_ERROR_STRING, 466
		<sup>22</sup> MPI_ADDRESS, 23, 775, 791, 1038
		<sup>23</sup> MPI_AINT_ADD, 24, 137, <u>138</u> , 553, 1036
		<sup>24</sup> MPI_AINT_DIFF, 24, 137, 138, <u>139</u> , 553, 1036
		<sup>25</sup> MPI_ALLGATHER, 183, 187, 188, <u>209</u> , 210,
		26 212, 214, 257
		<sub>27</sub> MPI_ALLGATHER_INIT, <u>280</u>
		<sup>28</sup> MPI_ALLGATHERV, 183, 187, 188, <u>211</u> , 212,
		29 258
		<sup>30</sup> MPI_ALLGATHERV_INIT, <u>282</u>
		$MP1_ALLOC_MEM, \underline{447}, 448-450, 403, 545,$
		547, 545-551, 554, 502, 555, 752-754,
		MDI ALLDEDUCE 199 196 199 999 990
		234 235 266 1045
		<sup>35</sup> MPI_ALLREDUCE_INIT, <u>290</u>
		<sup>36</sup> MPI_ALLTOALL, 183, 187, 188, <u>213</u> , 214,
		$^{37}$ 216, 217, 259, 1043
		<sup>38</sup> MPI_ALLTOALL_INIT, <u>283</u>
		<sup>39</sup> MPI_ALLTOALLV, 183, 187, 188, <u>215</u> , 216,
		40 217, 219, 261, 1043
		41 MPI_ALLTOALLV_INIT, <u>285</u>
		42 MPI_ALLTOALLW, 183, 187, 188, <u>217</u> , 218,
		$_{43}$ 219, 263, 1043
		MPI_ALLTOALLW_INIT, <u>287</u>
		MPI_AT <sup>*</sup> TR_DELE <sup>*</sup> TE, 23, 373, 770, <u>771</u>
		MP1_A11R_GE1, 23, 373, $\frac{771}{1}$ , 837
		<sup>46</sup> MPI_ATTR_PUT, 23, 373, <u>771</u> , 837, 840, 841
		<sup>47</sup> MPI_BARRIER, 183, 187, <u>189</u> , 248, 503,
		48 605-607, 703

MPI\_BARRIER\_INIT, 272 MPI\_BCAST, 14, 183, 187, 190, 191, 221, 249, 299, 300 MPI\_BCAST\_INIT, 14, 273 MPI\_BSEND, 14, 47, 48, 58 MPI\_BSEND\_INIT, 91, 95 MPI\_BUFFER\_ATTACH, 55, 68 MPI\_BUFFER\_DETACH, 56, 1032, 1041 MPI\_CANCEL, 23, 51, 68, 80, 88, 89, 90, 247, 272, 488, 574, 621, 624, 625, 772, 1032 MPI\_CART\_COORDS, 385, 400, 401, 1047 MPI\_CART\_CREATE, 349, 385, 386, 387-389, 408, 410, 807, 1035, 1046 MPI\_CART\_GET, 385, 399, 1046 MPI\_CART\_MAP, 385, 408, 1040 MPI\_CART\_RANK, 385, 400, 1047 MPI\_CART\_SHIFT, 385, 405, 406, 410, 1047 MPI\_CART\_SUB, 385, 407, 409, 1047 MPI\_CARTDIM\_GET, 385, 398, 399, 1046 MPI\_CLOSE\_PORT, 526, 530 MPI\_COMM\_ACCEPT, 525, 527, 528, 535, 536 MPI\_COMM\_C2F, 830 MPI\_COMM\_CALL\_ERRHANDLER, 467, 469 MPI\_COMM\_COMPARE, 321, 351 MPI\_COMM\_CONNECT, 464, 528, 535, 536 MPI\_COMM\_CREATE, 319, 321, 325, 326-330, 385, 1044 MPI\_COMM\_CREATE\_ERRHANDLER, 23, 452, 453, 454, 775, 863, 866, 1041 MPI\_COMM\_CREATE\_FROM\_GROUP, 321, 336, 338, 355, 445 MPI\_COMM\_CREATE\_GROUP, 321, 328, 329, 330, 338, 1039 MPI\_COMM\_CREATE\_KEYVAL, 23, 359, <u>360</u>, 362, 363, 373, 769, 836, 837, 862, 865, 1041, 1046 MPI\_COMM\_DELETE\_ATTR, 23, 359, 362-364, 365, 373, 771 MPI\_COMM\_DISCONNECT, 373, 494, 503, 518, 537, 538

MPI\_COMM\_DUP, 312, 319, 321, 322, 323, 324, 326, 352, 355, 359, 362, 366, 373, 381, 769, 777, 1033, 1039 MPI\_COMM\_DUP\_FN, 23, 362, 363, 787, 853, 1036, 1041 MPI\_COMM\_DUP\_WITH\_INFO, 321, 323, 324, 339, 777, 1039 MPI\_COMM\_F2C, <u>830</u> MPI\_COMM\_FREE, 319, 322, 338, 352, 355, 362-364, 366, 373, 487, 490, 494, 503, 518, 537, 538, 770 MPI\_COMM\_FREE\_KEYVAL, 23, 359, 363, 373, 770 MPI\_COMM\_GET\_ATTR, 23, 359, 364, 373, 445, 771, 788, 837, 840 MPI\_COMM\_GET\_ERRHANDLER, 23, 452, 454, 775, 1047 MPI\_COMM\_GET\_INFO, 339, 341, 1033, 1039 MPI\_COMM\_GET\_NAME, <u>376</u>, 377, 378, 1046 MPI\_COMM\_GET\_PARENT, 377, 481, 482, 487, 491, 514, <u>517</u>, 518 MPI\_COMM\_GROUP, 17, 310, 312, 313, 319, 320, 351, 452, 1047 MPI\_COMM\_IDUP, 319, 321, 323, 324, 349, 359, 362, 366, 373, 777, 1033, 1034, 1039 MPI\_COMM\_IDUP\_WITH\_INFO, 321, 324, 339, 777, 1033 MPI\_COMM\_JOIN, 491, 538, 539 MPI\_COMM\_NULL\_COPY\_FN, 23, <u>362</u>, 363, 782, 836, 853, 1036, 1041 MPI\_COMM\_NULL\_DELETE\_FN, 23, 362, 363, 853, 1036 MPI\_COMM\_RANK, 14, 320, 351, 789 MPI\_COMM\_RANK\_F08, 789 MPI\_COMM\_REMOTE\_GROUP, 352 MPI\_COMM\_REMOTE\_SIZE, 351, 352 MPI\_COMM\_SET\_ATTR, 23, 359, 362, 364, 373, 770, 788, 837, 841 MPI\_COMM\_SET\_ERRHANDLER, 23, 452, 454, 775 MPI\_COMM\_SET\_INFO, 272, 339, 340, 341, 1033, 1035, 1039 MPI\_COMM\_SET\_NAME, <u>375</u>, 376 MPI\_COMM\_SIZE, <u>319</u>, 320, 351, 614 MPI\_COMM\_SPAWN, 481, 482, 506, 507, 512, <u>513</u>, 514, 516–522, 535, 536, 1034 MPI\_COMM\_SPAWN\_MULTIPLE, 481, 482, 507, 512, 517, <u>518</u>, 519, 520, 536, 1034 MPI\_COMM\_SPLIT, 321, 325, 326, 329, 330, 331, 381, 385, 387, 389, 407, 409, 1044 MPI\_COMM\_SPLIT\_TYPE, <u>333</u>, 337, 339,

1032,1039	1
MPI_COMM_TEST_INTER, <b>349</b> , <u><b>350</b></u>	2
MPI_COMPARE_AND_SWAP, 541, 559, <u>573</u> ,	3
614	his is a callback
$\frac{1}{10000}, \frac{1}{10000}, 1$	rototype
1030	6
MPI_DATAREP_CONVERSION_FUNCTION	, <sup>7</sup> #-error!
693, 827	<sub>8</sub> #-other
MPI_DIMS_CREATE, 385, <u>387</u> , 388, 1033	Dente
MPI_DIST_GRAPH_CREATE, 339, 385, 390,	9 Done in
$\underline{393}, 394 - 396, 405, 410, 1044$	<sup>10</sup> PR449
MPI_DIST_GRAPH_CREATE_ADJACENT,	11
$339, 385, 390, \underline{391}, 392, 396, 405, 410,$	12
1040, 1044	13
MPI_DIST_GRAPH_NEIGHBORS, 385, 403,	14
404, 405, 410, 1040, 1044	15
MPI_DIST_GRAPH_NEIGHBORS_COUNT,	
$385, \underline{403}, 405, 1037, 1044$	16
MPI_DUP_FN, 23, 362, <u>770</u> , 854	17
MPI_ERRHANDLER_C2F, 504, <u>830</u>	18
MPI_ERRHANDLER_CREATE, 23, 775,	19
1038, 1041	20
MPI_ERRHANDLER_F2C, 504, <u>830</u>	21
MPI_ERRHANDLER_FREE, 452, <u>460</u> , 487,	22
494,504,1047	
MPI_ERRHANDLER_GET, 23, 775, 1038,	23
1047	24
MPI_ERRHANDLER_SET, 23, 775, 1038	25
MPI_ERROR_CLASS, <u>462</u> , 465, 504, 766	26
MPI_ERROR_STRING, <u>461</u> , 462, 465, 467,	27
504 Must be MPI_T_EVENT	··· <sub>28</sub> #-error!
MPI_EVENT_CALLBACK_GET_INFO, 756	<sup>29</sup> #-update9
MPI_EVENT_HANDLE_GET_INFO, 754	30 Done
MPI_EXSCAN, 184, 187, 222, 230, <u>243</u> , 271,	<sub>31</sub> in
1043 MDI EVICCAN INTE 205	PR449
MPI_EXSCAN_INIT, <u>295</u>	32
MPI_F_SYNC_REG, 137, 616, 780, <u>796</u> , 797,	33
816, 818, 819, 821, 1042 MDI FETCH AND OD 541 550 568 572	34
MPI_FETCH_AND_OP, 541, 559, 568, <u>572</u> MPI_FILE_C2F, <u>830</u>	35
MPI_FILE_C2F, <u>350</u> MPI_FILE_CALL_ERRHANDLER, <u>468</u> , <u>469</u>	36
MPI_FILE_CALL_EARMANDLER, $409$ MPI_FILE_CLOSE, 538, 633, 634, $636$	37
	38
MPI_FILE_CREATE_ERRHANDLER, 452,	39
<u>457</u> , 458, 863, 866, 1041 MPI_FILE_DELETE, 635, 636, 637, 641, 643,	40
706, 707	
MPI_FILE_F2C, <u>830</u>	41
MPI_FILE_GET_AMODE, <u>640</u>	42
MPI_FILE_GET_ATOMICITY, <u>698</u>	43
MPI_FILE_GET_BYTE_OFFSET, 658, 667,	44
675	45
MPI_FILE_GET_ERRHANDLER, 452, 458,	46
707, 1047	47
MPI_FILE_GET_GROUP, <u>639</u>	48
$\frac{1}{1000}$	4U

		1	MPI_FILE_GET_INFO	$D,  641,  \underline{642},  643,  1033,$
		2	1048	
		3	MPI_FILE_GET_POS	ITION, <u>666</u> , 667
		4		ITION_SHARED, $\underline{674}$ ,
		5	675,699	
		6	MPI_FILE_GET_SIZE	
		7		$E\_EXTENT, 686, \underline{687},$
		8	695, 1032	
			MPI_FILE_GET_VIE	$W, \underline{646}, 647$
		9	MPI_FILE_IREAD, 64	$17, \frac{662}{67}, 675, 696$
		10	MPI_FILE_IREAD_A	LL, $047$ , $003$ , $004$ , $1030$
		11	MPI_FILE_IREAD_A	
		12	MPI_FILE_IREAD_SH	
		13	MPI_FILE_IWRITE, (	
		14	MPI_FILE_IWRITE_A	
		15	MPI_FILE_IWRITE_A	
		16		AT_ALL, 647, <u>657</u> , 1036
		17	MPI_FILE_IWRITE_S	
		18	MPI_FILE_OPEN, 463	
		19		667, 701, 702, 706, 707
		20	MPI_FILE_PREALLO	
			701, 702	
		21	MPI_FILE_READ, 64	$7,  648,  \underline{658},  659,  660,$
		22	662,701,702	
		23	MPI_FILE_READ_AL	L, 647, <u>659</u> , 660, 664,
		24	676	
		25	MPI_FILE_READ_AL	
		26	676, <u>679</u> , 696,	
		27	MPI_FILE_READ_AL	L_END, 647, 675, 676,
		28	<u>680, 696, 821</u>	I EN 14
		29	MPI_FILE_READ_AL	5.2FN, 14 5.647, 650, 651, 652, 654
		30	MPI_FILE_READ_AT	, 047, <u>050</u> , 051, 052, 054 1 ALL 647 651 652
		31	655	$\_$ ALL, 047, <u>051</u> , 052,
		32	MPI_FILE_READ_AT	ALL BEGIN 647
		33	<u>677, 821</u>	
			MPI_FILE_READ_AT	ALL END. 647. 677.
		34	821	, <u></u> ,
		35	MPI_FILE_READ_AT	_ALL_FN, 14
		36		ADERED, 647, 672, 673
		37	MPI_FILE_READ_OF	RDERED_BEGIN, 647,
		38	<u>681</u> , 821	
		39	MPI_FILE_READ_OF	$RDERED_END, 647,$
		40	$\underline{682}, 821$	
		41	MPI_FILE_READ_OR	<i>,</i>
		42	MPI_FILE_READ_SH	ARED, 647, <u>668</u> , 669,
		43	670, 673	
		44	MPI_FILE_SEEK, <u>666</u>	
		45	MPI_FILE_SEEK_SH	
#-otl	her	46		MICITY, 635, <u>697</u> , 698
#-eri	ror!	47	MPI_FILE_SET_ERR	HANDLER, $452, \frac{457}{2}$ ,
ne			707	
			_READ_ALL_FN	
R449			_READ_AT_ALL_FN _READ_ORDERED_FN	all these must be
			_READ_ORDERED_FN _WRITE_ALL_FN	changed into _BEGIN instead of _FN
			_WRITE_ALL_FN	on page 14, line 37

Done

MPI\_FILE\_WRITE\_ORDERED\_FN

in PR449 1076

MPI\_FILE\_SET\_INFO, <u>641</u>, 642, 643, 1033, 1048 MPI\_FILE\_SET\_SIZE, <u>637</u>, 638, 697, 699, 701, 702 MPI\_FILE\_SET\_VIEW, 131, 464, 634, 641-643, <u>644</u>, 645, 646, 667, 675, 685, 692, 701, 707, 1048 MPI\_FILE\_SYNC, 636, 648, 696, 697, 698, 699, 704 MPI\_FILE\_WRITE, 647, 648, 660, 661, 662, 665.701 MPI\_FILE\_WRITE\_ALL, 647, 661, 662, 665 MPI\_FILE\_WRITE\_ALL\_BEGIN, 647, 680, 808, 821 MPI\_FILE\_WRITE\_ALL\_END, 647, 681, 821 MPI\_FILE\_WRITE\_ALL\_FN, 14 MPI\_FILE\_WRITE\_AT, 647, 648, 652, 653, 656 MPI\_FILE\_WRITE\_AT\_ALL, 647, 653, 657 MPI\_FILE\_WRITE\_AT\_ALL\_BEGIN, 647, 678, 821 MPI\_FILE\_WRITE\_AT\_ALL\_END, 647, 679, 821 MPI\_FILE\_WRITE\_AT\_ALL\_FN, 14 MPI\_FILE\_WRITE\_ORDERED, 647, 672, 673 MPI\_FILE\_WRITE\_ORDERED\_BEGIN, 647, 683, 821 MPI\_FILE\_WRITE\_ORDERED\_END, 647, 683, 821 MPI\_FILE\_WRITE\_ORDERED\_FN, 14 MPI\_FILE\_WRITE\_SHARED, 647, 648, 669, 671 - 673MPI\_FINALIZE, 18, 25, 26, 445, 451, 479, 486, 487-491, 509, 537, 538, 634, 720, 730, 743, 745, 829, 832, 833, 1034, 1040, 1047 MPI\_FINALIZED, 489, 490, 504, 829, 1036 MPI\_FREE\_MEM, 449, 463, 547, 549 MPI\_GATHER, 183, 186, 187, 191, 195, 196, 203, 210, 221, 251 MPI\_GATHER\_INIT, 274 MPI\_GATHERV, 183, 187, 194, 195-197, 205, 212, 252 MPI\_GATHERV\_INIT, 276 MPI\_GET, 541, 559, <u>563</u>, 564, 572, 577, 583, 605, 608, 616, 818, 1048 MPI\_GET\_ACCUMULATE, 541, 559, 568, 570, 571, 572, 581, 609, 614, 615, 1035 MPI\_GET\_ADDRESS, 23, 117, 137, 138, 139, 152, 553, 775, 791, 810, 815, 834-836 MPI\_GET\_COUNT, 37, 38, 39, 67, 151, 574, 629, 650, 1038, 1045 MPI\_GET\_ELEMENTS, 67, 150, 151, 152,

#### r Comment Only

629, 630, 650, 1038 MPI\_GET\_ELEMENTS\_X, 67, <u>150</u>, 151, 152, 629, 650, 1038 MPI\_GET\_LIBRARY\_VERSION, 444, 504, 1035, 1036, 1038 MPI\_GET\_PROCESSOR\_NAME, 446, 447, 1047 MPI\_GET\_VERSION, <u>443</u>, 444, 504, 795, 1036 MPI\_GRAPH\_CREATE, 385, 388, 390, 395, 398, 402, 409, 410, 1046 MPI\_GRAPH\_GET, 385, 398 MPI\_GRAPH\_MAP, 385, 409 MPI\_GRAPH\_NEIGHBORS, 385, 401, 402, 410, 1044 MPI\_GRAPH\_NEIGHBORS\_COUNT, 385, 401, 402, 1044 MPI\_GRAPHDIMS\_GET, 385, <u>397</u>, 398 MPI\_GREQUEST\_COMPLETE, 622-624, 625 MPI\_GREQUEST\_START, <u>622</u>, 864, 866, 1044 MPI\_GROUP\_C2F, 830 MPI\_GROUP\_COMPARE, 311, 315 MPI\_GROUP\_DIFFERENCE, 314 MPI\_GROUP\_EXCL, <u>315</u>, 317 MPI\_GROUP\_F2C, 830 MPI\_GROUP\_FREE, 318, 319, 320, 452, 487, 494.1047 MPI\_GROUP\_FROM\_SESSION\_PSET, 318 MPI\_GROUP\_INCL, <u>314</u>, <u>315</u>, <u>316</u> MPI\_GROUP\_INTERSECTION, 313 MPI\_GROUP\_RANGE\_EXCL, <u>317</u> MPI\_GROUP\_RANGE\_INCL, 316 MPI\_GROUP\_RANK, <u>310</u>, 320 MPI\_GROUP\_SIZE, <u>310</u>, 319 MPI\_GROUP\_TRANSLATE\_RANKS, 311, 1046 MPI\_GROUP\_UNION, 313 MPI\_IALLGATHER, 183, 187, 188, 256 MPI\_IALLGATHERV, 183, 187, 188, 257 MPI\_IALLREDUCE, 183, 187, 188, 265 MPI\_IALLTOALL, 183, 187, 188, 258 MPI\_IALLTOALLV, 183, 187, 188, 260 MPI\_IALLTOALLW, 183, 187, 188, 262 MPI\_IBARRIER, 183, 187, 246, 248, 300 MPI\_IBCAST, 14, 183, 187, 248, 249, 303 MPI\_IBSEND, <u>61</u>, 68, 95 MPI\_IEXSCAN, 184, 187, 270 MPI\_IGATHER, 183, 187, 250 MPI\_IGATHERV, 183, 187, 251 MPI\_IMPROBE, 14, 80, 83, 84, 85, 88, 510, 1034, 1039 MPI\_IMRECV, 83-85, 87, 88, 1039

MPI_INEIGHBOR_ALLGATHER, <u>386</u> , <u>422</u> ,	1
1040	<b>2</b>
MPI_INEIGHBOR_ALLGATHERV, <u>386</u> , <u>424</u> ,	3
1040	4
MPI_INEIGHBOR_ALLTOALL, <u>386</u> , <u>425</u> ,	5
1040	
MPI_INEIGHBOR_ALLTOALLV, <u>386</u> , <u>426</u> ,	6
1040	7
MPI_INEIGHBOR_ALLTOALLW, <u>386</u> , <u>428</u> ,	8
1040	9
MPI_INFO_C2F, 504, <u>830</u>	10
$MPI_INFO_CREATE, 472, 504$	11
MPI_INFO_CREATE_ENV, 477, 504, 1032	
MPI_INFO_DELETE, 463, <u>473</u> , 476, 504	12
MPI_INFO_DUP, <u>477</u> , 504	13
$MPI_INFO_F2C, 504, 830$	14
MPI_INFO_FREE, 341, <u>477</u> , 478, 487, 494,	15
$\begin{array}{c} \text{MF1_INFO_FREE, 541, \underline{411}, 410, 407, 494, \\ 497, 504, 559, 642, 748, 752, 754, 757 \end{array}$	16
MPI_INFO_GET, 23, 471, <u>473</u> , 504, <u>772</u> , 1034,	17
1048	18
MPI_INFO_GET_NKEYS, 471, <u>476</u> , 504, 1048	19
MPI_INFO_GET_NTHKEY, 471, <u>476</u> , 504,	20
1048	21
MPI_INFO_GET_STRING, 23, 471, <u>475</u> , 772,	22
773,1034	
MPI_INFO_GET_VALUELEN, 23, 471, <u>474</u> ,	23
$504, \underline{773}, 1034, 1048$	24
MPI_INFO_SET, <u>472</u> , 473–476, 504	25
MPI_INIT, 18, 25, 26, 309, 445, 451, 478, 479,	26
$\underline{480}, 481, 483 - 486, 489, 490, 504, 510,$	27
514-517, 534, 535, 720, 722, 730, 743,	28
829, 832, 833, 1034, 1038, 1040, 1042,	
1044	29
MPI_INIT_THREAD, 25, 309, 451, 478–481,	30
483, 484–486, 489, 490, 493, 510, 535,	31
$\overline{720}$ , 722, 723, 730, 829, 1038, 1040,	32
1044	33
MPI_INITIALIZED, <u>489</u> , 490, 504, 829, 1036	34
MPI_INTERCOMM_CREATE, 322, 328, 329,	
$352, \underline{353}, 354, 355, 1039$	35
MPI_INTERCOMM_CREATE_FROM_GROUPS,	36
321, 322, 352, <u>354</u> , 445	37
MPI_INTERCOMM_MERGE, 321, 328, 349,	38
352, 353, <u>355</u> , 1041	39
MPI_IPROBE, 14, 38, <u>80</u> , 81, 83–85, 88, 510,	40
1039	
MPI_IRECV, 14, <u>64</u> , 87, 809, 810, 812, 814	41
	42
MPI_IREDUCE, 183, 187, 188, <u>264</u> , 265	43
MPI_IREDUCE_SCATTER, 183, 187, 188,	44
$\frac{268}{2000}$	45
MPI_IREDUCE_SCATTER_BLOCK, 183,	46
187, 188, <u>266</u>	
MPI_IRSEND, <u>63</u>	47
MPI_IS_THREAD_MAIN, 483, <u>485</u> , 1036	48

	1	MPI_ISCAN, 184, 187, <u>269</u>
	2	MPI_ISCATTER, 183, 187, <u>253</u>
	3	MPI_ISCATTERV, 183, 187, <u>254</u>
	4	MPI_ISEND, 14, <u>60</u> , 95, 787, 788, 791,
	5	807 - 809, 814
		MPI_ISENDRECV, <u>65</u> , 1034
	6	MPI_ISENDRECV_REPLACE, <u>66</u> , 1034
	7	MPI_ISSEND, $\underline{62}$
	8	MPI_KEYVAL_CREATE, 23, <u>769</u> , 771, 867
	9	MPI_KEYVAL_FREE, 23, 373, 770
	10	MPI_LOOKUP_NAME, 463, 525, 530, 531
	11	MPI_MESSAGE_C2F, <u>830</u> , 1039
	12	MPI_MESSAGE_F2C, $\overline{\underline{830}}$ , 1039
		MPI_MPROBE, 14, 80, 83, 84, 85, 88, 510,
	13	1039
	14	MPI_MRECV, 14, 83–85, <u>86</u> , 87, 1039
	15	MPI_NEIGHBOR_ALLGATHER, 386, 411,
	16	413, 415, 423, 1032, 1040
	17	MPI_NEIGHBOR_ALLGATHER_INIT, 430
	18	MPI_NEIGHBOR_ALLGATHERV, 386, 413,
	19	425, 1032, 1040
		MPI_NEIGHBOR_ALLGATHERV_INIT, 431
	20	MPI_NEIGHBOR_ALLTOALL, 386, 415, 417,
	21	418, 426, 1032, 1040
	22	MPI_NEIGHBOR_ALLTOALL_INIT, <u>433</u>
	23	MPI_NEIGHBOR_ALLTOALLV, 386, 418,
#-other	24	427, 1032, 1040
	25	MPI_NEIGHBOR_ALLTOALLV_INIT, <u>434</u>
#-error!	26	MPI_NEIGHBOR_ALLTOALLW, 386, 420,
	27	429, 1032, 1040
		MPI_NEIGHBOR_ALLTOALLW_INIT, 436
	28	MPI_NULL_COPY_FN, 22, 23, 362, 770, 854
	29	MPI_NULL_DELETE_FN, 23, 363, 770, 854
	30	MPI_OP_C2F, 830
	31	MPI_OP_COMMUTATIVE, <u>237</u> , 1043
	32	MPI_OP_CREATE, <u>229</u> , 230, 232, 787, 862,
	33	865, 1041
	34	MPI_Op_create_c, 828
		MPI_OP_F2C, 830
	35	MPI_OP_FREE, <u>232</u> , 487, 494
	36	MPI_OPEN_PORT, <u>525</u> , <u>527–532</u>
	37	MPI_PACK, 58, <u>171</u> , <u>173</u> , <u>175</u> , <u>178</u> , <u>688</u> , <u>694</u>
	38	MPI_PACK_EXTERNAL, 8, <u>178</u> , 801, 1045
	39	MPI_PACK_EXTERNAL_SIZE, 180
	40	MPI_PACK_SIZE, 58, <u>174</u> , 175, 1039
	41	MPI_PARRIVED, 100, 106, <u>107</u>
	42	MPI_PCONTROL, 714, 715, 716
		MPI_PREADY, 100, 102, 104, 105–107
	43	MPI_PREADY_LIST, 106
	44	MPI_PREADY_RANGE, 105, 106
	45	MPI_PRECV_INIT, 100, <u>103</u> , 104, 108
	46	MPI_PROBE, 14, 36, 38, 39, 80, <u>81</u> , 82, 83, 85,
	47	87, 88, 510, 1039
	48	MPI_PSEND_INIT, 100, <u>102</u> , 103–105, 108
	10	

MPI\_PUBLISH\_NAME, 525, 529, 530, 531 MPI\_PUT, 541, 559, 560, 564, 568, 576, 583, 589, 599, 601, 606, 607, 616, 808, 818, 1048 MPI\_QUERY\_THREAD, 485, 486, 510, 1036 MPI\_RACCUMULATE, 541, 559, 568, 572, 578, 579 MPI\_RECV, 14, 30, <u>35</u>, 37-39, 80, 83, 84, 87, 116, 149, 173, 184, 192, 301, 630, 703, 743, 815, 818, 819 MPI\_RECV\_INIT, 14, 94, 95, 104 MPI\_REDUCE, 183, 187, 188, 220, 221, 222, 230-232, 235, 239, 241, 242, 244, 265, 568, 571-573, 1044 MPI\_REDUCE\_INIT, 289 MPI\_REDUCE\_LOCAL, 221, 222, 230, 236, 1041.1043 MPI\_REDUCE\_SCATTER, 183, 187, 188, 222, 230, 239, <u>240</u>, 241, 269 MPI\_REDUCE\_SCATTER\_BLOCK, 183, 187, 188, 222, 230, 238, 239, 267, 1043 MPI\_REDUCE\_SCATTER\_BLOCK\_INIT, 291 MPI\_REDUCE\_SCATTER\_INIT, 293 MPI\_REGISTER\_DATAREP, 463, <u>691</u>, 692, 693, 695, 707, 864, 867 MPI\_Register\_datarep\_c, 828 MPI\_Op\_create\_c MPI\_REQUEST\_C2F, 830 MPI\_Register\_datarep\_c shall also have primiry indexes here MPI\_REQUEST\_F2C, 830 MPI\_REQUEST\_FREE, <u>69</u>, 70, 88, 96, 247, 271, 272, 487, 494, 574, 624, 625, 1043 MPI\_REQUEST\_GET\_STATUS, 39, 79, 623, 1043 MPI\_RGET, 541, 559, 576, 577 MPI\_RGET\_ACCUMULATE, 541, 559, 568, 571, 580, 581 MPI\_RPUT, 541, 559, 575, 576 MPI\_RSEND, 14, 50 MPI\_RSEND\_INIT, 93 MPI\_SCAN, 184, 187, 222, 230, 241, 242, 244, 270MPI\_SCAN\_INIT, 294 MPI\_SCATTER, 183, 187, 202, 203, 205, 206, 239, 254 MPI\_SCATTER\_INIT, 277 MPI\_SCATTERV, 183, 187, 204, 205, 206, 241, 255 MPI\_SCATTERV\_INIT, 279 MPI\_SEND, 14, 29, <u>30</u>, 31, 39, 44, 116, 148, 172, 301, 634, 703, 716, 815, 816, 818 MPI\_SEND\_INIT, 14, 90, 95, 103 MPI\_SENDRECV, 40, 405 MPI\_SENDRECV\_REPLACE, 41 MPI\_SESSION\_C2F, 830

MPI\_SESSION\_CALL\_ERRHANDLER, 468, 469, 504 MPI\_SESSION\_CREATE\_ERRHANDLER, 452, 458, 459, 493, 504, 863, 866 MPI\_SESSION\_CREATE\_KEYVAL, 865 MPI\_SESSION\_F2C,  $\underline{830}$ MPI\_SESSION\_FINALIZE, 493, 494, 503 MPI\_SESSION\_GET\_ERRHANDLER, 452, 460 MPI\_SESSION\_GET\_INFO, 493, 497 MPI\_SESSION\_GET\_NTH\_PSET, 318, 496, 497MPI\_SESSION\_GET\_NUM\_PSETS, 495, 496 MPI\_SESSION\_GET\_PSET\_INFO, 498 MPI\_SESSION\_INIT, 452, 487, 492, 493, 504, 510, 535, 723 MPI\_SESSION\_SET\_ERRHANDLER, 452, 459MPI\_SIZEOF, 23, 773, 774, 1031 MPI\_SSEND, 49 MPI\_SSEND\_INIT, 92 MPI\_START, 95, 96, 99, 102, 105, 106, 271, 272, 815 MPI\_STARTALL, 95, 96, 99, 102, 105, 106, 271, 272, 815 MPI\_STATUS\_C2F, 832 MPI\_STATUS\_C2F08, 832, 1041 MPI\_STATUS\_F082C, 832, 1041 MPI\_STATUS\_F082F, 834, 1031, 1041 MPI\_STATUS\_F2C, 831 MPI\_STATUS\_F2F08, 833, 1031, 1041 MPI\_STATUS\_SET\_CANCELLED, 629, 1033 MPI\_STATUS\_SET\_ELEMENTS, 628, 629 MPI\_STATUS\_SET\_ELEMENTS\_X, <u>629</u>, 1038 MPI\_T\_CATEGORY\_CHANGED, 766 MPI\_T\_CATEGORY\_GET\_CATEGORIES, 765, 766, 768 MPI\_T\_CATEGORY\_GET\_CVARS, 764, 766, 768MPI\_T\_CATEGORY\_GET\_EVENTS, 765 MPI\_T\_CATEGORY\_GET\_INDEX, 764, 768, 1036 MPI\_T\_CATEGORY\_GET\_INFO, 763, 766, 768, 1035, 1036 MPI\_T\_CATEGORY\_GET\_NUM, 762 MPI\_T\_CATEGORY\_GET\_NUM\_EVENTS, <u>763</u>, 764 MPI\_T\_CATEGORY\_GET\_PVARS, 765, 766, 768MPI\_T\_CVAR\_GET\_INDEX, 728, 729, 768, 1036 MPI\_T\_CVAR\_GET\_INFO, 724, 726, 727, 728, 730-732, 766, 768, 1035, 1036

MPI_T_CVAR_GET_NUM, <u>726</u> , 731	1
MPI_T_CVAR_HANDLE_ALLOC, 724, 730,	$^{2}$
731, 732, 768	3
MPI_T_CVAR_HANDLE_FREE, <u>731</u> , 768	4
MPI_T_CVAR_READ, <u>731</u> , 768	
MPI_T_CVAR_WRITE, <u>731</u> , 768	5
MPI_T_ENUM_GET_INFO, 724, 725, 768	6
MPI_T_ENUM_GET_ITEM, <u>725</u> , 768	7
MPI_T_EVENT_CALLBACK_GET_INFO,	8
756	9
MPI_T_EVENT_CALLBACK_SET_INFO,	10
$\frac{756}{756}$	
MPI_T_EVENT_COPY, 749, 752, 760	11
MPI_T_EVENT_GET_INDEX, <u>752</u> , 753, 768	12
	13
MPI_T_EVENT_GET_INFO, 724, 750, <u>751</u> ,	14
753, 760, 768	15
MPI_T_EVENT_GET_NUM, 750	16
MPI_T_EVENT_GET_SOURCE, 749, 761	
MPI_T_EVENT_GET_TIMESTAMP, 747,	17
749, <u>760</u>	18
MPI_T_EVENT_HANDLE_ALLOC, <u>753</u> , 756,	19
768	20
MPI_T_EVENT_HANDLE_FREE, <u>757</u> , 768	21
MPI_T_EVENT_HANDLE_GET_INFO, <u>754</u>	
MPI_T_EVENT_HANDLE_SET_INFO, <u>754</u>	22
MPI_T_EVENT_READ, 749, 751, <u>759</u>	23
MPI_T_EVENT_REGISTER_CALLBACK,	24
755	25
MPI_T_EVENT_SET_DROPPED_HANDLER,	26
<u>758,</u> 759	27
MPI_T_FINALIZE, <u>723</u>	
MPI_T_INIT_THREAD, <u>722</u> , 723	28
MPI_T_PVAR_GET_INDEX, <u>737</u> , 768, 1036	29
MPI_T_PVAR_GET_INFO, 724, 735, 736,	30
739, 741–743, 766, 768, 1035, 1036	31
MPI_T_PVAR_GET_NUM, <u>735</u> , 739	32
$MPI_T_PVAR_HANDLE\_ALLOC, 724, 739,$	33
741 749 769	
/41, /42, /68 MPI_T_PVAR_HANDLE_FREE, 740, 768,	34
	35
1036 MDL T. DVAD. DEAD. 741, 749, 740, 769	36
MPI_T_PVAR_READ, <u>741</u> , 742, 749, 768,	37
	38
MPI_T_PVAR_READRESET, 737, <u>742</u> , 749,	
768,1036	39
MPI_T_PVAR_RESET, <u>742</u> , 749, 768, 1035,	40
1036	41
MPI_T_PVAR_SESSION_CREATE, <u>738</u> , 768	42
MPI_T_PVAR_SESSION_FREE, <u>738</u> , 768	43
MPI_T_PVAR_START, <u>740</u> , 749, 768, 1035,	
1036	44
MPI_T_PVAR_STOP, <u>740</u> , 749, 768, 1035,	45
	46
MPI_T_PVAR_WRITE, <u>741</u> , 742, 749, 768,	47
1036	48

#-err

#-oth

Done

PR44

lin

<ul> <li>MPI_T_SOURCE_GET_INFO, 747, 761, 768</li> <li>MPI_T_SOURCE_GET_INESTAMP, 748, 749, 768</li> <li>MPI_TEST, 39, 67, 68, 69, 70, 72, 74, 79, 88, 96, 99, 100, 106, 107, 272, 487, 625, 648, 649</li> <li>MPI_TEST_CANCELLED, 67-69, 89, 623, 630, 650</li> <li>MPI_TESTALL, 72, 75, 510, 623-625, 628</li> <li>MPI_TESTANY, 72, 72, 74, 77, 510, 623, 624, 628</li> <li>MPI_TESTSOME, 72, 77, 78, 510, 623-625, 628</li> <li>MPI_TYPE_COSMMIT, 146, 147, 831</li> <li>MPI_TYPE_CONTIGUOUS, 15, 117, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CONTIGUOUS, 15, 117, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 153, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 153, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_HVDEXED, 15, 23, 117, 120, 156, 775</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 120, 156, 775</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, 370, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RNDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_RNDEXED_BLOCK, 15, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 373, 1041</li> <li>MPI_TYPE_FREE_L47, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156, 157-159, 827</li> </ul>			
<ul> <li>MPL_T_SOURCE_GET_NUM, 746, 768</li> <li>MPL_T_SOURCE_GET_TIMESTAMP, 748, 749, 768</li> <li>MPL_TEST, 39, 67, 68, 69, 70, 72, 74, 79, 88, 96, 99, 100, 106, 107, 272, 487, 625, 648, 649</li> <li>MPL_TEST_CANCELLED, 67–69, 89, 623, 630, 650</li> <li>MPL_TESTALL, 72, 75, 510, 623–625, 628</li> <li>MPL_TESTANY, 72, 73, 74, 77, 510, 623, 624, 628</li> <li>MPL_TPE_CANCELLED, 72, 77, 78, 510, 623–625, 628</li> <li>MPL_TYPE_CET, 385, 397</li> <li>MPL_TYPE_CONMIT, 146, 147, 831</li> <li>MPL_TYPE_CONMIT, 146, 147, 831</li> <li>MPL_TYPE_CONTIGUOUS, 15, 117, 120, 141, 155, 632, 686</li> <li>MPL_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPL_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPL_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPL_TYPE_CREATE_HINDEXED, 15, 23, 117, 128, 125, 775, 827</li> <li>MPL_TYPE_CREATE_HINDEXED, 15, 23, 117, 128, 155, 775, 827</li> <li>MPL_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 156, 827, 1039</li> <li>MPL_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 125, 125, 125, 125, 126, 155</li> <li>MPL_TYPE_CREATE_INDEXED_BLOCK, 15, 122, 126, 155</li> <li>MPL_TYPE_CREATE_NEVEXD_8, 23, 117, 140, 144, 155, 638, 776, 1041</li> <li>MPL_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 134, 3455</li> <li>MPL_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 455</li> <li>MPL_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPL_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPL_TYPE_PUP_FN, 370, 853, 1036</li> <li>MPL_TYPE_FREE_KEVVAL, 359, 371, 373</li> </ul>		1	MPL T SOURCE GET INFO. 747, 761, 768
<ul> <li>MPI_T_SOURCE_GET_TIMESTAMP, <u>748</u>, 749, 768</li> <li>MPI_TEST, 39, 67, 68, 69, 70, 72, 74, 79, 88, 96, 99, 100, 106, 107, 272, 487, 625, 648, 649</li> <li>MPI_TEST_CANCELLED, 67–69, <u>89</u>, 623, 630, 650</li> <li>MPI_TESTALL, 72, <u>75</u>, 510, 623–625, 628</li> <li>MPI_TESTALL, 72, <u>75</u>, 510, 623–625, 628</li> <li>MPI_TESTSOME, 72, <u>77</u>, 78, 510, 623–625, 628</li> <li>MPI_TYPE_COP_TEST, 385, <u>397</u></li> <li>MPI_TYPE_CONTIGUOUS, 15, <u>117</u>, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, <u>117</u>, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HNDEXED_BLOCK, 15, <u>117</u>, 120, 155, 775</li> <li>MPI_TYPE_CREATE_HNDEXED_BLOCK, 15, <u>117</u>, 120, 155, 776, 1041</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u>, 120, 155, 775</li> <li>MPI_TYPE_CREATE_NDEXED_BLOCK, 15, <u>117</u>, 120, 155, 775, 827</li> <li>MPI_TYPE_CREATE_NDEXED_BLOCK, 15, <u>117</u>, 120, 155, 775</li> <li>MPI_TYPE_CREATE_NDEXED_BLOCK, 15, <u>117</u>, 120, 155, 775</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, <u>127</u>, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		2	
$\begin{array}{cccccccccccccccccccccccccccccccccccc$		3	
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$		4	
96, 99, 100, 106, 107, 272, 487, 625, 648, 649 MPI_TEST_CANCELLED, 67–69, <u>89</u> , 623, 630, 650 MPI_TESTALL, 72, <u>75</u> , 510, 623–625, 628 MPI_TESTANY, 72, <u>73</u> , 74, 77, 510, 623–625, 628 MPI_TESTSOME, 72, <u>77</u> , 78, 510, 623–625, 628 MPI_TYPE_COP_TEST, <u>385</u> , <u>397</u> MPI_TYPE_C2F, <u>330</u> MPI_TYPE_CONTIGUOUS, 15, <u>117</u> , 120, 141, 155, 632, 686 MPI_TYPE_CREATE_PO_COMPLEX, 15, 155, 158, 223, 688, 780, <u>799</u> , 801 MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, <u>799</u> , 801 MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, <u>799</u> , 801 MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, <u>799</u> , 801 MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, <u>124</u> , 127, 128, 155, 775, 827 MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, <u>126</u> , 155, 827, 1039 MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u> , 120, 155, 775 MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>125</u> , 126, 155 MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>125</u> , 126, 155 MPI_TYPE_CREATE_RUSIZED, <u>29</u> , 117, 140, <u>144</u> , 155, 686, 776, 1041 MPI_TYPE_CREATE_RUSIZED, <u>29</u> , 117, 140, <u>144</u> , 155, 686, 776, 1041 MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u> , 131, 133, 455 MPI_TYPE_DELETE_ATTR, <u>359</u> , <u>373</u> , 1041 MPI_TYPE_DUP_FN, <u>370</u> , 853, 1036 MPI_TYPE_DELETE_ATTR, <u>359</u> , <u>373</u> , 1041 MPI_TYPE_FEE, <u>147</u> , 157, 371, 487, 494 MPI_TYPE_FEE, <u>147</u> , 157, 371, 487, 494 MPI_TYPE_GET_ATTR, <u>359</u> , <u>372</u> , 373, 788, <u>837</u> , 1041 MPI_TYPE_GET_CONTENTS, 154, <u>156</u> ,			MPI_TEST, 39, 67, <u>68</u> , 69, 70, 72, 74, 79, 88,
<ul> <li>648, 649</li> <li>MPI_TEST_CANCELLED, 67–69, <u>89</u>, 623, 630, 650</li> <li>MPI_TESTALL, 72, <u>75</u>, 510, 623–625, 628</li> <li>MPI_TESTANY, 72, <u>73</u>, 74, 77, 510, 623, 624, 628</li> <li>MPI_TOPO_TEST, 385, <u>397</u></li> <li>MPI_TYPE_C2F, <u>830</u></li> <li>MPI_TYPE_CONTIGUOUS, 15, <u>117</u>, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, <u>132</u>, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>125</u>, 126, 155</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>125</u>, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u>, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP_F, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_DUP_F, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_FEE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>, 877, 1041</li> </ul>			
<ul> <li>MP1_TEST_LCANCELLED, 67–69, 82, 623, 630, 650</li> <li>MP1_TESTALL, 72, 75, 510, 623–625, 628</li> <li>MP1_TESTSOME, 72, 77, 78, 510, 623–625, 628</li> <li>MP1_TOPO_TEST, 385, 397</li> <li>MP1_TOPO_TEST, 385, 397</li> <li>MP1_TYPE_C2F, 830</li> <li>MP1_TYPE_CONTIGUOUS, 15, 117, 120, 141, 155, 632, 686</li> <li>MP1_TYPE_CREATE_DARRAY, 15, 39, 132, 133, 155</li> <li>MP1_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MP1_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MP1_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MP1_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MP1_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 120, 155, 775</li> <li>MP1_TYPE_CREATE_HINDEXED_BLOCK, 15, 125, 126, 155</li> <li>MP1_TYPE_CREATE_RESIZED, 23, 117, 120, 155, 775</li> <li>MP1_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 680, 766, 1041</li> <li>MP1_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 158, 1041</li> <li>MP1_TYPE_CREATE_STRUCT, 15, 23, 117, 140, 144, 155, 188, 775, 827</li> <li>MP1_TYPE_CREATE_STRUCT, 15, 23, 117, 140, 144, 155, 158, 1041</li> <li>MP1_TYPE_CREATE_STRUCT, 15, 23, 117, 140, 144, 155, 218, 775, 827</li> <li>MP1_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MP1_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MP1_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MP1_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MP1_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MP1_TYPE_GET_ATTR, 359, 372, 373, 788, 837, 1041</li> </ul>			648,  649
<ul> <li>MPI_TESTALL, 72, 75, 510, 623-625, 628</li> <li>MPI_TESTANY, 72, 73, 74, 77, 510, 623, 624, 628</li> <li>MPI_TESTSOME, 72, 77, 78, 510, 623-625, 628</li> <li>MPI_TOPO_TEST, 385, 397</li> <li>MPI_TYPE_C2F, 830</li> <li>MPI_TYPE_CONTIGUOUS, 15, 117, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, 132, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 125, 126, 155, 158, 223, 686, 776, 1039</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE, 447, 157, 371, 487, 494</li> <li>MPI_TYPE_REE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_GET_ATTR, 359, 373, 788, 837, 1041</li> </ul>			
<ul> <li>MPI_TESTANY, 72, 73, 74, 77, 510, 623, 624, 628</li> <li>MPI_TESTSOME, 72, 77, 78, 510, 623–625, 628</li> <li>MPI_TOPO_TEST, 385, 397</li> <li>MPI_TYPE_C2F, 830</li> <li>MPI_TYPE_CONTIGUOUS, 15, 117, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, 132, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_RESTRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MPI_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MPI_TYPE_FREE_LEXEVAL, 359, 371, 373</li> <li>MPI_TYPE_FREE_LEXEVAL, 359, 371, 373</li> <li>MPI_TYPE_GET_ATTR, 359, 372, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156,</li> </ul>		8	
1 628 MPI_TESTSOME, 72, $\underline{77}$ , 78, 510, 623-625, 628 MPI_TOPO_TEST, 385, 397 MPI_TYPE_C2F, <u>830</u> MPI_TYPE_CONTIGUOUS, 15, <u>117</u> , 120, 141, 155, 632, 686 MPI_TYPE_CREATE_DARRAY, 15, 39, <u>132</u> , 133, 155 MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801 MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801 MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801 MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801 MPI_TYPE_CREATE_HINDEXED_15, 23, 117, 123, 124, 127, 128, 155, 775, 827 MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039 MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039 MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 117, 126, 155, 827, 1039 MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u> , 373, 837, 863, 866, 1046 MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u> , 373, 837, 863, 766, 1041 MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155 MPI_TYPE_DUP_15, <u>148</u> , 155, 1041 MPI_TYPE_DUP_FN, <u>370</u> , 853, 1036 MPI_TYPE_DUP_FN, <u>370</u> , 853, 1036 MPI_TYPE_FREE, <u>447</u> , 157, 371, 487, 494 MPI_TYPE_FREE, <u>447</u> , 157, 371, 487, 494 MPI_TYPE_REET_CONTENTS, 154, <u>156</u> ,		9	
<ul> <li>MPI_TESTSOME, 72, 77, 78, 510, 623–625, 628</li> <li>MPI_TOPO_TEST, 385, 397</li> <li>MPI_TYPE_C2F, 830</li> <li>MPI_TYPE_COMMIT, 146, 147, 831</li> <li>MPI_TYPE_CONTIGUOUS, 15, 117, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, 132, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 798, 799–801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 1041</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_FREE, L47, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE, ATTR, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156,</li> </ul>		10	
<ul> <li>628</li> <li>MPI_TOPO_TEST, 385, <u>397</u></li> <li>MPI_TYPE_C2F, <u>830</u></li> <li>MPI_TYPE_COMMIT, <u>146</u>, 147, 831</li> <li>MPI_TYPE_CONTIGUOUS, 15, <u>117</u>, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, <u>132</u>, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, <u>799</u>, 801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u>, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_KEYVAL, <u>359</u>, <u>370</u>, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_KEYVAL, <u>359</u>, <u>370</u>, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 455</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 455</li> <li>MPI_TYPE_DELETE_ATTR, <u>359</u>, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_FREE_<u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, <u>359</u>, <u>371</u>, 373</li> <li>MPI_TYPE_FREE_KEYVAL, <u>359</u>, <u>372</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		11	
<ul> <li>MPI_TOPO_TEST, 385, 397</li> <li>MPI_TYPE_C2F, 830</li> <li>MPI_TYPE_COMMIT, 146, 147, 831</li> <li>MPI_TYPE_CONTIGUOUS, 15, 117, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, 132, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_NDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_NDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_F2C, 830</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 372, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_ATTR, 359, 372, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156,</li> </ul>		12	
<ul> <li>MPI_TYPE_C2F, <u>830</u></li> <li>MPI_TYPE_COMMIT, <u>146</u>, 147, 831</li> <li>MPI_TYPE_CONTIGUOUS, 15, <u>117</u>, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, <u>132</u>, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, <u>798</u>, 799–801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u>, <u>126</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, <u>140</u>, <u>144</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, <u>127</u>, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 455</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE_KEYVAL, <u>359</u>, <u>371</u>, 373</li> </ul>		13	
<ul> <li>MPI_TYPE_COMMIT, <u>146</u>, 147, 831</li> <li>MPI_TYPE_CONTIGUOUS, 15, <u>117</u>, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, <u>132</u>, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u>, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u>, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u>, <u>126</u>, 155, 775</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, <u>140</u>, <u>144</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, <u>140</u>, <u>144</u>, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 455</li> <li>MPI_TYPE_DUP_15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP, 5, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_FEEX_<u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_GRET_ATTR, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GRET_CONTENTS, 154, <u>156</u>,</li> </ul>		14	
<ul> <li>MPI_TYPE_CONTIGUOUS, 15, <u>117</u>, 120, 141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, <u>132</u>, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 799, 801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u>, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u>, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u>, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, <u>144</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, <u>127</u>, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 455</li> <li>MPI_TYPE_DUP_15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP, 5, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GRET_ATTR, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GRET_CONTENTS, 154, <u>156</u>,</li> </ul>		15	
<ul> <li>141, 155, 632, 686</li> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, <u>132</u>, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, <u>798</u>, 799-801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, <u>126</u>, 156, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>117</u>, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u>, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, <u>144</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 155</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 155</li> <li>MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP, 5, <u>370</u>, 373, 1041</li> <li>MPI_TYPE_REXTENT, 23, 775, 1038</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		16	
<ul> <li>MPI_TYPE_CREATE_DARRAY, 15, 39, <u>132</u>, 133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, <u>798</u>, 799-801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 17, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, <u>117</u>, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u>, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, <u>140</u>, <u>144</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP, 570, 853, 1036</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>			
<ul> <li>133, 155</li> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 798, 799-801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, 370, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MPI_TYPE_DUP_15, 148, 155, 1041</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_F2C, 830</li> <li>MPI_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_GET_ATTR, 359, 372, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156,</li> </ul>			
<ul> <li>MPI_TYPE_CREATE_F90_COMPLEX, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 798, 799-801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, 370, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 455</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP_15, 148, 155, 1041</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>			· · · · · · · · · · · · · · · · · · ·
<ul> <li>155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 798, 799-801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, 370, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_F2C, 830</li> <li>MPI_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_GET_ATTR, 359, 372, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156,</li> </ul>			
<ul> <li>MPI_TYPE_CREATE_F90_INTEGER, 15, 155, 158, 223, 688, 780, 799, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155, 158, 223, 688, 780, 798, 799-801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, 370, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156,</li> </ul>			
<ul> <li><sup>22</sup> 155, 158, 223, 688, 780, <u>799</u>, 801</li> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155,</li> <li><sup>23</sup> 158, 223, 688, 780, <u>798</u>, 799-801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23,</li> <li><sup>25</sup> 117, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK,</li> <li><sup>27</sup> 15, 117, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23,</li> <li><sup>28</sup> 117, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK,</li> <li><sup>31</sup> 15, <u>125</u>, 126, 155</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK,</li> <li><sup>33</sup> 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117,</li> <li><sup>40</sup> 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117,</li> <li><sup>127</sup> 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18,</li> <li><sup>129</sup> 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_PE_CREATE_SUBARRAY, 15, 18,</li> <li><sup>129</sup> 131, 133, 155</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_CREATE_ATTR, 359, <u>373</u>, 373</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788,</li> <li><sup>837</sup>, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		21	
<ul> <li>MPI_TYPE_CREATE_F90_REAL, 15, 155,</li> <li>158, 223, 688, 780, <u>798</u>, 799-801, 1043</li> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23,</li> <li>117, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK,</li> <li>15, 117, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23,</li> <li>117, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK,</li> <li>15, <u>125</u>, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u>,</li> <li>373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117,</li> <li><u>140</u>, <u>144</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117,</li> <li><u>127</u>, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18,</li> <li><u>129</u>, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP, FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_GET_ATTR, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788,</li> <li><u>837</u>, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		22	
<ul> <li>MPI_TYPE_CREATE_HINDEXED, 15, 23, 117, 123, 124, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u>, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_GET_ATTR, 359, <u>371</u>, 373, MPI_TYPE_GET_ATTR, 359, <u>371</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		23	
<ul> <li>26 117, 123, <u>124</u>, 127, 128, 155, 775, 827</li> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, <u>126</u>, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, <u>120</u>, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, <u>125</u>, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u>, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, <u>144</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, <u>127</u>, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 155</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		24	158, 223, 688, 780, <u>798</u> , 799–801, 1043
<ul> <li>MPI_TYPE_CREATE_HINDEXED_BLOCK, 15, 117, 126, 155, 827, 1039</li> <li>MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, 120, 155, 775</li> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, 370, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 455</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		25	
15, 117, 126, 155, 827, 1039         MPI_TYPE_CREATE_HVECTOR, 15, 23, 117, 120, 155, 775         MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155         MPI_TYPE_CREATE_KEYVAL, 359, 370, 373, 837, 863, 866, 1046         MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041         MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827         MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155         MPI_TYPE_DELETE_ATTR, 359, 373, 1041         MPI_TYPE_DUP_FN, 370, 853, 1036         MPI_TYPE_FREE, 147, 157, 371, 487, 494         MPI_TYPE_FREE_KEYVAL, 359, 371, 373         MPI_TYPE_GET_ATTR, 359, 371, 373         MPI_TYPE_FREE_KEYVAL, 359, 371, 373         MPI_TYPE_GET_CONTENTS, 154, 156,		26	
rof!       29 29 29 29 29 29 30 30 30 31 31 31 31 31 31 32 31 31 31 32 31 31 31 31 31 31 31 31 31 31 31 31 31		27	
117, 120, 155, 775         MPI_TYPE_CREATE_INDEXED_BLOCK,         1       15, 125, 126, 155         MPI_TYPE_CREATE_KEYVAL, 359, 370,         3       373, 837, 863, 866, 1046         34       MPI_TYPE_CREATE_RESIZED, 23, 117,         35       140, 144, 155, 686, 776, 1041         MPI_TYPE_CREATE_STRUCT, 15, 23, 117,         36       127, 128, 141, 155, 218, 775, 827         MPI_TYPE_CREATE_SUBARRAY, 15, 18,         38       129, 131, 133, 155         MPI_TYPE_DELETE_ATTR, 359, 373, 1041         MPI_TYPE_DUP, 15, 148, 155, 1041         MPI_TYPE_DUP_FN, 370, 853, 1036         MPI_TYPE_FREE, 147, 157, 371, 487, 494         MPI_TYPE_FREE_KEYVAL, 359, 371, 373         MPI_TYPE_GET_ATTR, 359, 372, 373, 788,         37         MPI_TYPE_GET_CONTENTS, 154, 156,		28	
<ul> <li>MPI_TYPE_CREATE_INDEXED_BLOCK, 15, 125, 126, 155</li> <li>MPI_TYPE_CREATE_KEYVAL, 359, 370, 373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_EXTENT, 23, 775, 1038</li> <li>MPI_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_GET_ATTR, 359, 372, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156,</li> </ul>	or!	29	
31       15, <u>125</u> , 126, 155         32       MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u> ,         33       373, 837, 863, 866, 1046         34       MPI_TYPE_CREATE_RESIZED, 23, 117,         35       140, <u>144</u> , 155, 686, 776, 1041         36       MPI_TYPE_CREATE_STRUCT, 15, 23, 117,         37       127, 128, 141, 155, 218, 775, 827         37       MPI_TYPE_CREATE_SUBARRAY, 15, 18,         38 <u>129</u> , 131, 133, 155         39       MPI_TYPE_DELETE_ATTR, 359, <u>373</u> , 1041         40       MPI_TYPE_DUP, 15, <u>148</u> , 155, 1041         41       MPI_TYPE_DUP_FN, <u>370</u> , 853, 1036         42       MPI_TYPE_FREE, <u>147</u> , 157, 371, 487, 494         44       MPI_TYPE_FREE, <u>147</u> , 157, 371, 487, 494         44       MPI_TYPE_GET_ATTR, 359, <u>371</u> , 373         45       837, 1041         47       MPI_TYPE_GET_CONTENTS, 154, <u>156</u> ,	her	30	
13, 125, 126, 135         32       MPI_TYPE_CREATE_KEYVAL, 359, <u>370</u> , 33         33       373, 837, 863, 866, 1046         34       MPI_TYPE_CREATE_RESIZED, 23, 117, 140, <u>144</u> , 155, 686, 776, 1041         35       140, <u>144</u> , 155, 686, 776, 1041         36       MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827         37       MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155         39       MPI_TYPE_DELETE_ATTR, 359, <u>373</u> , 1041         40       MPI_TYPE_DUP, 15, <u>148</u> , 155, 1041         41       MPI_TYPE_DUP_FN, <u>370</u> , 853, 1036         42       MPI_TYPE_F2C, <u>830</u> 43       MPI_TYPE_FREE, <u>147</u> , 157, 371, 487, 494         44       MPI_TYPE_GET_ATTR, 359, <u>371</u> , 373         45       MPI_TYPE_GET_ATTR, 359, <u>372</u> , 373, 788, 837, 1041         47       MPI_TYPE_GET_CONTENTS, 154, <u>156</u> ,		31	
<ul> <li>373, 837, 863, 866, 1046</li> <li>MPI_TYPE_CREATE_RESIZED, 23, 117, 140, 144, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, 127, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, 129, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MPI_TYPE_DUP, 15, 148, 155, 1041</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_GET_ATTR, 359, 372, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156,</li> </ul>			
<ul> <li>MPI_TYPE_CREATE_RESIZED, 23, 117,</li> <li>140, <u>144</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117,</li> <li><u>127</u>, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18,</li> <li><u>129</u>, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788,</li> <li><u>837</u>, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>	.9		
<ul> <li><sup>35</sup> 140, <u>144</u>, 155, 686, 776, 1041</li> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117, <u>127</u>, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_EXTENT, 23, 775, 1038</li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>			
<ul> <li>MPI_TYPE_CREATE_STRUCT, 15, 23, 117,</li> <li><u>127</u>, 128, 141, 155, 218, 775, 827</li> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18,</li> <li><u>129</u>, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li>MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788,</li> <li><u>837</u>, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>			
<ul> <li><sup>36</sup> <u>127</u>, 128, 141, 155, 218, 775, 827</li> <li><sup>37</sup> MPI_TYPE_CREATE_SUBARRAY, 15, 18, <u>129</u>, 131, 133, 155</li> <li><sup>39</sup> MPI_TYPE_DELETE_ATTR, 359, <u>373</u>, 1041</li> <li><sup>40</sup> MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li><sup>41</sup> MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li><sup>42</sup> MPI_TYPE_F2C, <u>830</u></li> <li><sup>43</sup> MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li><sup>44</sup> MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li><sup>44</sup> MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li><sup>45</sup> MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788, <u>837</u>, 1041</li> <li><sup>47</sup> MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>			
<ul> <li>MPI_TYPE_CREATE_SUBARRAY, 15, 18,</li> <li>129, 131, 133, 155</li> <li>MPI_TYPE_DELETE_ATTR, 359, 373, 1041</li> <li>MPI_TYPE_DUP, 15, 148, 155, 1041</li> <li>MPI_TYPE_DUP_FN, 370, 853, 1036</li> <li>MPI_TYPE_EXTENT, 23, 775, 1038</li> <li>MPI_TYPE_F2C, 830</li> <li>MPI_TYPE_FREE, 147, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, 371, 373</li> <li>MPI_TYPE_GET_ATTR, 359, 372, 373, 788,</li> <li>837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, 156,</li> </ul>			$\underline{127}, 128, 141, 155, 218, 775, 827$
129, 131, 133, 133         39       MPI_TYPE_DELETE_ATTR, 359, <u>373</u> , 1041         40       MPI_TYPE_DUP, 15, <u>148</u> , 155, 1041         41       MPI_TYPE_DUP_FN, <u>370</u> , 853, 1036         42       MPI_TYPE_EXTENT, 23, 775, 1038         43       MPI_TYPE_F2C, <u>830</u> 44       MPI_TYPE_FREE, <u>147</u> , 157, 371, 487, 494         45       MPI_TYPE_GET_ATTR, 359, <u>371</u> , 373         46       837, 1041         47       MPI_TYPE_GET_CONTENTS, 154, <u>156</u> ,		37	MPI_TYPE_CREATE_SUBARRAY, 15, 18,
<ul> <li>40 MPI_TYPE_DUP, 15, <u>148</u>, 155, 1041</li> <li>41 MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>42 MPI_TYPE_EXTENT, 23, 775, 1038</li> <li>43 MPI_TYPE_F2C, <u>830</u></li> <li>44 MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>44 MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>45 MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788,</li> <li>46 837, 1041</li> <li>47 MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		38	
<ul> <li>MPI_TYPE_DUP_FN, <u>370</u>, 853, 1036</li> <li>MPI_TYPE_EXTENT, 23, 775, 1038</li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		39	
<ul> <li>MPI_TYPE_EXTENT, 23, 775, 1038</li> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788, 837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		40	
<ul> <li>MPI_TYPE_F2C, <u>830</u></li> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788,</li> <li>837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		41	
<ul> <li>MPI_TYPE_FREE, <u>147</u>, 157, 371, 487, 494</li> <li>MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li>MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788,</li> <li>837, 1041</li> <li>MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		42	
<ul> <li><sup>44</sup> MPI_TYPE_FREE_KEYVAL, 359, <u>371</u>, 373</li> <li><sup>45</sup> MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788,</li> <li><sup>46</sup> 837, 1041</li> <li><sup>47</sup> MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		43	
<ul> <li><sup>45</sup> MPI_TYPE_GET_ATTR, 359, <u>372</u>, 373, 788,</li> <li><sup>46</sup> 837, 1041</li> <li><sup>47</sup> MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		44	
<ul> <li>46 837, 1041</li> <li>47 MPI_TYPE_GET_CONTENTS, 154, <u>156</u>,</li> </ul>		45	· · · · ·
47 MPI_TYPE_GET_CONTENTS, $154$ , $156$ ,		46	
			10. 100, 01.

MPI\_TYPE\_GET\_ENVELOPE, 153, 154, 157, 158, 800, 827 MPI\_TYPE\_GET\_EXTENT, 23, 142, 146, 775, 803, 834 MPI\_TYPE\_GET\_EXTENT\_X, 143, 1038 MPI\_TYPE\_GET\_NAME, 378, 1041 MPI\_TYPE\_GET\_TRUE\_EXTENT, 144, 145 MPI\_TYPE\_GET\_TRUE\_EXTENT\_X, 144, 145, 1038 MPI\_TYPE\_HINDEXED, 23, 775, 1038 MPI\_TYPE\_HVECTOR, 23, 775, 1038 MPI\_TYPE\_INDEXED, 15, 122, 123, 125, 155 MPI\_TYPE\_LB, 23, 775, 1038 MPI\_TYPE\_MATCH\_SIZE, 780, 802, 803, 1041 MPI\_TYPE\_NULL\_COPY\_FN, 370, 853, 1036 MPI\_TYPE\_NULL\_DELETE\_FN, <u>370</u>, 853, 1036, 1041 MPI\_TYPE\_SET\_ATTR, 359, 372, 373, 788, 837, 841, 1041 MPI\_TYPE\_SET\_NAME, 378, 1041 MPI\_TYPE\_SIZE, 139, 140, 716, 1039 MPI\_TYPE\_SIZE\_X, 140, 1038 MPI\_TYPE\_STRUCT, 23, 775, 1038 MPI\_TYPE\_UB, 23, 775, 1038 MPI\_TYPE\_VECTOR, 15, 118, 119, 120, 123, 155 MPI\_UNPACK, <u>172</u>, 173, 174, 178, 694 MPI\_UNPACK\_EXTERNAL, 8, 179, 801 MPI\_UNPUBLISH\_NAME, 464, 530, 531 MPI\_USER\_FUNCTION, 230, 827 MPI\_WAIT, 37, 39, 67, 68-71, 73, 74, 88, 96, 99, 100, 106, 107, 246, 272, 301, 487, 510, 621, 625, 648, 649, 675, 696, 698, 807, 808, 814, 815, 818 MPI\_WAITALL, 72, 74, 75, 246, 302, 510, 574, 623-625, 628 MPI\_WAITANY, 51, 72, 73, 78, 510, 623, 624, 628 MPI\_WAITSOME, 72, 76, 77, 78, 510, 623-625, 628 MPI\_WIN\_ALLOCATE, 542, <u>546</u>, 547, 549, 555-557, 562, 595, 792, 794, 1033, 1035 MPI\_WIN\_ALLOCATE\_CPTR, 547, 1035 MPI\_WIN\_ALLOCATE\_SHARED, 333, 542, 548, 549, 551, 556, 557, 595, 794, 1033, 1035 MPI\_WIN\_ALLOCATE\_SHARED\_CPTR, 549.1035 MPI\_WIN\_ATTACH, 552, 553, 554, 555, 595

MPI\_WIN\_C2F, <u>830</u>

MPI\_WIN\_CALL\_ERRHANDLER, 467, 469

Unofficial Draft for Comment Only

This is a callback prototype

MPI\_WIN\_COMPLETE, 555, 584, 588, 589-592, 600, 601, 607 MPI\_WIN\_CREATE, 511, 542, 543, 545, 547, 549, 553–555, 557, 599 MPI\_WIN\_CREATE\_DYNAMIC, 464, 542, 552, 553, 554, 556, 557, 600 MPI\_WIN\_CREATE\_ERRHANDLER, 452, 455, 456, 863, 866, 1041 MPI\_WIN\_CREATE\_KEYVAL, 359, 366, 373, 837, 862, 865, 1046 MPI\_WIN\_DELETE\_ATTR, 359, 369, 373 MPI\_WIN\_DETACH, 552, 554, 556 MPI\_WIN\_DUP\_FN, 366, 853, 1036 MPI\_WIN\_F2C, 830 MPI\_WIN\_FENCE, 555, 564, 584, <u>586</u>, 587, 588, 598-601, 604, 609, 818 MPI\_WIN\_FLUSH, 550, 574, 576, 595, 596, 600, 601, 614, 616 MPI\_WIN\_FLUSH\_ALL, 574, 576, 596, 600, 601 MPI\_WIN\_FLUSH\_LOCAL, 574, 596, 600 MPI\_WIN\_FLUSH\_LOCAL\_ALL, 574, 597, 600 MPI\_WIN\_FREE, 367, 487, 494, 538, 555, 556 MPI\_WIN\_FREE\_KEYVAL, 359, 368, 373 MPI\_WIN\_GET\_ATTR, 359, 368, 373, 556, 788, 837, 840 MPI\_WIN\_GET\_ERRHANDLER, 452, 456, 1047 MPI\_WIN\_GET\_GROUP, 557 MPI\_WIN\_GET\_INFO, 558, 559, 1033, 1039 MPI\_WIN\_GET\_NAME, 379 MPI\_WIN\_LOCK, 544, 555, 584, 592, 593-595, 598, 599, 601, 605, 606 MPI\_WIN\_LOCK\_ALL, 544, 584, 593, 594, 598, 599, 601, 606, 614, 615 MPI\_WIN\_NULL\_COPY\_FN, 366, 853, 1036 MPI\_WIN\_NULL\_DELETE\_FN, 367, 853, 1036MPI\_WIN\_POST, 555, 584, 588, 589, 590-592, 594, 598, 601, 607, 609 MPI\_WIN\_SET\_ATTR, 359, 368, 373, 556, 788, 837, 841 MPI\_WIN\_SET\_ERRHANDLER, 452, 456 MPI\_WIN\_SET\_INFO, 558, 1033, 1039 MPI\_WIN\_SET\_NAME, 379 MPI\_WIN\_SHARED\_QUERY, 549, 550, 794, 1035 MPI\_WIN\_SHARED\_QUERY\_CPTR, 552, 1035 MPI\_WIN\_START, 555, 584, 589-592, 598, 608, 614 MPI\_WIN\_SYNC, <u>597</u>, 601, 603, 604, 607, 614, 616

MPI\_WIN\_TEST, 590, 591 MPI\_WIN\_UNLOCK, 555, 576, 584, 593, 595,  $\mathbf{2}$ 600, 601, 604-606 MPI\_WIN\_UNLOCK\_ALL, 576, 584, 593, 594, 600, 601, 604, 615 MPI\_WIN\_WAIT, 555, 584, 590, 591, 592, 594, 601, 604, 607, 608 MPI\_WTICK, 24, <u>470</u>, 748 MPI\_WTIME, 14, 24, 446, 469, 470, 716, 735, 748 mpiexec, 481, 482, 484, 505, 506, 1034 10 mpirun, 506 11 12PMPI\_, 713, 788 13 PMPI\_AINT\_ADD, 24 14PMPI\_AINT\_DIFF, 24 15PMPI\_ISEND, 788, 791 PMPI\_T\_EVENT\_COPY, 749 16PMPI\_T\_EVENT\_GET\_SOURCE, 749 17PMPI\_T\_EVENT\_GET\_TIMESTAMP, 749 18 PMPI\_T\_EVENT\_READ, 749 19 PMPI\_T\_PVAR\_READ, 749 20PMPI\_T\_PVAR\_READRESET, 749 21PMPI\_T\_PVAR\_RESET, 749 22 PMPI\_T\_PVAR\_START, 749 PMPI\_T\_PVAR\_STOP, 749 23PMPI\_T\_PVAR\_WRITE, 749  $^{24}$ PMPI\_T\_SOURCE\_GET\_TIMESTAMP, 749 25#-other PMPI\_WTICK, 24 26PMPI\_WTIME, 24 #-error!? 27

Are these functions really new functions? # 20 new routines MPI\_T\_CATEGORY\_GET\_EVENTS MPI\_T\_CATEGORY\_GET\_NUM\_EVENTS MPI\_T\_EVENT\_CALLBACK\_GET\_INFO MPI\_T\_EVENT\_CALLBACK\_SET\_INFO MPI\_T\_EVENT\_COPY MPI\_T\_EVENT\_GET\_INDEX MPI\_T\_EVENT\_GET\_INFO MPI\_T\_EVENT\_GET\_NUM MPI T EVENT GET SOURCE MPI\_T\_EVENT\_GET\_TIMESTAMP MPI\_T\_EVENT\_HANDLE\_ALLOC MPI\_T\_EVENT\_HANDLE\_FREE MPI\_T\_EVENT\_HANDLE\_GET\_INFO MPI\_T\_EVENT\_HANDLE\_SET\_INFO MPI\_T\_EVENT\_READ MPI\_T\_EVENT\_REGISTER\_CALLBACK MPI\_T\_EVENT\_SET\_DROPPED\_HANDLER MPI\_T\_SOURCE\_GET\_INFO MPI\_T\_SOURCE\_GET\_NUM MPI\_T\_SOURCE\_GET\_TIMESTAMP # 11 PMPI\_T\_ functions listed in index Removed in PMPI\_T\_EVENT\_COPY PR423 PMPI\_T\_EVENT\_GET\_SOURCE PMPI\_T\_EVENT\_GET\_TIMESTAMP PMPI\_T\_EVENT\_READ PMPI\_T\_PVAR\_READ PMPI\_T\_PVAR\_READRESET PMPI\_T\_PVAR\_RESET PMPI\_T\_PVAR\_START Unofficial Draft for Com PMPI\_T\_PVAR\_STOP PMPI\_T\_PVAR\_WRITE PMPI\_T\_SOURCE\_GET\_TIMESTAMP