### MPI: A Message-Passing Interface Standard Version 3.2

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

December 6, 2016

1	This document describes the Message-Passing Interface (MPI) standard, version 3.1.
2	The MPI standard includes point-to-point message-passing, collective communications, group
3	and communicator concepts, process topologies, environmental management, process cre-
4	ation and management, one-sided communications, extended collective operations, external
5	interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for
6	C and Fortran are defined.
7	Historically, the evolution of the standards is from MPI-1.0 (May 5, 1994) to MPI-1.1
8	(June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and
9	published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality,
10	to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2
11	and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008),
12	combining the previous documents. Version MPI-2.2 (September 4, 2009) added additional
13	clarifications and seven new routines. Version MPI-3.0 (September 21, 2012) is an extension
14	of MPI-2.2. This version, MPI-3.1, adds clarifications and minor extensions to MPI-3.0
15	
16	Comments. Please send comments on MPI to the MPI Forum as follows:
17	
18	1. Subscribe to http://lists.mpi-forum.org/mailman/listinfo.cgi/mpi-comments
19	2. Send your comment to: <u>mpi-comments@mpi-forum.org</u> , together with the URL of
20	the version of the MPI standard and the page and line numbers on which you are
21	commenting. Only use the official versions.
22	
23	Your comment will be forwarded to MPI Forum committee members for consideration.
24	Messages sent from an unsubscribed e-mail address will not be considered.
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39 40	
40	
42	
43	
44 45	©1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015 University of Tennessee, Knoxville,
45 46	Tennessee. Permission to copy without fee all or part of this material is granted, provided
40 47	the University of Tennessee copyright notice and the title of this document appear, and
48	notice is given that copying is by permission of the University of Tennessee.
40	notice is given that copying is by permission of the University of Tennessee.

Version 3.1: June 4, 2015. This document contains mostly corrections and clarifications to the MPI-3.0 document. The largest change is a correction to the Fortran bindings introduced in MPI-3.0. Additionally, new functions added include routines to manipulate MPI\_Aint values in a portable manner, nonblocking collective I/O routines, and routines to get the index value by name for MPI\_T performance and control variables.

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

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

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

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

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

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

 $\mathbf{2}$ 

 $^{24}$ 

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

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

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

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

- ...

# Contents

A	cknov	wledgments	ix
1	Intr	oduction to MPI	1
	1.1	Overview and Goals	1
	1.2	Background of MPI-1.0	2
	1.3	Background of MPI-1.1, MPI-1.2, and MPI-2.0	2
	1.4	Background of MPI-1.3 and MPI-2.1	3
	1.5	Background of MPI-2.2.	4
	1.6	Background of MPI-3.0.	4
	1.7	Background of MPI-3.1.	4
	1.8	Who Should Use This Standard?	5
	1.9	What Platforms Are Targets For Implementation?	5
	1.10	What Is Included In The Standard?	5
		What Is Not Included In The Standard?	6
		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
	2.5	Data Types	12
		2.5.1 Opaque Objects	12
		2.5.2 Array Arguments	14
		2.5.3 State	14
		2.5.4 Named Constants	15
		2.5.5 Choice	16
		2.5.6 Absolute Addresses and Relative Address Displacements	16
		2.5.7 File Offsets	16
		2.5.8 Counts	17
	2.6	Language Binding	17
		2.6.1 Deprecated and Removed Names and Functions	17
		2.6.2 Fortran Binding Issues	19
		2.6.3 C Binding Issues	19
		2.6.4 Functions and Macros	20
	2.7	Processes	$20^{-0}$
	2.8	Error Handling	20

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

		4.1.9	Commit and Free	111
		4.1.10	Duplicating a Datatype	113
		4.1.11	Use of General Datatypes in Communication	113
		4.1.12	Correct Use of Addresses	117
		4.1.13	Decoding a Datatype	118
				125
	4.2			134
	4.3		*	140
5	Coll	ective	Communication	143
	5.1	Introd	uction and Overview	143
	5.2	Comm	unicator Argument	146
		5.2.1		146
		5.2.2		147
		5.2.3		148
	5.3			149
	5.4		* · · · · · · · · · · · · · · · · · · ·	150
		5.4.1		150
	5.5	-		151
	0.0	5.5.1		154
	5.6			161
	0.0	5.6.1		164
	5.7			167
	0.1	5.7.1		169
	5.8			170
	5.9		,	175
	0.0	5.9.1	-	176
		5.9.2		178
		5.9.3		180
		5.9.4		181
		5.9.5		185
		0.9.0		188
		5.9.6	*	189
		5.9.0 5.9.7		109
	5 10			191
	5.10			192 192
	F 11			193
	5.11			195
				195
				196
	F 10			197
	5.12		5 I	198
			0	200
		5.12.2		201
			• 0 -	201
				202
				204
		5.12.5	Nonblocking Gather-to-all	206

		5.12.6	Nonblocking All-to-All Scatter/Gather
		5.12.7	Nonblocking Reduce
		5.12.8	Nonblocking All-Reduce
		5.12.9	Nonblocking Reduce-Scatter with Equal Blocks
		5.12.1	0 Nonblocking Reduce-Scatter
		5.12.1	1 Nonblocking Inclusive Scan
		5.12.1	2 Nonblocking Exclusive Scan
	5.13	Correc	ctness
6	Gro	oups, C	Contexts, Communicators, and Caching 225
	6.1	Introd	uction $\ldots \ldots 225$
		6.1.1	Features Needed to Support Libraries
		6.1.2	MPI's Support for Libraries
	6.2	Basic	Concepts
		6.2.1	Groups
		6.2.2	Contexts
		6.2.3	Intra-Communicators
		6.2.4	Predefined Intra-Communicators
	6.3	Group	Management
		6.3.1	Group Accessors
		6.3.2	Group Constructors
		6.3.3	Group Destructors
	6.4	Comm	nunicator Management
		6.4.1	Communicator Accessors
		6.4.2	Communicator Constructors
		6.4.3	Communicator Destructors
		6.4.4	Communicator Info
	6.5	Motiv	ating Examples
		6.5.1	Current Practice $\#1$
		6.5.2	Current Practice $\#2$
		6.5.3	(Approximate) Current Practice $\#3$
		6.5.4	Example $\#4$
		6.5.5	Library Example $\#1$
		6.5.6	Library Example $\#2$
	6.6	Inter-0	Communication
		6.6.1	Inter-communicator Accessors
		6.6.2	Inter-communicator Operations
		6.6.3	Inter-Communication Examples
			Example 1: Three-Group "Pipeline"
			Example 2: Three-Group "Ring"
	6.7	Cachi	ng
		6.7.1	Functionality
		6.7.2	Communicators
		6.7.3	Windows
		6.7.4	Datatypes
		6.7.5	Error Class for Invalid Keyval
		6.7.6	Attributes Example
	6.8	Namir	ng Objects

<ul> <li>6.9.1 Basic Statements</li></ul>	287 288 288 288 289 <b>291</b> 292 292 292 292 294 294 294 294
Static Communicator Allocation       Dynamic Communicator Allocation         The General Case       The General Case         7 Process Topologies       7         7.1 Introduction       7         7.2 Virtual Topologies       7         7.3 Embedding in MPI       7         7.4 Overview of the Functions       7         7.5 Topology Constructors       7         7.5.1 Cartesian Convenience Function: MPI_DIMS_CREATE       7         7.5.3 Graph Constructor       7         7.5.4 Distributed Graph Constructor       7         7.5.5 Topology Inquiry Functions       7         7.5.6 Cartesian Shift Coordinates       7	288 288 289 <b>291</b> 292 292 292 294 294 294 294 294
Dynamic Communicator Allocation	288 289 <b>291</b> 292 292 292 292 294 294 294 294 294
The General Case         7       Process Topologies         7.1       Introduction         7.2       Virtual Topologies         7.3       Embedding in MPI         7.4       Overview of the Functions         7.5       Topology Constructors         7.5.1       Cartesian Constructor         7.5.2       Cartesian Convenience Function: MPI_DIMS_CREATE         7.5.3       Graph Constructor         7.5.4       Distributed Graph Constructor         7.5.5       Topology Inquiry Functions         7.5.6       Cartesian Shift Coordinates	289 <b>291</b> 292 292 292 294 294 294 294 294
<ul> <li>7 Process Topologies</li> <li>7.1 Introduction</li></ul>	<ul> <li>291</li> <li>292</li> <li>292</li> <li>292</li> <li>294</li> <li>294</li> <li>294</li> <li>294</li> <li>296</li> </ul>
<ul> <li>7.1 Introduction</li></ul>	291 292 292 292 294 294 294 294 294
<ul> <li>7.1 Introduction</li></ul>	291 292 292 292 294 294 294 294 294
<ul> <li>7.2 Virtual Topologies</li> <li>7.3 Embedding in MPI</li> <li>7.4 Overview of the Functions</li> <li>7.5 Topology Constructors</li> <li>7.5.1 Cartesian Constructor</li> <li>7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE</li> <li>7.5.3 Graph Constructor</li> <li>7.5.4 Distributed Graph Constructor</li> <li>7.5.5 Topology Inquiry Functions</li> <li>7.5.6 Cartesian Shift Coordinates</li> </ul>	292 292 292 294 294 294 294 294 296
<ul> <li>7.3 Embedding in MPI</li> <li>7.4 Overview of the Functions</li> <li>7.5 Topology Constructors</li> <li>7.5.1 Cartesian Constructor</li> <li>7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE</li> <li>7.5.3 Graph Constructor</li> <li>7.5.4 Distributed Graph Constructor</li> <li>7.5.5 Topology Inquiry Functions</li> <li>7.5.6 Cartesian Shift Coordinates</li> </ul>	292 292 294 294 294 294 296
<ul> <li>7.4 Overview of the Functions</li></ul>	292 294 294 294 294 296
7.5Topology Constructors7.5.1Cartesian Constructor7.5.2Cartesian Convenience Function: MPI_DIMS_CREATE7.5.3Graph Constructor7.5.4Distributed Graph Constructor7.5.5Topology Inquiry Functions7.5.6Cartesian Shift Coordinates	294 294 294 296
<ul> <li>7.5.1 Cartesian Constructor</li></ul>	294 294 296
<ul> <li>7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE</li> <li>7.5.3 Graph Constructor</li></ul>	294 296
<ul> <li>7.5.3 Graph Constructor</li></ul>	296
<ul> <li>7.5.4 Distributed Graph Constructor</li></ul>	
7.5.5Topology Inquiry Functions7.5.6Cartesian Shift Coordinates	(1(1(1)
7.5.6 Cartesian Shift Coordinates	298
	304
	312
7.5.7 Partitioning of Cartesian Structures	313
7.5.8 Low-Level Topology Functions	314
7.6 Neighborhood Collective Communication	316
7.6.1 Neighborhood Gather	317
7.6.2 Neighbor Alltoall	320
7.7 Nonblocking Neighborhood Communication	325
7.7.1 Nonblocking Neighborhood Gather	326
7.7.2 Nonblocking Neighborhood Alltoall	328
7.8 An Application Example	331
8 MPI Environmental Management	335
8.1 Implementation Information	335
8.1.1 Version Inquiries	335
8.1.2 Environmental Inquiries	336
Tag Values	337
Host Rank	337
· · · · · · · · · · · · · · · · · · ·	337
IO Rank	001
	338
IO Rank	
IO Rank	338
IO Rank	338 338
IO Rank	338 338 338
IO Rank	338 338 338 339
IO Rank	338 338 338 339 342
IO Rank	338 338 338 339 342 344
IO Rank	338 338 338 339 342 344 344
IO Rank	338 338 338 339 342 344 346 347
IO Rank	338 338 338 339 342 344 346 347 349

	8.7	Startup	357
		8.7.1 Allowing User Functions at Process Termination	363
		8.7.2 Determining Whether MPI Has Finished	363
	8.8	Portable MPI Process Startup	364
9	The	Info Object	367
10	) Pro	cess Creation and Management	373
	10.1	Introduction	373
	10.2	The Dynamic Process Model	374
		10.2.1 Starting Processes	
		10.2.2 The Runtime Environment	374
	10.3	Process Manager Interface	
		10.3.1 Processes in MPI	376
		10.3.2 Starting Processes and Establishing Communication	
		10.3.3 Starting Multiple Executables and Establishing Communication	
		10.3.4 Reserved Keys	
		10.3.5 Spawn Example	
		Manager-worker Example Using MPI_COMM_SPAWN	
	10.4	Establishing Communication	
		10.4.1 Names, Addresses, Ports, and All That	
		10.4.2 Server Routines	
		10.4.3 Client Routines	
		10.4.4 Name Publishing	
		10.4.5 Reserved Key Values	
		10.4.6 Client/Server Examples	
		Simplest Example — Completely Portable.	
		$Ocean/Atmosphere - Relies on Name Publishing \dots \dots$	
	10 5	Simple Client-Server Example	
	10.5	Other Functionality	
		10.5.1 Universe Size	
		10.5.2 Singleton MPI_INIT	
		10.5.3 MPI_APPNUM	
		10.5.4 Releasing Connections	
		10.5.5 Another way to Establish MFT Communication	401
11		-Sided Communications	403
		Introduction	
	11.2	Initialization	
		11.2.1 Window Creation	
		11.2.2 Window That Allocates Memory	
		11.2.3 Window That Allocates Shared Memory	
		11.2.4 Window of Dynamically Attached Memory	
		11.2.5 Window Destruction	
		11.2.6 Window Attributes $\dots$	
	11.0	11.2.7 Window Info $\ldots$	
	11.3	Communication Calls	
		11.3.1 Put	420

	11.3.2 Get	422
	11.3.3 Examples for Communication Calls	423
	11.3.4 Accumulate Functions	425
	Accumulate Function	426
	Get Accumulate Function	428
	Fetch and Op Function	429
	Compare and Swap Function	431
	11.3.5 Request-based RMA Communication Operations	432
11 4	Memory Model	437
	Synchronization Calls	438
11.0	11.5.1 Fence	442
	11.5.2 General Active Target Synchronization	443
	11.5.3 Lock	447
		450
	11.5.4 Flush and Sync	
	11.5.5 Assertions	452
11.0	11.5.6 Miscellaneous Clarifications	454
11.6	Error Handling	454
	11.6.1 Error Handlers	454
	11.6.2 Error Classes	454
11.7	Semantics and Correctness	455
	11.7.1 Atomicity	463
	11.7.2 Ordering	463
	11.7.3 Progress	464
	11.7.4 Registers and Compiler Optimizations	466
11.8	Examples	466
10 5 4		
	ernal Interfaces	477
	Introduction	477
12.2	Ceneralized Requests	477
10.0	12.2.1 Examples	482
	Associating Information with Status	484
12.4	MPI and Threads	486
		100
	12.4.1 General	486
	12.4.2 Clarifications	$480 \\ 487$
19 I/O	12.4.2 Clarifications       12.4.3 Initialization         12.4.3 Initialization       12.4.3 Initialization	487 489
<b>13 I/O</b>	12.4.2 Clarifications       12.4.3 Initialization         12.4.3 Initialization       12.4.3 Initialization	487 489 <b>493</b>
	12.4.2 Clarifications       12.4.3 Initialization         12.4.3 Initialization       12.4.3 Initialization         Introduction       12.4.3 Initialization	487 489 <b>493</b> 493
13.1	12.4.2 Clarifications       12.4.3 Initialization         12.4.3 Initialization       12.4.3 Initialization         Introduction       13.1.1 Definitions	487 489 <b>493</b> 493 493
13.1	12.4.2 Clarifications	487 489 <b>493</b> 493 493 495
13.1	12.4.2 Clarifications         12.4.3 Initialization         Introduction         13.1.1 Definitions         File Manipulation         13.2.1 Opening a File	487 489 <b>493</b> 493 493 495 495
13.1	12.4.2 Clarifications         12.4.3 Initialization         Introduction         13.1.1 Definitions         File Manipulation         13.2.1 Opening a File         13.2.2 Closing a File	487 489 <b>493</b> 493 493 495 495 497
13.1	12.4.2 Clarifications         12.4.3 Initialization         Introduction         13.1.1 Definitions         File Manipulation         13.2.1 Opening a File         13.2.2 Closing a File         13.2.3 Deleting a File	487 489 <b>493</b> 493 493 495 495 497 498
13.1	12.4.2 Clarifications12.4.3 Initialization12.4.3 InitializationIntroduction13.1.1 Definitions13.1.1 DefinitionsFile Manipulation13.2.1 Opening a File13.2.2 Closing a File13.2.3 Deleting a File13.2.4 Resizing a File	487 489 <b>493</b> 493 493 495 495 495 497 498 499
13.1	12.4.2 Clarifications12.4.3 Initialization12.4.3 Initialization13.1.1 Definitions13.1.1 Definitions2 File Manipulation13.2.1 Opening a File13.2.2 Closing a File13.2.3 Deleting a File13.2.4 Resizing a File13.2.5 Preallocating Space for a File	487 489 <b>493</b> 493 493 495 495 495 497 498 499 500
13.1	12.4.2 Clarifications12.4.3 Initialization12.4.3 Initialization13.1.1 Definitions13.1.1 Definitions13.1.1 Definitions13.2.1 Opening a File13.2.2 Closing a File13.2.3 Deleting a File13.2.4 Resizing a File13.2.5 Preallocating Space for a File13.2.6 Querying the Size of a File	487 489 <b>493</b> 493 493 495 495 495 497 498 499 500 500
13.1	12.4.2 Clarifications12.4.3 Initialization12.4.3 Initialization13.1.1 Definitions13.1.1 Definitions2 File Manipulation13.2.1 Opening a File13.2.2 Closing a File13.2.3 Deleting a File13.2.4 Resizing a File13.2.5 Preallocating Space for a File	487 489 <b>493</b> 493 493 495 495 495 497 498 499 500

		Reserved File Hints	 	 			•			504
13.3	File Vi	iews	 	 						505
13.4	Data A	Access	 	 						508
	13.4.1	Data Access Routines	 	 						508
		Positioning	 	 						509
		Synchronism	 	 						510
		Coordination								510
		Data Access Conventions								510
	13.4.2	Data Access with Explicit Offsets								511
		Data Access with Individual File Pointers .								516
		Data Access with Shared File Pointers								524
		Noncollective Operations								524
		Collective Operations								527
		Seek								528
	1345	Split Collective Data Access Routines								529
13.5		teroperability								536
10.0		Datatypes for File Interoperability								538
		External Data Representation: "external32"								540
		User-Defined Data Representations								540 541
	10.0.0	Extent Callback								543
										543 543
	19 5 4	Datarep Conversion Functions								
19.6		Matching Data Representations								546
13.0		tency and Semantics								546
		File Consistency								546
		Random Access vs. Sequential Files								549
		Progress								550
		Collective File Operations								550
		Nonblocking Collective File Operations								550
		Type Matching								551
		Miscellaneous Clarifications								551
		MPI_Offset Type								551
		Logical vs. Physical File Layout								552
		)File Size								552
	13.6.11	Examples								552
		Asynchronous I/O								555
		rror Handling								557
13.8	I/O Ei	rror Classes	 	 					•	557
13.9		$\mathbf{bles}$								557
	13.9.1	Double Buffering with Split Collective I/O	 	 					•	557
	13.9.2	Subarray Filetype Constructor	 	 					•	560
 	. ~									
	l Supp									563
		uction								563
14.2		ng Interface								563
		Requirements								563
		Discussion								564
		Logic of the Design								564
	14.2.4	Miscellaneous Control of Profiling	 • •	 	•	• •	•	•	•	565

	14.2.5	Profiler Implementation Example				566
	14.2.6	MPI Library Implementation Example				566
		Systems with Weak Symbols				566
		Systems Without Weak Symbols				
	14.2.7	Complications				
		Multiple Counting				567
		Linker Oddities				568
		Fortran Support Methods				568
	1428	Multiple Levels of Interception				
14.3		Pl Tool Information Interface				569
11.0		Verbosity Levels				570
		Binding MPI Tool Information Interface Variables to MPI				
		Convention for Returning Strings				
		Initialization and Finalization				572
						572
		Datatype System				
	14.3.0	Control Variables				
		Control Variable Query Functions				
		Example: Printing All Control Variables				
		Handle Allocation and Deallocation				579
		Control Variable Access Functions				
		Example: Reading the Value of a Control Variable				581
	14.3.7	Performance Variables				582
		Performance Variable Classes				
		Performance Variable Query Functions				584
		Performance Experiment Sessions				587
		Handle Allocation and Deallocation				587
		Starting and Stopping of Performance Variables				589
		Performance Variable Access Functions				590
		Example: Tool to Detect Receives with Long Unexpected	d M	less	age	
		Queues				592
	14.3.8	Variable Categorization				594
	14.3.9	Return Codes for the MPI Tool Information Interface				598
	14.3.10	Profiling Interface				598
		-				
		nult Tolerance				601
15.1	Introd	$\operatorname{uction}$				601
15.2	Failure	Notification				602
	15.2.1	Startup and Finalize				602
	15.2.2	Point-to-Point and Collective Communication				603
	15.2.3	Dynamic Process Management				604
	15.2.4	One-Sided Communication				605
	15.2.5	I/O				606
15.3		$\stackrel{\prime}{{ m Mitigation}}$ Functions				607
		Communicator Functions				607
		One-Sided Functions				
		I/O Functions				612
$15 \ 4$		s Failure Error Codes and Classes				
		bles				
10.0			• •	• •	• •	010

15.5.1 Safe Communicator Creat	on
15.5.2 Obtaining the consistent g	roup of failed processes 614
15.5.3 Fault-Tolerant Master/We	rker
15.5.4 Fault-Tolerant Iterative R	efinement 616
16 Deprecated Functions	619
16.1 Deprecated since MPI-2.0 $\ldots$	
16.2 Deprecated since MPI-2.2 $\ldots$	
17 Removed Interfaces	623
<u> </u>	
	623
	$Prototypes \dots \dots$
$17.2 \text{ C}++ \text{ Bindings } \dots \dots \dots$	
18 Language Bindings	625
* *	
	the mpi_f08 Module 626
•••	the mpi Module 629
•••	the mpif.h Include File 631
	cocedure Names, and the Profiling Interface 632
	Standard Versions637Compilers641
· · · · · · · · · · · · · · · · · · ·	Compilers641stran Register-Memory-Synchronization642
	tran Numeric Intrinsic Types 643
	vith Specified Precision and Exponent Range644
	IPI Datatypes
	-specific Types
	ndings for MPI
	yping
	by and Sequence Association with Sub-
18.1.13 Problems Due to Data Cop	ying and Sequence Association with Vector
$\operatorname{Subscripts}$	
18.1.14 Special Constants	
	n Overview
	ment and Register Optimization 660
	bining Independent Variables in Datatypes 661
The Fortran ASYNCHRO	NOUS Attribute

		Calling MPI_F_SYNC_REG	664
		A User Defined Routine Instead of MPI_F_SYNC_REG	665
		Module Variables and COMMON Blocks	666
		The (Poorly Performing) Fortran VOLATILE Attribute	666
		The Fortran TARGET Attribute	
		18.1.18 Temporary Data Movement and Temporary Memory Modification .	
		18.1.19 Permanent Data Movement	
		18.1.20 Comparison with C	
	18.2	Language Interoperability	
		18.2.1 Introduction	
		18.2.2 Assumptions	
		18.2.3 Initialization	
		18.2.4 Transfer of Handles	
		18.2.5 Status	
		18.2.6 MPI Opaque Objects	
		Datatypes	
		Callback Functions	
		Error Handlers	
		Reduce Operations	
		18.2.7 Attributes	
		18.2.8 Extra-State	
		18.2.9 Constants	
		18.2.10 Interlanguage Communication	
Α	Lan	guage Bindings Summary	689
	A.1	Defined Values and Handles	689
		A.1.1 Defined Constants	689
		A.1.2 Types	702
		A.1.3 Prototype Definitions	704
		C Bindings	704
		Fortran 2008 Bindings with the mpi_f08 Module	704
		Fortran Bindings with mpif.h or the mpi Module	
		A.1.4 Deprecated Prototype Definitions	709
		A.1.5 Info Keys	710
		A.1.6 Info Values	710
	A.2	C Bindings	712
	A.3	Fortran 2008 Bindings with the mpi_f08 Module	713
	A.4	Fortran Bindings with mpif.h or the mpi Module	714
Β	Cha	inge-Log	715
	B.1	Changes from Version 3.1 to Version 4.0	
		B.1.1 Changes from Version 3.1 to Version 4.0	
	B.2	Changes from Version 3.0 to Version 3.1	
		B.2.1 Fixes to Errata in Previous Versions of MPI	
		B.2.2 Changes in MPI-3.1	
	B.3	Changes from Version 2.2 to Version 3.0	
		B.3.1 Fixes to Errata in Previous Versions of MPI	
		B.3.2 Changes in MPI-3.0	719

B.4 Changes from Version 2.1 to Version 2.2	724
B.5 Changes from Version 2.0 to Version 2.1	726
Bibliography	731
General Index	736
Examples Index	740
MPI Constant and Predefined Handle Index	743
MPI Declarations Index	748
MPI Callback Function Prototype Index	749
MPI Function Index	750

# List of Figures

5.1	Collective communications, an overview	145
5.2	Intercommunicator allgather	148
5.3	Intercommunicator reduce-scatter	149
5.4	Gather example	155
5.5	Gatherv example with strides	156
5.6	Gatherv example, 2-dimensional	157
5.7	Gatherv example, 2-dimensional, subarrays with different sizes	158
5.8	Gatherv example, 2-dimensional, subarrays with different sizes and strides .	160
5.9	Scatter example	165
5.10	Scattery example with strides	165
5.11	Scattery example with different strides and counts	166
5.12	Race conditions with point-to-point and collective communications	219
5.13	Overlapping Communicators Example	223
6.1	Intercommunicator creation using MPI_COMM_CREATE	244
6.2	Intercommunicator construction with MPI_COMM_SPLIT	248
6.3	Three-group pipeline	265
6.4	Three-group ring	266
7.1	Neighborh and method and institution and and	318
7.1 7.2	Neighborhood gather communication example	332
		<u>აა</u> 2
7.3	Communication routine with local data copying and sparse neighborhood	333
7.4	all-to-all	ააა
1.4	local data copying.	334
	local data copying.	<b>3</b> 34
11.1	Schematic description of the public/private window operations in the	
	MPI_WIN_SEPARATE memory model for two overlapping windows.	438
11.2	Active target communication	440
11.3	Active target communication, with weak synchronization	441
	Passive target communication	442
	Active target communication with several processes	446
	Symmetric communication	464
	Deadlock situation	465
	No deadlock	465
11.0		100
13.1	Etypes and filetypes	494
13.2	Partitioning a file among parallel processes	494
13.3	Displacements	507

13.4	Example array file layout	560
13.5	Example local array filetype for process 1	561
18.1	Status conversion routines	377

# List of Tables

2.1	Deprecated and Removed constructs	18
3.1 3.2 3.3 3.4	Predefined MPI datatypes corresponding to Fortran datatypes Predefined MPI datatypes corresponding to C datatypes Predefined MPI datatypes corresponding to both C and Fortran datatypes . Predefined MPI datatypes corresponding to C++ datatypes	27 28 29 29
4.1	combiner values returned from MPI_TYPE_GET_ENVELOPE	119
6.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode)	261
8.1 8.2	Error classes (Part 1)	$351 \\ 352$
	C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR	$\begin{array}{c} 416\\ 454 \end{array}$
13.2	Data access routines	$509 \\ 542 \\ 558$
$14.2 \\ 14.3 \\ 14.4$	MPI tool information interface verbosity levels	570 571 573 577 599
15.1	Additional process fault tolerance error classes	613
17.2 17.3 17.4 18.1	Removed MPI-1 functions and their replacements	623 624 624 624 633 659

## Acknowledgments

6

 $\overline{7}$ 

8

9 10 11

12

13

14

18 19

20 21

22 23

 $^{24}$ 

25 26

27

28 29

30 31

32 33

34

35

36 37

38 39

40

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

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

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

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

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communication
- Al Geist, Marc Snir, Steve Otto, Collective Communication
  - Steve Otto, Editor
- Rolf Hempel, Process Topologies
- Ewing Lusk, Language Binding
- William Gropp, Environmental Management
- James Cownie, Profiling
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators
- Steven Huss-Lederman, Initial Implementation Subset

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

Unofficial Draft for Comment Only

Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz	1
Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster	2
Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower	3
Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison	4
Leslie Hart	Tom Haupt	Don Heller	Tom Henderson	5
Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga	6
James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe	7
Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley	8
Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer	9
Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison	10
Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson	11
Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith	12
				13
-		-	atory made the draft avail-	14
			distributing the document.	15
			RPA and NSF under grant	16
			nology Center Cooperative	17
-		mmission of the Eur	opean Community through	18
Esprit project P6643 (PI	PPE).			19
				20
MPI-1.2 and MPI-2.0:				21
Those who served as prin	mary goordinators i	MDI 1.2 and MDI	2.0 are:	22
Those who served as prin	mary coordinators in	n wir i-1.2 and wir i-	2.0 are.	23 24
• Ewing Lusk, Conve	ener and Meeting Cl	hair		24 25
	Ū			26
• Steve Huss-Lederm	an, Editor			27
• Ewing Lusk, Miscellany				
• Lwing Lusk, Wiscenary				
• Bill Saphir, Process Creation and Management				
• Marc Snir, One-Sided Communications				31
	• Marc Shir, One-Sided Communications			
• Bill Gropp and Ant	thony Skjellum, Ext	ended Collective Op	perations	33
• Steve Huss-Lederm	on Extornal Interfe	0.005		34
• Steve Huss-Lederm	an, External interna	ices		35
• Bill Nitzberg, I/O				36
. A., J.,	D:11 Caralia and I	- ff C	Diss liss	37 38
• Andrew Lumsdaine	e, Bill Saphir, and J	eff Squyres, Langua	ge Bindings	39
• Anthony Skjellum a	and Arkady Kanevs	ky, Real-Time		40
				41
0		active participants w	who attended $MPI-2$ Forum	42
meetings and are not me	entioned above.			43
				44
				45
				46
				47
				48

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

Intel Corporation	1
International Business Machines	2
Khoral Research	3
Lawrence Livermore National Laboratory	4
Los Alamos National Laboratory	5
MPI Software Techology, Inc.	6
Mississippi State University	7
NEC Corporation	8
National Aeronautics and Space Administration	9
National Energy Research Scientific Computing Center	10
National Institute of Standards and Technology	11
National Oceanic and Atmospheric Adminstration	12
Oak Ridge National Laboratory	13
The Ohio State University	14
PALLAS GmbH	15
Pacific Northwest National Laboratory	16
Pratt & Whitney	17
San Diego Supercomputer Center	18
Sanders, A Lockheed-Martin Company	19
Sandia National Laboratories	20
Schlumberger	21
Scientific Computing Associates, Inc.	22
Silicon Graphics Incorporated	23
Sky Computers	24
Sun Microsystems Computer Corporation	25
Syracuse University	26
The MITRE Corporation	27
Thinking Machines Corporation	28
United States Navy	29
University of Colorado	30
University of Denver	31
University of Houston	32
University of Illinois	33
University of Maryland	34
University of Notre Dame	35
University of San Fransisco	36
University of Stuttgart Computing Center	37
University of Wisconsin	38
MPL2 operated on a very tight hudget (in reality, it had no hudget when the first	39

MPI-2 operated on a very tight budget (in reality, it had no budget when the first meeting was announced). Many institutions helped the MPI-2 effort by supporting the efforts and travel of the members of the MPI Forum. Direct support was given by NSF and DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and Esprit under project HPC Standards (21111) for European participants.

 $45 \\ 46$ 

40

 $^{41}$ 

42

43

MPI-1.3 and MPI-2.1:
The editors and organizers of the combined documents have been:
• Richard Graham, Convener and Meeting Chair
• Jack Dongarra, Steering Committee
• Al Geist, Steering Committee
• Bill Gropp, Steering Committee
• Rainer Keller, Merge of MPI-1.3
• Andrew Lumsdaine, Steering Committee
• Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)
• Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)
All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:
• Bill Gropp, Front matter, Introduction, and Bibliography
• Richard Graham, Point-to-Point Communication
• Adam Moody, Collective Communication
• Richard Treumann, Groups, Contexts, and Communicators
• Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communica- tions
• George Bosilca, Environmental Management
• David Solt, Process Creation and Management
• Bronis R. de Supinski, External Interfaces, and Profiling
• Rajeev Thakur, I/O
• Jeffrey M. Squyres, Language Bindings and MPI-2.1 Secretary
• Rolf Rabenseifner, Deprecated Functions and Annex Change-Log
• Alexander Supalov and Denis Nagorny, Annex Language Bindings
The following list includes some of the active participants who attended MPI-2 Forum meetings and in the e-mail discussions of the errata items and are not mentioned above.

	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett	1	
	Richard Barrett	Christian Bell	Robert Blackmore	2	
	Gil Bloch	Ron Brightwell	Jeffrey Brown	3	
	Darius Buntinas	Jonathan Carter	Nathan DeBardeleben	4	
	Terry Dontje	Gabor Dozsa	Edric Ellis	5	
	Karl Feind	Edgar Gabriel	Patrick Geoffray	6	
	David Gingold	Dave Goodell	Erez Haba	7	
	Robert Harrison	Thomas Herault	Steve Hodson	8	
	Torsten Hoefler	Joshua Hursey	Yann Kalemkarian	9	
	Matthew Koop	Quincey Koziol	Sameer Kumar	10	
	Miron Livny	Kannan Narasimhan	Mark Pagel	11	
	Avneesh Pant	Steve Poole	Howard Pritchard	12	
	Craig Rasmussen	Hubert Ritzdorf	Rob Ross	13	
	Tony Skjellum	Brian Smith	Vinod Tipparaju	14	
	Jesper Larsson Träff	Keith Underwood	vinou ripparaju	15	
	Jesper Laisson fran	Ketti Ulderwood		16	
Tł	ne MPI Forum also ackr	nowledges and appreciates the	valuable input from people via	10	
	and in person.			18	
	-	s supported the MPI-2 effort the	hrough time and travel support	19	
	people listed above.			20	
				20	
	gonne National Labora	atory		22	
Bi Ci				22	
	sco Systems, Inc.				
	Cray Inc.				
	The HDF Group <sup>22</sup> Herelett Packard <sup>24</sup>				
	newlett-1 ackalu				
	iDivi 1.5. Watson Research				
	Indiana University Institut National de Recherche en Informatique et Automatique (INRIA)				
	institut National de Recherche en informatique et Automatique (INRIA)				
				30	
	wrence Berkeley Nation	0		31	
	wrence Livermore Nati	0		32	
	os Alamos National Lab	boratory		33	
	athworks			34	
	ellanox Technologies			35	
	icrosoft			36	
	yricom			37	
NI	EC Laboratories Europ	e, NEC Europe Ltd.		38	
Oa	ak Ridge National Labo	oratory		39	
Tł	ne Ohio State Universit	y		40	
Pa	acific Northwest Nation	al Laboratory		41	
QI	Logic Corporation			42	
Sa	ndia National Laborate	ories		43	
Si	Cortex			44	
Sil	licon Graphics Incorpor	rated		45	
Su	in Microsystems, Inc.			46	
Ur	niversity of Alabama at	Birmingham		47	
Ur	niversity of Houston			48	

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

	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett	1	
	Richard Barrett	Christian Bell	Robert Blackmore	2	
	Gil Bloch	Ron Brightwell	Greg Bronevetsky	3	
	Jeff Brown	Darius Buntinas	Jonathan Carter	4	
	Nathan DeBardeleben	Terry Dontje	Gabor Dozsa	5	
	Edric Ellis	Karl Feind	Edgar Gabriel	6	
	Patrick Geoffray	Johann George	David Gingold	7	
	David Goodell	Erez Haba	Robert Harrison	8	
	Thomas Herault	Marc-André Hermanns	Steve Hodson	9	
	Joshua Hursey	Yutaka Ishikawa	Bin Jia	10	
	Hideyuki Jitsumoto	Terry Jones	Yann Kalemkarian	11	
	Ranier Keller	Matthew Koop	Quincey Koziol	12	
	Manojkumar Krishnan	Sameer Kumar	Miron Livny	13	
	Andrew Lumsdaine	Miao Luo	Ewing Lusk	14	
	Timothy I. Mattox	Kannan Narasimhan	Mark Pagel	15	
	Avneesh Pant	Steve Poole	Howard Pritchard	16	
	Craig Rasmussen	Hubert Ritzdorf	Rob Ross	17	
	Martin Schulz	Pavel Shamis	Galen Shipman	18	
	Christian Siebert	Anthony Skjellum	Brian Smith	19	
	Naoki Sueyasu	Vinod Tipparaju	Keith Underwood	20	
	Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu	21	
(T)		1 1 1 1 1		22	
		vledges and appreciates the va	luable input from people via	23	
	nd in person.			24	
	_	pported the MPI-2.2 effort three	bugh time and travel support	25	
for the p	people listed above.			26	
Arg	onne National Laborator	ſy		27	
Aub	ourn University			28	
Bul	1			29	
Cise	Cisco Systems, Inc.				
Cra	Cray Inc.				
Fors	Forschungszentrum Jülich <sup>33</sup>				
Fuji	itsu			33	
The	e HDF Group			34	
Hew	vlett-Packard			35	
Inte	ernational Business Mach	nines		36	
Indi	iana University			37	
Inst	itut National de Rechero	che en Informatique et Autom	atique (INRIA)	38	
Inst	itute for Advanced Scier	nce & Engineering Corporation	n	39	
Inte	el Corporation			40	
Law	vrence Berkeley National	Laboratory		41	
Law	vrence Livermore Nationa	al Laboratory		42	
	Alamos National Labora	atory		43	
Mat	thworks			44	
Mel	lanox Technologies			45	
Mic	rosoft			46	
Myı	ricom			47	
NEO	C Corporation			48	

1	Oak Ridge National Laboratory
2	The Ohio State University
3	Pacific Northwest National Laboratory
4	QLogic Corporation
5	RunTime Computing Solutions, LLC
6	Sandia National Laboratories
7	SiCortex, Inc.
8	Silicon Graphics Inc.
9	Sun Microsystems, Inc.
10	Tokyo Institute of Technology
11	University of Alabama at Birmingham
12	University of Houston
13	University of Illinois at Urbana-Champaign
14	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
15	University of Tennessee, Knoxville
16	University of Tokyo
17	University of Wisconsin
18	Funding for the MPI Forum meetings was partially supported by awards $\#$ CCF-0816909
19	and $\#CCF-1144042$ from the National Science Foundation. In addition, the HDF Group
20	provided travel support for one U.S. academic. $\prod$ addition, the HDT croup
21	
22 23	MPI-3.0:
24	MPI-3.0 is a significant effort to extend and modernize the MPI Standard.
25	The editors and organizers of the MPI-3.0 have been:
26	
27	• William Gropp, Steering committee, Front matter, Introduction, Groups, Contexts,
28	and Communicators, One-Sided Communications, and Bibliography
29	• Richard Graham, Steering committee, Point-to-Point Communication, Meeting Con-
30	vener, and MPI-3.0 chair
31	voner; and wir 1-5.0 chan
32	• Torsten Hoefler, Collective Communication, One-Sided Communications, and Process
33	Topologies
34	
35	• George Bosilca, Datatypes and Environmental Management
36	• David Solt, Process Creation and Management
37	
38	• Bronis R. de Supinski, External Interfaces and Tool Support
39 40	• Rajeev Thakur, I/O and One-Sided Communications
40	• Rajeev Haadai, 1/ o and one sided communications
42	• Darius Buntinas, Info Object
43 44	• Jeffrey M. Squyres, Language Bindings and MPI-3.0 Secretary
45	• Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindings,
46	Deprecated Functions, Annex Change-Log, and Annex Language Bindings
47	
48	• Craig Rasmussen, Fortran Bindings

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.

Tatsuya Abe	Tomoya Adachi	Sadaf Alam	3 4
Reinhold Bader	Pavan Balaji	Purushotham V. Bangalore	4 5
Brian Barrett	Richard Barrett	Robert Blackmore	6
Aurelien Bouteiller	Ron Brightwell	Greg Bronevetsky	7
Jed Brown	Darius Buntinas	Devendar Bureddy	8
Arno Candel	George Carr	Mohamad Chaarawi	9
Raghunath Raja Chandraseka	0	Terry Dontje	10
Edgar Gabriel	Balazs Gerofi	Brice Goglin	11
David Goodell	Manjunath Gorentla	Erez Haba	12
Jeff Hammond	Thomas Herault	Marc-André Hermanns	13
Jennifer Herrett-Skjellum	Nathan Hjelm	Atsushi Hori	14
Joshua Hursey	Marty Itzkowitz	Yutaka Ishikawa	15
Nysal Jan	Bin Jia	Hideyuki Jitsumoto	16
Yann Kalemkarian	Krishna Kandalla	Takahiro Kawashima	17
Chulho Kim	Dries Kimpe	Christof Klausecker	18
Alice Koniges	Quincey Koziol	Dieter Kranzlmueller	19
Manojkumar Krishnan	Sameer Kumar	Eric Lantz	20
Jay Lofstead	Bill Long	Andrew Lumsdaine	21
Miao Luo	Ewing Lusk	Adam Moody	22
Nick M. Maclaren	Amith Mamidala	Guillaume Mercier	23
Scott McMillan	Douglas Miller	Kathryn Mohror	24
Tim Murray	Tomotake Nakamura	Takeshi Nanri	25
Steve Oyanagi	Mark Pagel	Swann Perarnau	26
Sreeram Potluri	Howard Pritchard	Rolf Riesen	27
Hubert Ritzdorf	Kuninobu Sasaki	Timo Schneider	28
Martin Schulz	Gilad Shainer	Christian Siebert	29
Anthony Skjellum	Brian Smith	Marc Snir	30
Raffaele Giuseppe Solca	Shinji Sumimoto	Alexander Supalov	31
Sayantan Sur	Masamichi Takagi	Fabian Tillier	32
Vinod Tipparaju	Jesper Larsson Träff	Richard Treumann	33
Keith Underwood	Rolf Vandevaart	Anh Vo	34
Abhinav Vishnu	Min Xie	Enqiang Zhou	35

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.

4
4
4
4
4

36

37

38

39

40

41

42

 $1 \\ 2$ 

1	ETH Zurich
2	Fujitsu Ltd.
3	German Research School for Simulation Sciences
4	The HDF Group
5	Hewlett-Packard
6	International Business Machines
7	IBM India Private Ltd
8	Indiana University
9	Institut National de Recherche en Informatique et Automatique (INRIA)
10	Institute for Advanced Science & Engineering Corporation
11	Intel Corporation
12	Lawrence Berkeley National Laboratory
13	Lawrence Livermore National Laboratory
14	Los Alamos National Laboratory
15	Mellanox Technologies, Inc.
16	Microsoft Corporation
17	NEC Corporation
18	National Oceanic and Atmospheric Administration, Global Systems Division
19	NVIDIA Corporation
20	Oak Ridge National Laboratory
21	The Ohio State University
22	Oracle America
23	Platform Computing
24	RIKEN AICS
25	RunTime Computing Solutions, LLC
26	Sandia National Laboratories
27	Technical University of Chemnitz
28	Tokyo Institute of Technology
29	University of Alabama at Birmingham
30	University of Chicago
31	University of Houston
32	University of Illinois at Urbana-Champaign
33	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
34	University of Tennessee, Knoxville
35	University of Tokyo
36	Funding for the MPI Forum meetings was partially supported by awards $\#CCF$ 0816000
37	Funding for the MPI Forum meetings was partially supported by awards $\#CCF-0816909$
38	and #CCF-1144042 from the National Science Foundation. In addition, the HDF Group and Sandia National Laboratoria provided travel support for and U.S. academic cash
39	and Sandia National Laboratories provided travel support for one U.S. academic each.
40	
41	MPI-3.1:
42	MPI-3.1 is a minor update to the MPI Standard.
43	The editors and organizers of the MPI-3.1 have been:
44	
45	• Martin Schulz, MPI-3.1 chair
46	• William Gropp, Steering committee, Front matter, Introduction, One-Sided Commu-
47	• Winnam Gropp, Steering committee, Front matter, introduction, One-Sided Commu- nications, and Bibliography; Overall editor
48	meanons, and Dibliography, Overan eutor

• Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindings,	
Deprecated Functions, Annex Change-Log, and Annex Language Bindings	
· Disbard I. Craham Stanning committee Meeting Convener	3
• Richard L. Graham, Steering committee, Meeting Convener	4
• Jeffrey M. Squyres, Language Bindings and MPI-3.1 Secretary	5
• Daniel Holmes, Point-to-Point Communication	6 7
• George Bosilca, Datatypes and Environmental Management	8 9
• Torsten Hoefler, Collective Communication and Process Topologies	10 11
• Pavan Balaji, Groups, Contexts, and Communicators, and External Interfaces	12
• Jeff Hammond, The Info Object	$13 \\ 14$
• David Solt, Process Creation and Management	15
• Quincey Koziol, I/O	16 17
	18
• Kathryn Mohror, Tool Support	19
• Rajeev Thakur, One-Sided Communications	
	21
The following list includes some of the active participants who attended MPI Forum	22

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

mgs or partic	ipatea in th	c c man anoabbionb.		
				24
Charles Arc	her	Pavan Balaji	Purushotham V. Bangalore	25
Brian Barre	tt	Wesley Bland	Michael Blocksome	26
George Bosi	lca	Aurelien Bouteiller	Devendar Bureddy	27
Yohann Bur	ette	Mohamad Chaarawi	Alexey Cheptsov	28
James Dina	n	Dmitry Durnov	Thomas Francois	29
Edgar Gabr	iel	Todd Gamblin	Balazs Gerofi	30
Paddy Gillie	es	David Goodell	Manjunath Gorentla Venkata	31
Richard L.	Graham	Ryan E. Grant	William Gropp	32
Khaled Han	nidouche	Jeff Hammond	Amin Hassani	33
Marc-André	Hermanns	Nathan Hjelm	Torsten Hoefler	34
Daniel Holn	nes	Atsushi Hori	Yutaka Ishikawa	35
Hideyuki Jit	tsumoto	Jithin Jose	Krishna Kandalla	36
Christos Ka	vouklis	Takahiro Kawashima	Chulho Kim	37
Michael Kno	obloch	Alice Koniges	Quincey Koziol	38
Sameer Kur	nar	Joshua Ladd	Ignacio Laguna	39
Huiwei Lu		Guillaume Mercier	Kathryn Mohror	40
Adam Mood	ły	Tomotake Nakamura	Takeshi Nanri	41
Steve Oyana	agi	Antonio J. Pẽna	Sreeram Potluri	42
Howard Prit	tchard	Rolf Rabenseifner	Nicholas Radcliffe	43
Ken Raffene	etti	Raghunath Raja	Craig Rasmussen	44
Davide Ross	setti	Kento Sato	Martin Schulz	45
Sangmin Se	0	Christian Siebert	Anthony Skjellum	46
Brian Smith	l	David Solt	Jeffrey M. Squyres	47

48

1	Hari Subramoni Shinji Sumimoto Alexander Supalov
2	Bronis R. de Supinski Sayantan Sur Masamichi Takagi
3	Keita Teranishi Rajeev Thakur Fabian Tillier
4	Yuichi Tsujita Geoffroy Vallée Rolf vandeVaart
5	Akshay Venkatesh Jerome Vienne Venkat Vishwanath
6	Anh Vo Huseyin S. Yildiz Junchao Zhang
7	Xin Zhao
8	The MPI Forum also acknowledges and appreciates the valuable input from people via
9	e-mail and in person.
10	The following institutions supported the MPI-3.1 effort through time and travel support
11	for the people listed above.
12	
13	Argonne National Laboratory
14	Auburn University
15	Cisco Systems, Inc.
16	Cray EBCC The University of Ediphyreth
17 18	EPCC, The University of Edinburgh ETH Zurich
19	Forschungszentrum Jülich
20	Fujitsu
21	German Research School for Simulation Sciences
22	The HDF Group
23	International Business Machines
24	INRIA
25	Intel Corporation
26	Jülich Aachen Research Alliance, High-Performance Computing (JARA-HPC)
27	Kyushu University
28	Lawrence Berkeley National Laboratory
29	Lawrence Livermore National Laboratory
30	Lenovo
31	Los Alamos National Laboratory
32	Mellanox Technologies, Inc.
33	Microsoft Corporation
34	NEC Corporation
35	NVIDIA Corporation
36	Oak Ridge National Laboratory
37	The Ohio State University
38	RIKEN AICS
39	Sandia National Laboratories
40	Texas Advanced Computing Center
41	Tokyo Institute of Technology
42	University of Alabama at Birmingham
43	University of Houston
44	University of Illinois at Urbana-Champaign
45	University of Oregon
46	University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
47	University of Tennessee, Knoxville
48	University of Tokyo

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

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

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

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

- 43
- $45 \\ 46$

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

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

#### Unofficial Draft for Comment Only

3 4 5

6

7

8 9 10

11 12

13

14

15

16

17

1

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

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

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

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

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

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

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

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

#### Unofficial Draft for Comment Only

1 2

3

4

5 6

7

8 9

10 11

12

13

14 15

16

17

18

19

20

21

22

23

24 25

26

27

28

29

30

31 32

33

34 35

36

37

38 39

40

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

 $15 \\ 16$ 

17 18

19

20

21 22

23

24 25

26

27 28

29

30 31

32 33

34

35

36

37

#### 1.5 Background of MPI-2.2

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

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

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

#### 1.6 Background of MPI-3.0

MPI-3.0 is a major update to the MPI standard. The updates include the extension of collective operations to include nonblocking versions, extensions to the one-sided operations, and a new Fortran 2008 binding. In addition, the deprecated C++ bindings have been removed, as well as many of the deprecated routines and MPI objects (such as the MPI\_UB datatype).

38 39 40

41

#### 1.7 Background of MPI-3.1

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

47

# 1.8 Who Should Use This Standard?

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

# 1.9 What Platforms Are Targets For Implementation?

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

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

# 1.10 What Is Included In The Standard?

The standard includes:

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

## Unofficial Draft for Comment Only

 $^{24}$ 

• One-sided communication, • External interfaces. • Parallel file I/O, • Language bindings for Fortran and C, • Tool support. What Is Not Included In The Standard? 1.11 The standard does not specify: • Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,

- Program construction tools,
- Debugging facilities.

There are many features that have been considered and not included in this standard. This happened for a number of reasons, one of which is the time constraint that was selfimposed in finishing the standard. Features that are not included can always be offered as extensions by specific implementations. Perhaps future versions of MPI will address some of these issues.

#### Organization of this Document 1.12

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

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

# **Unofficial Draft for Comment Only**

6

1

 $\mathbf{2}$ 

3 4

56

7

8 9 10

11 12

13

14

1516

1718

19 20

21

22

23

 $^{24}$ 

2526

2728

29

30  $^{31}$ 

32

33 34

35 36

37

38

39 40

41

42

43

44

45

- Chapter 7, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 8, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.
- Chapter 9, The Info Object, defines an opaque object, that is used as input in several MPI routines.
- Chapter 10, Process Creation and Management, defines routines that allow for creation of processes.
- Chapter 11, One-Sided Communications, defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations.
- Chapter 12, External Interfaces, defines routines designed to allow developers to layer on top of MPI. This includes generalized requests, routines that decode MPI opaque objects, and threads.
- Chapter 13, I/O, defines MPI support for parallel I/O.
- Chapter 14, Tool Support, covers interfaces that allow debuggers, performance analyzers, and other tools to obtain data about the operation of MPI processes. This chapter includes Section 14.2 (Profiling Interface), which was a chapter in previous versions of MPI.
- Chapter 15, Process Fault Tolerance, covers interfaces that allow developers to design applications tolerant to process failures. The interfaces presented in this chapter define the state of the MPI library after a process crash, and provide supplementary interfaces to restore the communication capabilities of MPI.
- Chapter 16, Deprecated Functions, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
- Chapter 17, Removed Interfaces, describes routines and constructs that have been removed from MPI. These were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard.
- Chapter 18, Language Bindings, discusses Fortran issues, and describes language interoperability aspects between C and Fortran.

The Appendices are:

- Annex A, Language Bindings Summary, gives specific syntax in C and Fortran, for all MPI functions, constants, and types.
- Annex B, Change-Log, summarizes some changes since the previous version of the standard.

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

 $46 \\ 47$ 

• Several Index pages show the locations of examples, constants and predefined handles, callback routine prototypes, and all MPI functions.

MPI provides various interfaces to facilitate interoperability of distinct MPI implementations. Among these are the canonical data representation for MPI I/O and for MPI\_PACK\_EXTERNAL and MPI\_UNPACK\_EXTERNAL. The definition of an actual bind-ing of these interfaces that will enable interoperability is outside the scope of this document. A separate document consists of ideas that were discussed in the MPI Forum during the MPI-2 development and deemed to have value, but are not included in the MPI Standard. They are part of the "Journal of Development" (JOD), lest good ideas be lost and in order to provide a starting point for further work. The chapters in the JOD are 

- Chapter 2, Spawning Independent Processes, includes some elements of dynamic process management, in particular management of processes with which the spawning processes do not intend to communicate, that the Forum discussed at length but ultimately decided not to include in the MPI Standard.
  - Chapter 3, Threads and MPI, describes some of the expected interaction between an MPI implementation and a thread library in a multi-threaded environment.
    - Chapter 4, Communicator ID, describes an approach to providing identifiers for communicators.
  - Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particular single-copy routines for use in shared-memory environments and new datatype constructors.
    - Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.
  - Chapter 7, Split Collective Communication, describes a specification for certain nonblocking collective operations.
  - Chapter 8, Real-Time MPI, discusses MPI support for real time processing.

 $\mathbf{2}$ 

# Chapter 2

# **MPI** Terms and Conventions

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

# 2.1 Document Notation

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

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

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

# 2.2 Naming Conventions

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

- 1. In C, all routines associated with a particular type of MPI object should be of the form MPI\_Class\_action\_subset or, if no subset exists, of the form MPI\_Class\_action. In Fortran, all routines associated with a particular type of MPI object should be of the form MPI\_CLASS\_ACTION\_SUBSET or, if no subset exists, of the form MPI\_CLASS\_ACTION.
- 2. If the routine is not associated with a class, the name should be of the form MPI\_Action\_subset in C and MPI\_ACTION\_SUBSET in Fortran.

 $45 \\ 46$ 

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

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

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

# 2.3 Procedure Specification

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

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

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

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

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

28 29

30

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

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

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

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

<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 12

13 14

15

16

17

18 19

20 21

<pre>void copyIntBuffer( int *pin, int *pout, int len ) {    int i;</pre>	1 2
for (i=0; i <len; *pout++="*pin++;&lt;/td" ++i)=""><td>3</td></len;>	3
}	4
	5
then a call to it in the following code fragment has aliased arguments.	6
int a[10];	7
copyIntBuffer( a, a+3, 7);	8
copyincballer( a, a, b, 7),	9
Although the C language allows this, such usage of MPI procedures is forbidden unless	10
otherwise specified. Note that Fortran prohibits aliasing of arguments.	11
All MPI functions are first specified in the language-independent notation. Immediately	12
below this, language dependent bindings follow:	13
• The ISO C version of the function.	14
• The 150 C version of the function.	15
• The Fortran version used with USE mpi_f08.	16 17
• The Fortran version of the same function used with USE mpi or INCLUDE 'mpif.h'.	18
An exception is Section 14.3 "The MPI Tool Information Interface", which only provides	19
ISO C interfaces.	20
"Fortran" in this document refers to Fortran 90 and higher; see Section 2.6.	21 22
	22
0.4 Compartie Terres	24
2.4 Semantic Terms	25
When discussing MPI procedures the following semantic terms are used.	26
when discussing with procedures the following semantic terms are used.	27
<b>nonblocking</b> A procedure is nonblocking if it may return before the associated operation	28
completes, and before the user is allowed to reuse resources (such as buffers) specified	29
in the call. The word complete is used with respect to operations and any associated	30
requests and/or communications. An <b>operation completes</b> when the user is allowed	31
to reuse resources, and any output buffers have been updated.	32
	33
blocking A procedure is blocking if return from the procedure indicates the user is allowed	34
to reuse resources specified in the call.	35
<b>local</b> A procedure is local if completion of the procedure depends only on the local executing	36
process.	37
	38
non-local A procedure is non-local if completion of the operation may require the exe-	39
cution of some MPI procedure on another process. Such an operation may require	40
communication occurring with another user process.	41
collective A procedure is collective if all processes in a process group need to invoke the	42
procedure. A collective call may or may not be synchronizing. Collective calls over	43
the same communicator must be executed in the same order by all members of the	44
process group.	45
L COLORE	46
	47
	48

	12	CHAPTER 2. MPI TERMS AND CONVENTIONS
1 2 3 4 5 6	pred	<b>defined</b> 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 <b>named</b> whereas the latter are <b>unnamed</b> .
7	deriv	<b>ved</b> A derived datatype is any datatype that is not predefined.
8 9 10 11 12 13 14 15 16 17 18 19 20 21 20 21 22 23 23 24	port	able A datatype is portable if it is a predefined datatype, or it is derived from a portable datatype using only the type constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY, MPI_TYPE_DUP, and MPI_TYPE_CREATE_DARRAY. Such a datatype is portable because all displacements in the datatype are in terms of extents of one predefined datatype. Therefore, if such a datatype fits a data layout in one memory, it will fit the corresponding data layout in another memory, if the same declarations were used, even if the two systems have different architectures. On the other hand, if a datatype was constructed using MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, MPI_TYPE_CREATE_HVECTOR or MPI_TYPE_CREATE_STRUCT, then the datatype contains explicit byte displace- 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 used for data layouts on another process, running on a processor with a different architecture.
25 26 27 28	equi	valent 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.
29	2.5	Data Types
	∠.J	

- 2.5 Data Types
- 2.5.1 Opaque Objects

<sup>33</sup> MPI manages **system memory** that is used for buffering messages and for storing internal <sup>34</sup> representations of various MPI objects such as groups, communicators, datatypes, etc. This <sup>35</sup> memory is not directly accessible to the user, and objects stored there are **opaque**: their <sup>36</sup> size and shape is not visible to the user. Opaque objects are accessed via **handles**, which <sup>37</sup> exist in user space. MPI procedures that operate on opaque objects are passed handle <sup>38</sup> arguments to access these objects. In addition to their use by MPI calls for object access, <sup>39</sup> 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:

48

30 31

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

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

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

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

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

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

A call to a deallocate routine invalidates the handle and marks the object for deallocation. The object is not accessible to the user after the call. However, MPI need not deallocate the object immediately. Any operation pending (at the time of the deallocate) that involves this object will complete normally; the object will be deallocated afterwards.

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

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

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

The explicit separation of handles in user space and objects in system space allows space-reclaiming and deallocation calls to be made at appropriate points in the user

### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

 $46 \\ 47$ 

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

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. The alternative 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 then copies of its components; a call to MPI\_COMM\_GROUP may return a reference to the group associated with the communicator, rather than a copy of this group. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects in such a way that the visible effect is as if the objects were copied. (End of advice to implementors.)

<sup>33</sup> 2.5.2

Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of 35 handles. The array-of-handles is a regular array with entries that are handles to objects 36 of the same type in consecutive locations in the array. Whenever such an array is used, 37 an additional len argument is required to indicate the number of valid entries (unless this 38 number can be derived otherwise). The valid entries are at the beginning of the array; 39 len indicates how many of them there are, and need not be the size of the entire array. 40 The same approach is followed for other array arguments. In some cases NULL handles are 41 considered valid entries. When a NULL argument is desired for an array of statuses, one 42uses MPI\_STATUSES\_IGNORE. 43

 $44 \\ 45$ 

## 2.5.3 State

 $^{46}_{47}$  MPI procedures use at various places arguments with *state* types. The values of such a data type are all identified by names, and no operation is defined on them. For example, the

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

MPI\_TYPE\_CREATE\_SUBARRAY routine has a state argument order with values MPI\_ORDER\_C and MPI\_ORDER\_FORTRAN.

### 2.5.4 Named Constants

MPI procedures sometimes assign a special meaning to a special value of a basic type argument; e.g., tag is an integer-valued argument of point-to-point communication operations, with a special wild-card value, MPI\_ANY\_TAG. Such arguments will have a range of regular values, which is a proper subrange of the range of values of the corresponding basic type; special values (such as MPI\_ANY\_TAG) will be outside the regular range. The range of regular values, such as tag, can be queried using environmental inquiry functions, see Chapter 8. The range of other values, such as source, depends on values given by other MPI routines (in the case of source it is the communicator size).

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

All named constants, with the exceptions noted below for Fortran, can be used in 15initialization expressions or assignments, but not necessarily in array declarations or as 16 labels in C switch or Fortran select/case statements. This implies named constants 17 to be link-time but not necessarily compile-time constants. The named constants listed 18 below are required to be compile-time constants in both C and Fortran. These constants 19 do not change values during execution. Opaque objects accessed by constant handles are 20defined and do not change value between MPI initialization (MPI\_INIT) and MPI completion 21(MPI\_FINALIZE). The handles themselves are constants and can be also used in initialization 22expressions or assignments. 23

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

ar	e:	26
	MPI_MAX_PROCESSOR_NAME	27
	MPI_MAX_LIBRARY_VERSION_STRING	28
	MPI_MAX_ERROR_STRING	29
	MPI_MAX_DATAREP_STRING	30
	MPI_MAX_INFO_KEY	31
	MPI_MAX_INFO_VAL	32
	MPI_MAX_OBJECT_NAME	33
	MPI_MAX_PORT_NAME	34
	MPI_VERSION	35
	MPI_SUBVERSION	36
	MPI_STATUS_SIZE (Fortran only)	37
	MPI_ADDRESS_KIND (Fortran only)	38
	MPI_COUNT_KIND (Fortran only)	39
	MPI_INTEGER_KIND (Fortran only)	40
	MPI_OFFSET_KIND (Fortran only)	41
	MPI_SUBARRAYS_SUPPORTED (Fortran only)	42
	MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)	43
	The constants that cannot be used in initialization expressions or assignments in For-	44
$\operatorname{tr}$	an are as follows:	45
	MPI_BOTTOM	46
	MPI_STATUS_IGNORE	47
	MPI_STATUSES_IGNORE	48
		10

Unofficial Draft for Comment Only

1 2

3 4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

24

1	MPI_ERRCODES_IGNORE
2	MPI_IN_PLACE
3	MPI_ARGV_NULL
4	MPI_ARGVS_NULL
5	MPI_UNWEIGHTED
6	MPI_WEIGHTS_EMPTY

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

2.5.5 Choice

<sup>19</sup> MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to <sup>20</sup> the same routine may pass by reference actual arguments of different types. The mechanism <sup>21</sup> for providing such arguments will differ from language to language. For Fortran with the <sup>22</sup> include file mpif.h or the mpi module, the document uses <type> to represent a choice <sup>23</sup> variable; with the Fortran mpi\_f08 module, such arguments are declared with the Fortran <sup>24</sup> 2008 + TR 29113 syntax TYPE(\*), DIMENSION(..); for C, we use void \*.

- Advice to implementors. Implementors can freely choose how to implement choice arguments in the mpi module, e.g., with a non-standard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the mpi\_f08 module. See details in Section 18.1.1. (End of advice to implementors.)
- 31 32 2.5

2.5.6 Absolute Addresses and Relative Address Displacements

Some MPI procedures use *address* arguments that represent an *absolute address* in the calling program, or *relative displacement* arguments that represent differences of two absolute addresses. The datatype of such arguments is MPI\_Aint in C and INTEGER (KIND=

36 MPI\_ADDRESS\_KIND) in Fortran. These types must have the same width and encode address 37 values in the same manner such that address values in one language may be passed directly 38 to another language without conversion. There is the MPI constant MPI\_BOTTOM to in-39 dicate the start of the address range. For retrieving absolute addresses or any calculation 40 with absolute addresses, one should use the routines and functions provided in Section 4.1.5. 41 Section 4.1.12 provides additional rules for the correct use of absolute addresses. For ex-42pressions with relative displacements or other usage without absolute addresses, intrinsic 43 operators (e.g., +, -, \*) can be used. 44

44 45

46

# 2.5.7 File Offsets

<sup>47</sup> For I/O there is a need to give the size, displacement, and offset into a file. These quantities <sup>48</sup> can easily be larger than 32 bits which can be the default size of a Fortran integer. To

# Unofficial Draft for Comment Only

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16 17

18

25

26

27

28

29

overcome this, these quantities are declared to be INTEGER (KIND=MPI\_OFFSET\_KIND) in Fortran. In C one uses MPI\_Offset. These types must have the same width and encode address values in the same manner such that offset values in one language may be passed directly to another language without conversion.

#### 2.5.8 Counts

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

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

# 2.6 Language Binding

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

MPI bindings are for Fortran 90 or later, though they were originally designed to be usable in Fortran 77 environments. With the mpi\_f08 module, two new Fortran features, assumed type and assumed rank, are also required, see Section 2.5.5.

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

Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid any prefix of the form "MPI\_" and "PMPI\_", where any of the letters are either upper or lower case.

#### 2.6.1 Deprecated and Removed Names and Functions

A number of chapters refer to deprecated or replaced MPI constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 16, but that users are recommended not to continue using, since better solutions were provided with newer

#### 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

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.

<sup>8</sup> Some of the deprecated constructs are now removed, as documented in Chapter 17.
 <sup>9</sup> 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.

4	Deprecated or removed	deprecated	removed	Replacement
5	construct	since	since	heplacement
6	MPI_ADDRESS	MPI-2.0	MPI-3.0	MPI_GET_ADDRESS
7	MPI_TYPE_HINDEXED	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HINDEXED
8	MPI_TYPE_HVECTOR	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HVECTOR
9	MPI_TYPE_STRUCT	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_STRUCT
	MPI_TYPE_EXTENT	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
C	MPI_TYPE_UB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
	MPI_TYPE_LB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
	MPI_LB <sup>1</sup>	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
	MPI_UB <sup>1</sup>	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
	MPI_ERRHANDLER_CREATE	MPI-2.0	MPI-3.0	MPI_COMM_CREATE_ERRHANDLER
	MPI_ERRHANDLER_GET	MPI-2.0	MPI-3.0	MPI_COMM_GET_ERRHANDLER
	MPI_ERRHANDLER_SET	MPI-2.0	MPI-3.0	MPI_COMM_SET_ERRHANDLER
	MPI_Handler_function <sup>2</sup>	MPI-2.0	MPI-3.0	$MPI\_Comm\_errhandler\_function^2$
	MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL
	MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL
	MPI_DUP_FN <sup>3</sup>	MPI-2.0		MPI_COMM_DUP_FN <sup>3</sup>
	MPI_NULL_COPY_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_COPY_FN <sup>3</sup>
	MPI_NULL_DELETE_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_DELETE_FN <sup>3</sup>
	MPI_Copy_function <sup>2</sup>	MPI-2.0		$MPI\_Comm\_copy\_attr\_function^2$
	COPY_FUNCTION <sup>3</sup>	MPI-2.0		COMM_COPY_ATTR_FUNCTION <sup>3</sup>
	$MPI_Delete_function^2$	MPI-2.0		$MPI\_Comm\_delete\_attr\_function^2$
	DELETE_FUNCTION <sup>3</sup>	MPI-2.0		COMM_DELETE_ATTR_FUNCTION <sup>3</sup>
	MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR
	MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR
	MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR
	MPI_COMBINER_HVECTOR_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_HVECTOR <sup>4</sup>
	MPI_COMBINER_HINDEXED_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_HINDEXED <sup>4</sup>
	MPI_COMBINER_STRUCT_INTEGER <sup>4</sup>	-	MPI-3.0	$MPI_COMBINER_STRUCT^4$
	MPI:	MPI-2.2	MPI-3.0	C language binding
	<sup>1</sup> Predefined datatype.			
	<sup>2</sup> Callback prototype definition.			
	<sup>3</sup> Predefined callback routine.			
	<sup>4</sup> Constant.			
	Other entries are regular MPI routines.			
			1 D	1
;	Table 2.1: $D$	eprecated a	nd Remov	ed constructs

#### 2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term "Fortran" is used it means Fortran 90 or later; it means Fortran 2008 + TR 29113 and later if the mpi\_f08 module is used.

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

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

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

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

Handles are represented in Fortran as INTEGERS, or as a BIND(C) derived type with the mpi\_f08 module; see Section 2.5.1. Binary-valued variables are of type LOGICAL.

Array arguments are indexed from one.

The older MPI Fortran bindings (mpif.h and use mpi) are inconsistent with the Fortran standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 18.1.16.

#### 2.6.3 C Binding Issues

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

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

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

Type declarations are provided for handles to each category of opaque objects. Array arguments are indexed from zero.

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

Choice arguments are pointers of type void \*.

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

### 20

1

2

3

4

5

6 7

8

9 10

11

12 13

14

# 2.6.4 Functions and Macros

An implementation is allowed to implement MPI\_WTIME, PMPI\_WTIME, MPI\_WTICK, PMPI\_WTICK, MPI\_AINT\_ADD, PMPI\_AINT\_ADD, MPI\_AINT\_DIFF, PMPI\_AINT\_DIFF, and the handle-conversion functions (MPI\_Group\_f2c, etc.) in Section 18.2.4, and no others, as macros in C.

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

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

# 2.7 Processes

<sup>15</sup> An MPI program consists of autonomous processes, executing their own code, in an MIMD <sup>16</sup> style. The codes executed by each process need not be identical. The processes communicate <sup>17</sup> via calls to MPI communication primitives. Typically, each process executes in its own <sup>18</sup> address space, although shared-memory implementations of MPI are possible.

19This document specifies the behavior of a parallel program assuming that only MPI 20calls are used. The interaction of an MPI program with other possible means of commu-21nication, I/O, and process management is not specified. Unless otherwise stated in the 22specification of the standard, MPI places no requirements on the result of its interaction 23with external mechanisms that provide similar or equivalent functionality. This includes,  $^{24}$ but is not limited to, interactions with external mechanisms for process control, shared and 25remote memory access, file system access and control, interprocess communication, process 26signaling, and terminal I/O. High quality implementations should strive to make the results 27of such interactions intuitive to users, and attempt to document restrictions where deemed 28necessary.

29 30 31

32

33

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

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

34 35 36

37

# 2.8 Error Handling

<sup>38</sup> MPI provides the user with reliable message transmission. A message sent is always received <sup>39</sup> correctly, and the user does not need to check for transmission errors, time-outs, or other <sup>40</sup> error conditions. In other words, MPI does not provide mechanisms for dealing with failures <sup>41</sup> in the communication system. If the MPI implementation is built on an unreliable underly-<sup>42</sup> ing mechanism, then it is the job of the implementor of the MPI subsystem to insulate the <sup>43</sup> user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, <sup>44</sup> such failures will be reflected as errors in the relevant communication call.

The mechanisms for handling process failures are defined in Chapter 15. When a process failure happens, the MPI implementation may raise one of the MPI error classes related to process failure as defined in that chapter. In this case, the MPI implementation is still in a defined state and continues to operate.

# Unofficial Draft for Comment Only

Advice to users. MPI does not provide the user with transparent process recovery upon process failure. In particular, restoring the lost dataset, spawning spare processes or taking other recovery actions are the responsibility of the user.

It is possible that some error (e.g. memory corruption) may cause the state of MPI to become corrupted. The behavior of MPI in this case is undefined, and it is possible that the implementation returns incorrect error codes (including MPI\_SUCCESS). While a high-quality implementation will strive to always return correct return codes from MPI operations, it may not be possible in all cases. (End of advice to users.)

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 1920of the operation. Whenever possible, MPI calls return an error code if an error occurred 21during 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 23specify that no error is fatal, and handle error codes returned by MPI calls by himself or herself. Also, the user may provide his or her own error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 8.3.

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

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

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

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

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

1

2

3

4

5

6

 $\overline{7}$ 

8

9 10

11

12

13

14

15

16

17

18

 $^{24}$ 

25

26

2728

29

30

 $^{31}$ 

42

43

44

4546

47

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

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

# 2.9 Implementation Issues

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

14 15 16

25

43

5

 $\frac{6}{7}$ 

8 9

10

11

12

13

## 2.9.1 Independence of Basic Runtime Routines

<sup>17</sup> MPI programs require that library routines that are part of the basic language environment
 <sup>18</sup> (such as write in Fortran and printf and malloc in ISO C) and are executed after
 <sup>19</sup> MPI\_INIT and before MPI\_FINALIZE operate independently and that their completion is

<sup>13</sup> MPI\_INIT and before MPI\_FINALIZE operate independently and that their *completion* is <sup>20</sup> independent of the action of other processes in an MPI program.

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

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

<sup>31</sup> The corresponding Fortran programs are also expected to complete.

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

```
<sup>37</sup> MPI_Comm_rank(MPI_COMM_WORLD, &rank);
<sup>38</sup> printf("Output from task rank %d\n", rank);
<sup>39</sup>
```

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

44 2.9.2 Interaction with Signals

<sup>45</sup> MPI does not specify the interaction of processes with signals and does not require that MPI <sup>46</sup> be signal safe. The implementation may reserve some signals for its own use. It is required <sup>47</sup> that the implementation document which signals it uses, and it is strongly recommended

## Unofficial Draft for Comment Only

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

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

# 2.10 Examples

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

# Chapter 3

# **Point-to-Point Communication**

#### 3.1Introduction

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

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

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

7

9

1516

17

18

19

24

25

26

38

40

41

46

47

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

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

#### 3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

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

```
38
     MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
39
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
40
         INTEGER, INTENT(IN) :: count, dest, tag
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
         INTEGER, OPTIONAL, INTENT(OUT) ::
43
                                             ierror
44
```

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

```
47
```

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

1718

19 20

21

2223 $^{24}$ 

35

36

#### 3.2.2 Message Data

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

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

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

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

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

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

MPI requires support of these datatypes, which match the basic datatypes of Fortran and ISO C. Additional MPI datatypes should be provided if the host language has additional data types: MPI\_DOUBLE\_COMPLEX for double precision complex in Fortran declared to be of type DOUBLE COMPLEX; MPI\_REAL2, MPI\_REAL4, and MPI\_REAL8 for Fortran reals, declared to be of type REAL\*2, REAL\*4 and REAL\*8, respectively; MPI\_INTEGER1, MPI\_INTEGER2, and MPI\_INTEGER4 for Fortran integers, declared to be of type INTEGER\*1, INTEGER\*2, and INTEGER\*4, respectively; etc.

*Rationale.* One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit argument. The need for such datatype information will become clear in Section 3.3.2. (*End of rationale.*)

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

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{31}$ 

1	MPI datatype	C datatype
2	MPI_CHAR	char
3		(treated as printable character)
4	MPI_SHORT	signed short int
5	MPI_INT	signed int
6	MPI_LONG	signed long int
7	MPI_LONG_LONG_INT	signed long long int
8	MPI_LONG_LONG (as a synonym)	signed long long int
9	MPI_SIGNED_CHAR	signed char
10		(treated as integral value)
11	MPI_UNSIGNED_CHAR	unsigned char
12		(treated as integral value)
13	MPI_UNSIGNED_SHORT	unsigned short int
14	MPI_UNSIGNED	unsigned int
15	MPI_UNSIGNED_LONG	unsigned long int
16	MPI_UNSIGNED_LONG_LONG	unsigned long long int
17	MPI_FLOAT	float
18	MPI_DOUBLE	double
19	MPI_LONG_DOUBLE	long double
20	MPI_WCHAR	wchar_t
21		(defined in <stddef.h>)</stddef.h>
22		(treated as printable character)
23	MPI_C_BOOL	_Bool
24	MPI_INT8_T	int8_t
25	MPI_INT16_T	int16_t
26	MPI_INT32_T	int32_t
27	MPI_INT64_T	int64_t
28	MPI_UINT8_T	uint8_t
29	MPI_UINT16_T	uint16_t
30	MPI_UINT32_T	uint32_t
31	MPI_UINT64_T	uint64_t
32	MPI_C_COMPLEX	float _Complex
33	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
34	MPI_C_DOUBLE_COMPLEX	double _Complex
35	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
36	MPI_BYTE	
37	MPI_PACKED	
38		1
39		
40	Table 3.2: Predefined $MPI$ datatypes co	prresponding to C datatypes
41		
42 INT	EGER (KIND=MPI_ADDRESS_KIND), INTEGER (I	KIND=MPI_OFFSET_KIND), and IM

INTEGER (KIND=MPI\_ADDRESS\_KIND), INTEGER (KIND=MPI\_OFFSET\_KIND), and INTEGER
 (KIND=MPI\_COUNT\_KIND). This is described in Table 3.3. All predefined datatype handles
 are available in all language bindings. See Sections 18.2.6 and 18.2.10 on page 678 and 686
 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.

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

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

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

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

#### 3.2.3 Message Envelope

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

source
destination
$\operatorname{tag}$
communicator

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

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

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

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

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

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

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

#### Unofficial Draft for Comment Only

1 2

5 6

15 16 17

18

19

20

27

28 29

30

 $^{31}$ 

32 33

34

35 36

37

38 39

40 41

42

43 44

1 Advice to users. Users that are comfortable with the notion of a flat name space 2 for processes, and a single communication context, as offered by most existing com-3 munication libraries, need only use the predefined variable MPI\_COMM\_WORLD as the 4 comm argument. This will allow communication with all the processes available at 5initialization time. 6 Users may define new communicators, as explained in Chapter 6. Communicators 7 provide an important encapsulation mechanism for libraries and modules. They allow 8 modules to have their own disjoint communication universe and their own process 9 numbering scheme. (End of advice to users.) 10 11 Advice to implementors. The message envelope would normally be encoded by a 12fixed-length message header. However, the actual encoding is implementation depen-13 dent. Some of the information (e.g., source or destination) may be implicit, and need 14not be explicitly carried by messages. Also, processes may be identified by relative 15ranks, or absolute ids, etc. (End of advice to implementors.) 16173.2.4 **Blocking Receive** 18 19The syntax of the blocking receive operation is given below. 2021MPI\_RECV (buf, count, datatype, source, tag, comm, status) 22 OUT 23buf initial address of receive buffer (choice)  $^{24}$ IN count number of elements in receive buffer (non-negative in-25teger) 26IN datatype datatype of each receive buffer element (handle) 2728IN rank of source or MPI\_ANY\_SOURCE (integer) source 29 IN message tag or MPI\_ANY\_TAG (integer) tag 30 IN comm communicator (handle)  $^{31}$ 32 OUT status status object (Status) 33 34int MPI\_Recv(void\* buf, int count, MPI\_Datatype datatype, int source, 35 int tag, MPI\_Comm comm, MPI\_Status \*status) 36 MPI\_Recv(buf, count, datatype, source, tag, comm, status, ierror) 37 TYPE(\*), DIMENSION(..) :: buf 38 INTEGER, INTENT(IN) :: count, source, tag 39 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 40 TYPE(MPI\_Comm), INTENT(IN) :: comm 41 TYPE(MPI\_Status) :: status 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44MPI\_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) 45<type> BUF(\*) 46INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), 47IERROR 48

#### 3.2. BLOCKING SEND AND RECEIVE OPERATIONS

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

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

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

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

Advice to implementors. Even though no specific behavior is mandated by MPI for erroneous programs, the recommended handling of overflow situations is to return in status information about the source and tag of the incoming message. The receive operation will return an error code. A quality implementation will also ensure that no memory that is outside the receive buffer will ever be overwritten.

In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow for some optimizations but this is not allowed. The implementation must be ready to end a copy into the receiver memory exactly at the end of the receive buffer, even if it is an odd address. (End of advice to implementors.)

The selection of a message by a receive operation is governed by the value of the 23message envelope. A message can be received by a receive operation if its envelope matches  $^{24}$ 25the source, tag and comm values specified by the receive operation. The receiver may 26specify a wildcard MPI\_ANY\_SOURCE value for source, and/or a wildcard MPI\_ANY\_TAG value for tag, indicating that any source and/or tag are acceptable. It cannot specify a 2728 wildcard value for comm. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching communicator, has matching 2930 source unless source=MPI\_ANY\_SOURCE in the pattern, and has a matching tag unless  $^{31}$ tag=MPI\_ANY\_TAG in the pattern.

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

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

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

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

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

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9

10 11

12

13

14

15

1617

18

19

20

21

22

32

37

38

39

40 41

42

43

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

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

3.2.5 Return Status

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

In C, status is a structure that contains three fields named MPI\_SOURCE, MPI\_TAG, and MPI\_ERROR; the structure may contain additional fields. Thus,

status.MPI\_SOURCE, status.MPI\_TAG and status.MPI\_ERROR contain the source, tag, and
 error code, respectively, of the received message.

In Fortran with USE mpi or INCLUDE 'mpif.h', status is an array of INTEGERs of size MPI\_STATUS\_SIZE. The constants MPI\_SOURCE, MPI\_TAG and MPI\_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI\_SOURCE), status(MPI\_TAG) and status(MPI\_ERROR) contain, respectively, the source, tag and error code of the received message.

With Fortran USE mpi\_f08, status is defined as the Fortran BIND(C) derived type TYPE(MPI\_Status) containing three public INTEGER fields named MPI\_SOURCE, MPI\_TAG, and MPI\_ERROR. TYPE(MPI\_Status) may contain additional, implementation-specific fields. Thus, status%MPI\_SOURCE, status%MPI\_TAG and status%MPI\_ERROR contain the source, tag, and error code of a received message respectively. Additionally, within both the mpi and the mpi\_f08 modules, the constants MPI\_STATUS\_SIZE, MPI\_SOURCE, MPI\_TAG,

<sup>30</sup> MPI\_ERROR, and TYPE(MPI\_Status) are defined to allow conversion between both status <sup>31</sup> representations. Conversion routines are provided in Section 18.2.5.

- *Rationale.* The Fortran TYPE(MPI\_Status) is defined as a BIND(C) derived type so that it can be used at any location where the status integer array representation can be used, e.g., in user defined common blocks. (*End of rationale.*)
  - *Rationale.* It is allowed to have the same name (e.g., MPI\_SOURCE) defined as a constant (e.g., Fortran parameter) and as a field of a derived type. (*End of rationale.*)

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

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

## Unofficial Draft for Comment Only

2 3 4

5 6 7

8

32

33

34

35 36

37

38

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

MPI\_GET\_COUNT(status, datatype, count)

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

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

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

Returns the number of entries received. (Again, we count *entries*, each of type *datatype*, not *bytes*.) The datatype argument should match the argument provided by the receive call that set the status variable. If the number of entries received exceeds the limits of the count parameter, then MPI\_GET\_COUNT sets the value of count to MPI\_UNDEFINED. There are other situations where the value of count can be set to MPI\_UNDEFINED; see Section 4.1.11.

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

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

The value returned as the count argument of MPI\_GET\_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI\_UNDEFINED is returned.

*Rationale.* Zero-length datatypes may be created in a number of cases. An important case is MPI\_TYPE\_CREATE\_DARRAY, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style

 $^{31}$ 

 $\frac{44}{45}$ 

will not check for this special case and may want to use MPI\_GET\_COUNT to check the status. (End of rationale.)

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

All send and receive operations use the buf, count, datatype, source, dest, tag, comm, and status arguments in the same way as the blocking MPI\_SEND and MPI\_RECV operations 10 described in this section. 11

1213

1

2

3

4

5

6

7 8

9

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

14Every call to MPI\_RECV includes a status argument, wherein the system can return details 15about the message received. There are also a number of other MPI calls where status 16is returned. An object of type MPI\_Status is not an MPI opaque object; its structure 17is declared in mpi.h and mpif.h, and it exists in the user's program. In many cases, 18 application programs are constructed so that it is unnecessary for them to examine the 19 status fields. In these cases, it is a waste for the user to allocate a status object, and it is 20particularly wasteful for the MPI implementation to fill in fields in this object. 21

To cope with this problem, there are two predefined constants, MPI\_STATUS\_IGNORE 22 and MPI\_STATUSES\_IGNORE, which when passed to a receive, probe, wait, or test function, 23inform the implementation that the status fields are not to be filled in. Note that  $^{24}$ 

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

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

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

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

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

- 47
- 48

## 3.3 Data Type Matching and Data Conversion

### 3.3.1 Type Matching Rules

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

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

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

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

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

 $^{24}$ The type of a variable in a host program matches the type specified in the commu-25nication operation if the datatype name used by that operation corresponds to the basic 26type of the host program variable. For example, an entry with type name MPI\_INTEGER matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran 2728and C appears in Section 3.2.2. There are two exceptions to this last rule: an entry with type name MPI\_BYTE or MPI\_PACKED can be used to match any byte of storage (on a 2930 byte-addressable machine), irrespective of the datatype of the variable that contains this  $^{31}$ byte. The type MPI\_PACKED is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 4.2. The type MPI\_BYTE allows 32 33 one to transfer the binary value of a byte in memory unchanged.

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

- Communication of typed values (e.g., with datatype different from MPI\_BYTE), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.
- Communication of untyped values (e.g., of datatype MPI\_BYTE), where both sender and receiver use the datatype MPI\_BYTE. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.
- Communication involving packed data, where MPI\_PACKED is used.

The following examples illustrate the first two cases.

**Example 3.1** Sender and receiver specify matching types.

#### Unofficial Draft for Comment Only

5

6 7

8 9

10 11

12

13

14 15

16

17

18

19 20

21

22

23

34

35

36

37

38 39

40

41

42

43

44 45

46 47

48

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

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

## Unofficial Draft for Comment Only

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

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

*Rationale.* The alternative choice would be for MPI\_CHARACTER to match a character of arbitrary length. This runs into problems.

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

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

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

#### 3.3.2 Data Conversion

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

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

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

The type matching rules imply that MPI communication never entails type conversion. <sup>47</sup> On the other hand, MPI requires that a representation conversion be performed when a <sup>48</sup>

#### Unofficial Draft for Comment Only

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

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

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

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

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

26

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

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

<sup>34</sup> Data representation conversion also applies to the envelope of a message: source, des-<sup>35</sup> tination and tag are all integers that may need to be converted.

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

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

44 45

36

37

38

39

40

41 42

43

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

# 3.4 Communication Modes

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

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

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

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

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

There are three additional communication modes.

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

A send that uses the **synchronous** mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its 44 45 46 47 48

#### Unofficial Draft for Comment Only

2

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23 24

25

26

27

28

29

30

31

32

33 34

35

36

37

38

39

40

41

42

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

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

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

19 20

21

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

22IN buf initial address of send buffer (choice) 23IN count number of elements in send buffer (non-negative inte-24ger) 25datatype of each send buffer element (handle) IN datatype 2627dest IN rank of destination (integer) 28IN message tag (integer) tag 29 IN comm communicator (handle) 30  $^{31}$ int MPI\_Bsend(const void\* buf, int count, MPI\_Datatype datatype, int dest, 32 33 int tag, MPI\_Comm comm) 34MPI\_Bsend(buf, count, datatype, dest, tag, comm, ierror) 35 TYPE(\*), DIMENSION(...), INTENT(IN) :: buf 36 INTEGER, INTENT(IN) :: count, dest, tag 37 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 38 TYPE(MPI\_Comm), INTENT(IN) :: comm 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 41 MPI\_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 42<type> BUF(\*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 43 44 Send in buffered mode. 45 46 4748

MPI_SS	END (buf, count, datatype	e, dest, tag, comm)	1		
IN	buf	initial address of send buffer (choice)	2		
IN	count	number of elements in send buffer (non-negative integer)	3 4 5		
IN	datatype	datatype of each send buffer element (handle)	6		
IN	dest	rank of destination (integer)	7		
IN	tag	message tag (integer)	8		
	-		9 10		
IN	comm	communicator (handle)	11		
int MPI	_Ssend(const void* bu int tag, MPI_C	<pre>if, int count, MPI_Datatype datatype, int dest, omm comm)</pre>	12 13 14		
MPI_Sse	nd(buf, count, dataty	ype, dest, tag, comm, ierror)	15		
	E(*), DIMENSION(),		16		
	EGER, INTENT(IN) ::	-	17		
	E(MPI_Datatype), INT E(MPI_Comm), INTENT(1		18		
	EGER, OPTIONAL, INTEN		19 20		
			21		
	mpe> BUF(*)	YPE, DEST, TAG, COMM, IERROR)	22		
•	-	, DEST, TAG, COMM, IERROR	23		
			24		
Sen	d in synchronous mode.		25 26		
			20		
MPI_RS	END (buf, count, datatyp	e, dest, tag, comm)	28		
IN	buf	initial address of send buffer (choice)	29		
IN	count	number of elements in send buffer (non-negative integer)	30 31		
IN	datatype	datatype of each send buffer element (handle)	32 33		
IN	dest	rank of destination (integer)	34		
IN	tag	message tag (integer)	35		
	-	,	36		
IN	comm	communicator (handle)	37		
int MDT	Beend (const woidt h	if int count MPI Datature datature int dest	38		
	<pre>int MPI_Rsend(const void* buf, int count, MPI_Datatype datatype, int dest,</pre>				
MDT D	<u> </u>		41		
	E(*), DIMENSION(),	ype, dest, tag, comm, ierror)	42		
	EGER, INTENT(IN) ::		43		
	E(MPI_Datatype), INTH		44		
	E(MPI_Comm), INTENT(		45		
INT	EGER, OPTIONAL, INTEN	NT(OUT) :: ierror	46 47		
MPI_RSE	MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 48				
	TITI_RSEND(BOF, COUNT, DATATIFE, DEST, TAG, COPPI, TERROR) 48				

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

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

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

Example 3.5 An example of non-overtaking messages.

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

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

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

An example of two, intertwined matching pairs. Example 3.6

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
                                                                                42
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
    CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

1

 $\mathbf{2}$ 

3

4

56

 $\overline{7}$ 

8 9

10

11

12

13

1415

16

17

18

1920

21

22

23

 $^{24}$ 

25

26

27

2829

30

3132

33

34

35

36

37 38

> 39 40

41

43

44

45

46

47

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

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

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

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

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

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

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

47 48

```
1
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                       2
IF (rank.EQ.0) THEN
                                                                                       3
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
                                                                                       4
ELSE IF (rank.EQ.1) THEN
                                                                                       5
                                                                                       6
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
                                                                                       8
                                                                                       9
This program will succeed even if no buffer space for data is available. The standard send
                                                                                       10
operation can be replaced, in this example, with a synchronous send.
                                                                                       11
                                                                                       12
Example 3.8
               An errant attempt to exchange messages.
                                                                                       13
                                                                                       14
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                       15
IF (rank.EQ.0) THEN
                                                                                       16
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
                                                                                       17
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
                                                                                       18
ELSE IF (rank.EQ.1) THEN
                                                                                       19
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
                                                                                       20
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
                                                                                       21
END IF
                                                                                       22
The receive operation of the first process must complete before its send, and can complete
                                                                                       23
only if the matching send of the second processor is executed. The receive operation of the
                                                                                       24
second process must complete before its send and can complete only if the matching send
                                                                                       25
of the first process is executed. This program will always deadlock. The same holds for any
                                                                                       26
other send mode.
                                                                                       27
                                                                                       28
Example 3.9
               An exchange that relies on buffering.
                                                                                       29
CALL MPI_COMM_RANK(comm, rank, ierr)
                                                                                       30
IF (rank.EQ.0) THEN
                                                                                       31
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
                                                                                       32
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
                                                                                       33
ELSE IF (rank.EQ.1) THEN
                                                                                       34
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
                                                                                       35
```

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

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

Advice to users.When standard send operations are used, then a deadlock situation43may occur where both processes are blocked because buffer space is not available.44same will certainly happen, if the synchronous mode is used.1f the buffered mode isused, and not enough buffer space is available, then the program will not complete46either.However, rather than a deadlock situation, we shall have a buffer overflow474848

### Unofficial Draft for Comment Only

36

37 38

39

40

41

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

Many programmers prefer to have more leeway and opt to use the "unsafe" programming style shown in Example 3.9. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that "common practice" programs will 10 not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode 12might also be used for debugging purposes, as buffer overflow conditions are easier to 13 diagnose than deadlock conditions. 14

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

19 2021

22 23

 $^{24}$ 

2526

2728

29

30  $^{31}$ 

32 33

34

35

36

37

38

39

40

#### 3.6 Buffer Allocation and Usage

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

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

```
IN
           buffer
                                      initial buffer address (choice)
  IN
           size
                                      buffer size, in bytes (non-negative integer)
int MPI_Buffer_attach(void* buffer, int size)
MPI_Buffer_attach(buffer, size, ierror)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                           ierror
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR
    Provides to MPI a buffer in the user's memory to be used for buffering outgoing mes-
```

41 42sages. The buffer is used only by messages sent in buffered mode. Only one buffer can be 43 attached to a process at a time. In C, buffer is the starting address of a memory region. In 44Fortran, one can pass the first element of a memory region or a whole array, which must be 45'simply contiguous' (for 'simply contiguous,' see also Section 18.1.12).

4748

1

2

3

4

5

6

7

8

9

11

15

16

17

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

associated with the process.

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

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

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

#### 19 20 21

22

12

13

14

15 16

17

18

## 3.6.1 Model Implementation of Buffered Mode

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

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

30 31

32

33

34

35

36

37

38

39

40

41 42

43

44

45 46

47

48

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

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
- Compute the number, n, of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function

MPI\_PACK\_SIZE(count, datatype, comm, size), with the count, datatype and comm arguments used in the MPI\_BSEND call, returns an upper bound on the amount of space needed to buffer the message data (see Section 4.2). The MPI constant MPI\_BSEND\_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).

- Find the next contiguous empty space of n bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; MPI\_PACK is used to pack data.

- Post nonblocking send (standard mode) for packed data.
- Return

#### 3.7 Nonblocking Communication

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

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

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

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

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

If the send mode is **standard** then the send-complete call may return before a matching 4546receive is posted, if the message is buffered. On the other hand, the receive-complete may 47not complete until a matching receive is posted, and the message was copied into the receive 48 buffer.

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

1

2

3 4 5

6  $\overline{7}$ 

8

9

10 11

12

13

1415

16

17

18

19

 $^{24}$ 

25

26

2728

2930

31

32 33

34

35

36

37

38

39

40 41

42

43

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

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

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

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

23 24 25

# 3.7.1 Communication Request Objects

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

## 3.7.2 Communication Initiation

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

39

34

35

- 40 41
- 42
- 43
- 44 45
- 46
- 47
- . 48

1

2 3

4

5 6

7 8

9

10

11

12

13

14

15

16

17

18

19

20

21

# 3.7. NONBLOCKING COMMUNICATION

MPI_ISEND(buf, count, datatype, dest, tag, comm, request) <sup>1</sup>					
IN	buf	initial address of send buffer (choice)	2		
IN	count	number of elements in send buffer (non-negative integer)	3 4 5		
IN	datatype	datatype of each send buffer element (handle)	6		
IN	dest	rank of destination (integer)	7		
IN	tag	message tag (integer)	8		
IN	comm	communicator (handle)	9 10		
			11		
OUT	request	communication request (handle)	12		
int MPI		<pre>buf, int count, MPI_Datatype datatype, int dest, [_Comm comm, MPI_Request *request)</pre>	13 14 15		
MPI_Ise	nd(buf, count, dat	atype, dest, tag, comm, request, ierror)	16		
TYP	E(*), DIMENSION(	), INTENT(IN), ASYNCHRONOUS :: buf	17		
		: count, dest, tag	18 19		
	• •	NTENT(IN) :: datatype	20		
	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request				
	-	TENT(OUT) :: ierror	22		
MPI_ISE	MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)				
	<pre> type&gt; BUF(*) </pre>				
INT	INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR				
Star	Start a standard mode, nonblocking send. <sup>27</sup>				
	24				
MPI_IBS	END(buf, count, data	type, dest, tag, comm, request)	29 30		
IN	buf	initial address of send buffer (choice)	31		
IN	count	number of elements in send buffer (non-negative inte-	32		
	count	ger)	33 34		
IN	datatype	datatype of each send buffer element (handle)	35		
IN	dest	rank of destination (integer)	36		
IN	tag	message tag (integer)	37		
IN	comm	communicator (handle)	38 39		
OUT			40		
001	request	communication request (handle)	41		
int MPI	_Ibsend(const void	* buf, int count, MPI_Datatype datatype, int dest,	42		
int tag, MPI_Comm comm, MPI_Request *request)					
MPI Ibs	end(buf, count. da	tatype, dest, tag, comm, request, ierror)	44 45		
		), INTENT(IN), ASYNCHRONOUS :: buf	46		
	INTEGER, INTENT(IN) :: count, dest, tag 47				
TYPE(MPI_Datatype), INTENT(IN) :: datatype 48					

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

# 3.7. NONBLOCKING COMMUNICATION

MPI_IRSEND(buf, count, datatype, dest, tag, comm, request) <sup>1</sup>				
IN	buf	initial address of send buffer (choice)	2 3	
IN	count	number of elements in send buffer (non-negative integer)	3 4 5	
IN	datatype	datatype of each send buffer element (handle)	6	
IN	dest	rank of destination (integer)	7	
IN	tag	message tag (integer)	8 9	
IN	comm	communicator (handle)	10	
OUT	request	communication request (handle)	11 12	
int MPI_		nt count, MPI_Datatype datatype, int dest, omm, MPI_Request *request)	13 14 15	
<pre>MPI_Irsend(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    <sup>20</sup>    21 </pre>				
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR Start a ready mode nonblocking send. 23 23 24 25 26 27 27 28 27 28 27 28 27 28 29 29 20 20 20 20 20 20 20 20 20 20</type>				
MPI_IREC	V (buf, count, datatype, source	e, tag, comm, request)	29 30	
OUT	buf	initial address of receive buffer (choice)	31	
IN	count	number of elements in receive buffer (non-negative in- teger)	32 33 34	
IN	datatype	datatype of each receive buffer element (handle)	35	
IN	source	rank of source or MPI_ANY_SOURCE (integer)	36	
IN	tag	message tag or MPI_ANY_TAG (integer)	37	
IN	comm	communicator (handle)	38 39	
OUT	request	communication request (handle)	40	
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)				
<pre>MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) 45 TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf 46 INTEGER, INTENT(IN) :: count, source, tag 47 TYPE(MPI_Datatype), INTENT(IN) :: datatype 48</pre>				

1 TYPE(MPI\_Comm), INTENT(IN) :: comm 2 TYPE(MPI\_Request), INTENT(OUT) :: request 3 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 MPI\_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 5<type> BUF(\*) 6 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR 7 8 Start a nonblocking receive. 9 These calls allocate a communication request object and associate it with the request 10 handle (the argument request). The request can be used later to query the status of the 11communication or wait for its completion. 12A nonblocking send call indicates that the system may start copying data out of the 13send buffer. The sender should not modify any part of the send buffer after a nonblocking 14send operation is called, until the send completes. 15A nonblocking receive call indicates that the system may start writing data into the re-16ceive buffer. The receiver should not access any part of the receive buffer after a nonblocking 17receive operation is called, until the receive completes. 18 19Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 18.1.10-2018.1.20. (End of advice to users.) 2122233.7.3 Communication Completion  $^{24}$ The functions MPI\_WAIT and MPI\_TEST are used to complete a nonblocking communica-25tion. The completion of a send operation indicates that the sender is now free to update 26the locations in the send buffer (the send operation itself leaves the content of the send 27buffer unchanged). It does not indicate that the message has been received, rather, it may 28 have been buffered by the communication subsystem. However, if a synchronous mode 29 send was used, the completion of the send operation indicates that a matching receive was 30 initiated, and that the message will eventually be received by this matching receive.  $^{31}$ The completion of a receive operation indicates that the receive buffer contains the 32 received message, the receiver is now free to access it, and that the status object is set. It 33 does not indicate that the matching send operation has completed (but indicates, of course, 34 that the send was initiated). 35 We shall use the following terminology: A null handle is a handle with value 36 MPI\_REQUEST\_NULL. A persistent request and the handle to it are **inactive** if the request 37 is not associated with any ongoing communication (see Section 3.9). A handle is **active** 38 if it is neither null nor inactive. An **empty** status is a status which is set to return tag 39 = MPI\_ANY\_TAG, source = MPI\_ANY\_SOURCE, error = MPI\_SUCCESS, and is also internally 40 configured so that calls to MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and 41  $MPI_GET_ELEMENTS_X$  return count = 0 and  $MPI_TEST_CANCELLED$  returns false. We 42set a status variable to empty when the value returned by it is not significant. Status is set 43 in this way so as to prevent errors due to accesses of stale information. 44 The fields in a status object returned by a call to MPI\_WAIT, MPI\_TEST, or any

<sup>45</sup> The fields in a status object returned by a call to MPI\_WAIT, MPI\_TEST, or any <sup>46</sup> of the other derived functions (MPI\_{TEST|WAIT}{ALL|SOME|ANY}), where the request <sup>47</sup> corresponds to a send call, are undefined, with two exceptions: The error status field will <sup>48</sup>

### 3.7. NONBLOCKING COMMUNICATION

contain valid information if the wait or test call returned with MPI\_ERR\_IN\_STATUS; and the returned status can be queried by the call MPI\_TEST\_CANCELLED.

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

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

INTEGER REQUEST, STATUS(MPI\_STATUS\_SIZE), IERROR

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

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

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

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

Advice to implementors. In a multithreaded environment, a call to MPI\_WAIT should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (*End of advice to implementors.*)

 $\mathbf{2}$ 

 $^{24}$ 

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

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

```
4
MPI_REQUEST_FREE(request)
                                                                                         5
                                                                                         6
 INOUT
                                      communication request (handle)
           request
                                                                                         7
                                                                                         8
int MPI_Request_free(MPI_Request *request)
                                                                                        9
MPI_Request_free(request, ierror)
                                                                                        10
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                        11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        12
                                                                                        13
MPI_REQUEST_FREE(REQUEST, IERROR)
                                                                                        14
    INTEGER REQUEST, IERROR
                                                                                        15
    Mark the request object for deallocation and set request to MPI_REQUEST_NULL. An
                                                                                        16
ongoing communication that is associated with the request will be allowed to complete. The
                                                                                        17
request will be deallocated only after its completion.
                                                                                        18
                                                                                        19
     Rationale. The MPI_REQUEST_FREE mechanism is provided for reasons of perfor-
                                                                                        20
     mance and convenience on the sending side. (End of rationale.)
                                                                                        21
                                                                                        22
     Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not
                                                                                        23
     possible to check for the successful completion of the associated communication with
                                                                                        24
     calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the
                                                                                        25
     communication, an error code cannot be returned to the user — such an error must
                                                                                        26
     be treated as fatal. An active receive request should never be freed as the receiver
                                                                                        27
     will have no way to verify that the receive has completed and the receive buffer can
                                                                                        28
     be reused. (End of advice to users.)
                                                                                        29
                                                                                        30
                                                                                        31
                  An example using MPI_REQUEST_FREE.
Example 3.12
                                                                                        32
                                                                                        33
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
                                                                                        34
IF (rank.EQ.0) THEN
                                                                                        35
    DO i=1, n
                                                                                        36
      CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
                                                                                        37
      CALL MPI_REQUEST_FREE(req, ierr)
                                                                                        38
      CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
                                                                                        39
      CALL MPI_WAIT(req, status, ierr)
                                                                                        40
    END DO
                                                                                        41
ELSE IF (rank.EQ.1) THEN
                                                                                        42
    CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
                                                                                        43
                                                                                        44
    DO I=1, n-1
                                                                                        45
       CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
                                                                                        46
       CALL MPI_REQUEST_FREE(req, ierr)
                                                                                        47
       CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
```

Unofficial Draft for Comment Only

CALL MPI\_WAIT(req, status, ierr)

1

 $\mathbf{2}$ 

3

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

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

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

3.7.5 Multiple Completions

IERROR

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

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

				21
IN		count	list length (non-negative integer)	22
IN	OUT	array_of_requests	array of requests (array of handles)	23
Ol	JT	index	index of handle for operation that completed (integer)	24
01	JT	status	status object (Status)	25 26
00	01	Status	status object (Status)	20 27
int	MPI Wa	aitanv(int count, MPI Reg	<pre>uest array_of_requests[], int *index,</pre>	28
				29
мрт				30
MP1_	<pre>MPI_Waitany(count, array_of_requests, index, status, ierror) INTEGER, INTENT(IN) :: count</pre>			31
	TYPE(MPI Request) INTENT(INOUT) · array of requests(count)			32
		ER, INTENT(OUT) :: index		33
	TYPE(M	IPI_Status) :: status		34 35
	INTEGE	ER, OPTIONAL, INTENT(OUT)	:: ierror	36
MPT	WATTAN	IY(COUNT, ARRAY OF REQUES	TS, INDEX, STATUS, IERROR)	37
				38

Blocks until one of the operations associated with the active requests in the array has completed. If more than one operation is enabled and can terminate, one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing operation. (The array is indexed from zero in C, and from one in Fortran.) If the request is an active persistent request, it is marked inactive. Any other type of request is deallocated and the request handle is set to MPI\_REQUEST\_NULL.

The array\_of\_requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns

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

1

 $\mathbf{2}$ 

3

4

5 6

 $\overline{7}$ 

8

9 10 11

12

13

14

15

16

17

18 19 20

21

39

40

41

42

43

44

45

46

47

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

then the call returns immediately with flag = true, index = MPI\_UNDEFINED, and an empty status.

If the array of requests contains active handles then the execution of

<sup>43</sup> MPI\_TESTANY(count, array\_of\_requests, index, status) has the same effect as the execution <sup>44</sup> of MPI\_TEST( &array\_of\_requests[i], flag, status), for i=0, 1,..., count-1, in some arbitrary <sup>45</sup> order, until one call returns flag = true, or all fail. In the former case, index is set to the <sup>46</sup> last value of i, and in the latter case, it is set to MPI\_UNDEFINED. MPI\_TESTANY with an <sup>47</sup> array containing one active entry is equivalent to MPI\_TEST.

MPI_WAITALL( count, array_of_requests, array_of_statuses)			
IN	count	lists length (non-negative integer)	2
	array_of_requests	array of requests (array of handles)	3
		· - · · · /	4 5
OUT	array_of_statuses	array of status objects (array of Status)	6
int MDT L	Joitall (int count MDT Po	<pre>quest array_of_requests[],</pre>	7
IIIC MFI_V	MPI_Status array_of_		8
	· ·		9
	v 1	sts, array_of_statuses, ierror)	10
INTEGER, INTENT(IN) :: count			11 12
	TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count) TYPE(MPI_Status) :: array_of_statuses(*)		
	GER, OPTIONAL, INTENT(OUT		13 14
			15
	<pre>MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR) INTEGER COUNT, ARRAY_OF_REQUESTS(*)</pre>		
	GER ARRAY_OF_STATUSES(MPI		17
			18
Blocks until all communication operations associated with active handles in the list <sup>19</sup>			

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The i-th entry in array\_of\_statuses is set to the return status of the i-th operation. Active persistent requests are marked inactive. Requests of any other type are deallocated and the corresponding handles in the array are set to MPI\_REQUEST\_NULL. The list may contain null or inactive handles. The call sets to empty the status of each such entry.

The error-free execution of MPI\_WAITALL(count, array\_of\_requests, array\_of\_statuses) has the same effect as the execution of MPI\_WAIT(&array\_of\_request[i], &array\_of\_statuses[i]), for i=0,..., count-1, in some arbitrary order. MPI\_WAITALL with an array of length one is equivalent to MPI\_WAIT.

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

*Rationale.* This design streamlines error handling in the application. The application code need only test the (single) function result to determine if an error has occurred. It needs to check each individual status only when an error occurred. (*End of rationale.*)

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

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

If the list contains no active handles, then the call returns immediately with outcount = MPI\_UNDEFINED.

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

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

	IN	incount	length of array_of_requests (non-negative integer)	36
	INOUT	array_of_requests	array of requests (array of handles)	37
	moor	allay_ol_lequests	array of requests (array of nandres)	38
	OUT	outcount	number of completed requests (integer)	39
	OUT	array_of_indices	array of indices of operations that completed (array of	40
	5	integers)	41	
	OUT	array_of_statuses	array of status objects for operations that completed	42
	001	array_01_statuses		43
			(array of Status)	44
				45
<pre>int MPI_Testsome(int incount, MPI_Request array_of_requests[], 4</pre>				46
	<pre>int *outcount, int array_of_indices[], 47</pre>			
	MPI_Status array_of_statuses[]) 48			

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 

```
1
     MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
\mathbf{2}
                    array_of_statuses, ierror)
3
          INTEGER, INTENT(IN) :: incount
4
          TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
5
          INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
6
          TYPE(MPI_Status) :: array_of_statuses(*)
7
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
9
                    ARRAY_OF_STATUSES, IERROR)
10
          INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
11
              ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
12
13
         Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has
14
     completed it returns outcount = 0. If there is no active handle in the list it returns outcount
15
     = MPI_UNDEFINED.
16
          MPI_TESTSOME is a local operation, which returns immediately, whereas
17
     MPI_WAITSOME will block until a communication completes, if it was passed a list that
18
     contains at least one active handle. Both calls fulfill a fairness requirement: If a request
19
     for a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or
20
     MPI_TESTSOME, and a matching send has been posted, then the receive will eventually
21
     succeed, unless the send is satisfied by another receive; and similarly for send requests.
22
          Errors that occur during the execution of MPI_TESTSOME are handled as for
23
     MPI_WAITSOME.
^{24}
           Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use
25
           of MPI_TESTANY. The former returns information on all completed communications,
26
           with the latter, a new call is required for each communication that completes.
27
28
           A server with multiple clients can use MPI_WAITSOME so as not to starve any client.
29
           Clients send messages to the server with service requests. The server calls
30
           MPI_WAITSOME with one receive request for each client, and then handles all receives
31
           that completed. If a call to MPI_WAITANY is used instead, then one client could starve
32
           while requests from another client always sneak in first. (End of advice to users.)
33
34
           Advice to implementors. MPI_TESTSOME should complete as many pending com-
           munications as possible. (End of advice to implementors.)
35
36
37
     Example 3.15
                        Client-server code (starvation can occur).
38
39
40
     CALL MPI_COMM_SIZE(comm, size, ierr)
41
     CALL MPI_COMM_RANK(comm, rank, ierr)
42
     IF(rank .GT. 0) THEN
                                      ! client code
43
          DO WHILE(.TRUE.)
44
             CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
45
             CALL MPI_WAIT(request, status, ierr)
46
          END DO
47
     ELSE
                    ! rank=0 -- server code
48
             DO i=1, size-1
```

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

### 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.

38

39

40

41

42

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

```
2
       IN
                                              request (handle)
                 request
3
       OUT
                 flag
                                              boolean flag, same as from MPI_TEST (logical)
4
       OUT
                                              status object if flag is true (Status)
                 status
5
6
\overline{7}
     int MPI_Request_get_status(MPI_Request request, int *flag,
8
                     MPI_Status *status)
9
     MPI_Request_get_status(request, flag, status, ierror)
10
          TYPE(MPI_Request), INTENT(IN) :: request
11
          LOGICAL, INTENT(OUT) :: flag
12
          TYPE(MPI_Status) :: status
13
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
14
15
     MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
16
          INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
17
          LOGICAL FLAG
18
          Sets flag=true if the operation is complete, and, if so, returns in status the request
19
     status. However, unlike test or wait, it does not deallocate or inactivate the request; a
20
     subsequent call to test, wait or free should be executed with that request. It sets flag=false
21
     if the operation is not complete.
22
          One is allowed to call MPI_REQUEST_GET_STATUS with a null or inactive request
23
     argument. In such a case the operation returns with flag=true and empty status.
24
```

2526

27

# 3.8 Probe and Cancel

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

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

3.8.1 Probe

38 39

40

41

36 37

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

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

## Unofficial Draft for Comment Only

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

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

If MPI\_IPROBE returns flag = true, then the content of the status object can be subsequently accessed as described in Section 3.2.5 to find the source, tag and length of the probed message.

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

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

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

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

IN	source	rank of source or $MPI\_ANY\_SOURCE$ (integer)	39	
IN	tag	message tag or $MPI\_ANY\_TAG$ (integer)	40	
IN	comm	communicator (handle)	41	
IIN	comm	communicator (nandic)	42	
OUT	status	status object (Status)	43	
			44	
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status) 45				
4				
MPI_Probe(source, tag, comm, status, ierror)				
INTEGER, INTENT(IN) :: source, tag			48	

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

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

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

100

200

0, comm, status, ierr) ELSE CALL MPI\_RECV(x, 1, MPI\_REAL, MPI\_ANY\_SOURCE, 0, comm, status, ierr) END IF END DO

IF (status(MPI\_SOURCE) .EQ. 0) THEN

END IF

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

comm, status, ierr)

CALL MPI\_RECV(i, 1, MPI\_INTEGER, MPI\_ANY\_SOURCE,

Advice to users. In a multithreaded MPI program, MPI\_PROBE and MPI\_IPROBE might need special care. If a thread probes for a message and then immediately posts a matching receive, the receive may match a message other than that found by the probe since another thread could concurrently receive that original message [29]. MPI\_MPROBE and MPI\_IMPROBE solve this problem by matching the incoming message so that it may only be received with MPI\_MRECV or MPI\_IMRECV on the corresponding message handle. (*End of advice to users.*)

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

## 3.8.2 Matching Probe

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

Like MPI\_PROBE and MPI\_IPROBE, the MPI\_MPROBE and MPI\_IMPROBE operations allow incoming messages to be queried without actually receiving them, except that MPI\_MPROBE and MPI\_IMPROBE provide a mechanism to receive the specific message that was matched regardless of other intervening probe or receive operations. This gives the application an opportunity to decide how to receive the message, based on the information returned by the probe. In particular, the user may allocate memory for the receive buffer, according to the length of the probed message. 42 44 43 44 44 45 46 46 47 48

### Unofficial Draft for Comment Only

69

1

2

3

4

5 6

78

9

10

11

12

13

14

15

16

17

18

19

20

21

22 23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36 37

38 39

<sup>1</sup> MPI\_IMPROBE(source, tag, comm, flag, message, status)

1	MPI_IMPR	OBE(source, tag, comm, flag, l	message, status)
2 3	IN	source	rank of source or $MPI\_ANY\_SOURCE$ (integer)
4	IN	tag	message tag or MPI_ANY_TAG (integer)
5	IN	comm	communicator (handle)
6	OUT	flag	flag (logical)
7 8	OUT	message	returned message (handle)
9	OUT	status	status object (Status)
10	001	Status	status object (Status)
11 12 13	int MPI_I	mprobe(int source, int ta MPI_Message *message;	ng, MPI_Comm comm, int *flag, , MPI_Status *status)
14 15 16 17 18 19 20	INTEG TYPE( LOGIC TYPE( TYPE(	be(source, tag, comm, fla ER, INTENT(IN) :: source MPI_Comm), INTENT(IN) :: AL, INTENT(OUT) :: flag MPI_Message), INTENT(OUT) MPI_Status) :: status ER, OPTIONAL, INTENT(OUT)	comm :: message
21 22	MPI_IMPRO	BE(SOURCE, TAG, COMM, FLA	G, MESSAGE, STATUS, IERROR) SAGE, STATUS(MPI_STATUS_SIZE), IERROR
23 24	LOGIC	AL FLAG	
25		. –	flag, message, status) returns flag = true if there is
26	0		t matches the pattern specified by the arguments s the same message that would have been received
27 28			comm, status) executed at the same point in the
29	program ar	nd returns in <b>status</b> the same va	alue that would have been returned by MPI_RECV.
30			dle to the matched message. Otherwise, the call
31 32		g = false, and leaves status and ched receive (MPL MRECV o	r MPI_IMRECV) executed with the message han-
33			natched by the probe. Unlike MPI_IPROBE, no
34		8	match the message returned by MPI_IMPROBE.
35		0	BE must be received with either MPI_MRECV or
36	MPI_IMRE		
37		_	OBE can be MPI_ANY_SOURCE, and the tag argu- can probe for messages from an arbitrary source
$\frac{38}{39}$			a specific communication context must be provided
40	,	omm argument.	i specific communication context must be provided
41		0	is matched with MPI_IMPROBE or MPI_MPROBE
42	•	-	matching receive is posted with MPI_MRECV or
43	-	о о	n has started to receive the message sent by the
44	synchronou	· · · · ·	
45	There	is a special predefined messa	ge: MPI_MESSAGE_NO_PROC, which is a message
46	which has	MPI_PROC_NULL as its source	e process. The predefined constant
47	MPI_MESSA	AGE_NULL is the value used for	r invalid message handles.
48			
			the form Community Orabi

	AGE_NO_PROC, and the stat $NY_TAG$ , and count = 0; see S	C_NULL as source returns flag = true, message = us object returns source = MPI_PROC_NULL, tag Section 3.11. It is not necessary to call MPI_MRECV NO_PROC, but it is not erroneous to do so.	1 2 3 4 5		
MPI.		<b>D_PROC</b> was chosen instead of avoid possible confusion as another null handle con-	6 7 8 9 10		
MPI_MPR	OBE(source, tag, comm, mess	age, status)	11 12		
IN	source	rank of source or MPI_ANY_SOURCE (integer)	12		
IN	tag	message tag or MPI_ANY_TAG (integer)	14		
IN	comm	communicator (handle)	15		
OUT	message	returned message (handle)	16 17		
OUT	status	status object (Status)	18		
001	Status	status object (Status)	19		
int MPI_N	<pre>/probe(int source, int ta</pre>	ag, MPI_Comm comm, MPI_Message *message,	20		
	MPI_Status *status)		21 22		
MPI Mprob	pe(source, tag, comm, mes	ssage, status, ierror)	22		
-	GER, INTENT(IN) :: sourc	-	24		
	(MPI_Comm), INTENT(IN) ::		25		
	TYPE(MPI_Message), INTENT(OUT) :: message TYPE(MPI_Status) :: status				
	(MPI_Status) :: status GER, OPTIONAL, INTENT(OU7	C) :: ierror	27 28		
			29		
	BE(SOURCE, TAG, COMM, MES BER SOURCE, TAG, COMM, ME		30		
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR					
		APROBE except that it is a blocking call that returns	32 33		
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.				
			36		
3.8.3 Ma	atched Receives		37 38		
The functi	The functions MPI_MRECV and MPI_IMRECV receive messages that have been previously				
matched b	y a matching probe (Section	3.8.2).	40		
			41		
			42		
			43		
			44 45		
			46		
			47		
			48		

1 MPI\_MRECV(buf, count, datatype, message, status) 2 OUT buf initial address of receive buffer (choice) 3 IN count number of elements in receive buffer (non-negative in-4 teger) 56 IN datatype of each receive buffer element (handle) datatype 7 INOUT message message (handle) 8 OUT status status object (Status) 9 10 11int MPI\_Mrecv(void\* buf, int count, MPI\_Datatype datatype, MPI\_Message \*message, MPI\_Status \*status) 1213 MPI\_Mrecv(buf, count, datatype, message, status, ierror) 14TYPE(\*), DIMENSION(..) :: buf 15INTEGER, INTENT(IN) :: count 16TYPE(MPI\_Datatype), INTENT(IN) :: datatype 17 TYPE(MPI\_Message), INTENT(INOUT) :: message 18 TYPE(MPI\_Status) :: status 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI\_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR) 2122 <type> BUF(\*) 23INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 24This call receives a message matched by a matching probe operation (Section 3.8.2). 25The receive buffer consists of the storage containing **count** consecutive elements of the 26type specified by datatype, starting at address buf. The length of the received message must 27be less than or equal to the length of the receive buffer. An overflow error occurs if all 28incoming data does not fit, without truncation, into the receive buffer. 29 If the message is shorter than the receive buffer, then only those locations corresponding 30 to the (shorter) message are modified.  $^{31}$ On return from this function, the message handle is set to MPI\_MESSAGE\_NULL. All 32 errors that occur during the execution of this operation are handled according to the error 33 handler set for the communicator used in the matching probe call that produced the message 34 handle. 35 If MPI\_MRECV is called with MPI\_MESSAGE\_NO\_PROC as the message argument, the 36 call returns immediately with the status object set to source = MPI\_PROC\_NULL, tag =37 MPI\_ANY\_TAG, and count = 0, as if a receive from MPI\_PROC\_NULL was issued (see Sec-38 tion 3.11). A call to MPI\_MRECV with MPI\_MESSAGE\_NULL is erroneous. 39 40 41 4243 44 4546 47 48

MPI	IMRE	CV(buf, count, datatype, messa	are request)	1
OU <sup>-</sup>		buf	initial address of receive buffer (choice)	2
				3
IN		count	number of elements in receive buffer (non-negative in-teger)	4 5
IN		datatype	datatype of each receive buffer element (handle)	6
INO	UT	message	message (handle)	7
OU <sup>-</sup>	Г	request	communication request (handle)	8
	-			9 10
int N	int MPI_Imrecv(void* buf, int count, MPI_Datatype datatype,			
			MPI_Request *request)	12
мрт і	mreci	v(buf, count, datatype, m	essage request jerror)	13
		*), DIMENSION(), ASYNCH		14
	INTEGER, INTENT(IN) :: count			15
]	TYPE(MPI_Datatype), INTENT(IN) :: datatype		:: datatype	16
		<pre>MPI_Message), INTENT(INOU</pre>		17 18
		PE(MPI_Request), INTENT(OUT) :: request		
_	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			20
MPI_]	MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)			21
	<type> BUF(*)</type>			22
]	INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR			23
Ν	MPI_IMRECV is the nonblocking variant of MPI_MRECV and starts a nonblocking		24	
receiv	e of a	matched message. Completion	n semantics are similar to $MPI\_IRECV$ as described	25 26
			unction, the message handle is set to	20
		GE_NULL.		28
	If MPI_IMRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the all returns immediately with a request object which, when completed, will yield a status bject set to source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0, as if a		29	
			30	
-	evice from MPI_PROC_NULL was issued (see Section 3.11). A call to MPI_IMRECV with			31
		GE_NULL is erroneous.		32
	1			33 34
	Advice to implementors. If reception of a matched message is started with			35
	MPI_IMRECV, then it is possible to cancel the returned request with MPI_CANCEL. If MPI_CANCEL succeeds, the matched message must be found by a subsequent message			
			, MPI_MPROBE, or MPI_IMPROBE), received by	37
	-	,	ancelled by the sender. See Section 3.8.4 for details	38
			ion of operations initiated with $MPI_{IMRECV}$ may	39
	fail. (	End of advice to implementor	rs.)	40
204	C	1		41 42
3.8.4	Can	cel		42
				10

 MPI\_CANCEL(request)

 IN
 request

 communication request (handle)

# Unofficial Draft for Comment Only

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

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

<sup>18</sup> MPI\_CANCEL can be used to cancel a communication that uses a persistent request (see <sup>19</sup> Section 3.9), in the same way it is used for nonpersistent requests. A successful cancellation <sup>20</sup> cancels the active communication, but not the request itself. After the call to MPI\_CANCEL <sup>21</sup> and the subsequent call to MPI\_WAIT or MPI\_TEST, the request becomes inactive and can <sup>22</sup> be activated for a new communication.

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

Either the cancellation succeeds, or the communication succeeds, but not both. If a 25send is marked for cancellation, then it must be the case that either the send completes 26normally, in which case the message sent was received at the destination process, or that 27the send is successfully cancelled, in which case no part of the message was received at the 28destination. Then, any matching receive has to be satisfied by another send. If a receive is 29 marked for cancellation, then it must be the case that either the receive completes normally, 30 or that the receive is successfully cancelled, in which case no part of the receive buffer is  $^{31}$ altered. Then, any matching send has to be satisfied by another receive. 32

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

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

### MPI\_TEST\_CANCELLED(status, flag)

43	IN	status	status object (Status)
44	OUT	flag	(logical)
45		5	( )
46 47	int MPI_	<pre>Fest_cancelled(const M</pre>	PI_Status *status, int *flag)
48	MPI_Test	_cancelled(status, fla	g, ierror)

Unofficial Draft for Comment Only

8

33

34 35

36

37

38

39 40 41

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

Returns flag = true if the communication associated with the status object was cancelled successfully. In such a case, all other fields of status (such as count or tag) are undefined. Returns flag = false, otherwise. If a receive operation might be cancelled then one should call MPI\_TEST\_CANCELLED first, to check whether the operation was cancelled, before checking on the other fields of the return status.

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

Advice to implementors. If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement

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

# 3.9 Persistent Communication Requests

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

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

> 44 45 46

47 48

9

10

11

12

13

14

15 16

17

18

19

20

21

22

23

 $^{24}$ 

25 26 27

28 29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

1 MPI\_SEND\_INIT(buf, count, datatype, dest, tag, comm, request) 2 IN buf initial address of send buffer (choice) 3 IN count number of elements sent (non-negative integer) 4 5IN datatype type of each element (handle) 6 IN dest rank of destination (integer) 7 IN message tag (integer) tag 8 9 IN communicator (handle) comm 10 OUT communication request (handle) request 11 12int MPI\_Send\_init(const void\* buf, int count, MPI\_Datatype datatype, 13 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 1415MPI\_Send\_init(buf, count, datatype, dest, tag, comm, request, ierror) 16TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 17 INTEGER, INTENT(IN) :: count, dest, tag 18 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 19 TYPE(MPI\_Comm), INTENT(IN) :: comm 20TYPE(MPI\_Request), INTENT(OUT) :: request 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 MPI\_SEND\_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 23<type> BUF(\*) 24INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 2526Creates a persistent communication request for a standard mode send operation, and 27binds to it all the arguments of a send operation. 2829 MPI\_BSEND\_INIT(buf, count, datatype, dest, tag, comm, request) 30  $^{31}$ IN buf initial address of send buffer (choice) 32 IN number of elements sent (non-negative integer) count 33 34IN datatype type of each element (handle) 35 IN dest rank of destination (integer) 36 IN message tag (integer) tag 37 IN comm communicator (handle) 38 39 OUT request communication request (handle) 4041 int MPI\_Bsend\_init(const void\* buf, int count, MPI\_Datatype datatype, 42int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 43 MPI\_Bsend\_init(buf, count, datatype, dest, tag, comm, request, ierror) 44TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 45INTEGER, INTENT(IN) :: count, dest, tag 4647 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 48 TYPE(MPI\_Comm), INTENT(IN) :: comm

	TYPE(MPI_Request), INTENT(OUT) INTEGER, OPTIONAL, INTENT(OUT)	-	1 $2$
MPI	BSEND INIT(BUF. COUNT. DATATYP	E, DEST, TAG, COMM, REQUEST, IERROR)	3
-	<type> BUF(*)</type>		4 5
	INTEGER COUNT, DATATYPE, DEST,	TAG, COMM, REQUEST, IERROR	6
	Creates a persistent communication	request for a buffered mode send.	7
			8
MDI	_SSEND_INIT(buf, count, datatype, d	lest tog comm request)	9
		<b>c</b> . ,	10 11
IN	buf	initial address of send buffer (choice)	12
IN	count	number of elements sent (non-negative integer)	13
IN	datatype	type of each element (handle)	14
IN	dest	rank of destination (integer)	15
IN	tag	message tag (integer)	16 17
IN	comm	communicator (handle)	17
οι	T request	communication request (handle)	19
	·		20
int	MPI_Ssend_init(const void* buf	, int count, MPI_Datatype datatype,	21
	int dest, int tag, MF	PI_Comm comm, MPI_Request *request)	22 23
MPI_	MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)		
	TYPE(*), DIMENSION(), INTENT		24 25
	INTEGER, INTENT(IN) :: count,	-	26
	TYPE(MPI_Datatype), INTENT(IN)	v1	27
	TYPE(MPI_Comm), INTENT(IN) :: TYPE(MPI_Request), INTENT(OUT)		28
	INTEGER, OPTIONAL, INTENT(OUT)	-	29 30
			31
MP1_	SSEND_INIT(BUF, CUUNT, DATATYP <type> BUF(*)</type>	E, DEST, TAG, COMM, REQUEST, IERROR)	32
	INTEGER COUNT, DATATYPE, DEST,	TAG, COMM, REQUEST, IERROR	33
			34
	Creates a persistent communication	object for a synchronous mode send operation.	35
			36 37
MPI.	_RSEND_INIT(buf, count, datatype, c	lest, tag, comm, request)	38
IN	buf	initial address of send buffer (choice)	39
IN	count	number of elements sent (non-negative integer)	40
IN	datatype	type of each element (handle)	41
IN	dest	rank of destination (integer)	42 43
IN	tag	message tag (integer)	43 44
IN			45
	comm	communicator (handle)	46
Οι	T request	communication request (handle)	47
			48

```
1
     int MPI_Rsend_init(const void* buf, int count, MPI_Datatype datatype,
\mathbf{2}
                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
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_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
12
         <type> BUF(*)
13
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
14
         Creates a persistent communication object for a ready mode send operation.
15
16
17
     MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)
18
       OUT
                buf
                                            initial address of receive buffer (choice)
19
       IN
                                            number of elements received (non-negative integer)
                count
20
21
       IN
                datatype
                                            type of each element (handle)
22
       IN
                                            rank of source or MPI_ANY_SOURCE (integer)
                source
23
       IN
                                            message tag or MPI_ANY_TAG (integer)
                tag
24
25
       IN
                comm
                                            communicator (handle)
26
       OUT
                request
                                            communication request (handle)
27
28
     int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
29
                    int tag, MPI_Comm comm, MPI_Request *request)
30
^{31}
     MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
32
33
         INTEGER, INTENT(IN) :: count, source, tag
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
36
         TYPE(MPI_Request), INTENT(OUT) :: request
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
39
         <type> BUF(*)
40
         INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
41
42
         Creates a persistent communication request for a receive operation. The argument buf
43
     is marked as OUT because the user gives permission to write on the receive buffer by passing
44
     the argument to MPI_RECV_INIT.
45
         A persistent communication request is inactive after it was created — no active com-
46
     munication is attached to the request.
47
         A communication (send or receive) that uses a persistent request is initiated by the
48
     function MPI_START.
```

78

		·	
MPI_ST/	ART(request)		1
INOUT	request	communication request (handle)	2
			3 4
int MPI	_Start(MPI_Request *re	quest)	т 5
MDT C+a	nt (noquoat ionnon)		6
	rt(request, ierror) E(MPI_Request), INTENT	(INOUT) :: request	7
	EGER, OPTIONAL, INTENT	±	8
			9
	RT(REQUEST, IERROR)		10
INT	EGER REQUEST, IERROR		11
The	argument, request, is a h	andle returned by one of the previous five calls. The	12
	<b>e</b> , <b>i</b> ,	ve. The request becomes active once the call is made.	13
If th	ne request is for a send with	h ready mode, then a matching receive should be posted	14
before the	ne call is made. The comm	nunication buffer should not be modified after the call,	15
and unti	l the operation completes.		16
	,	semantics to the nonblocking communication operations	17 18
		a call to MPI_START with a request created by	18
		ication in the same manner as a call to MPI_ISEND; a	20
	•	created by MPI_BSEND_INIT starts a communication	21
in the sa	me manner as a call to MF	PI_IBSEND; and so on.	22
			23
MPI_ST/	ARTALL(count, array_of_re	quests)	24
IN	count	list length (non-negative integer)	25
			26
INOUT	array_of_requests	array of requests (array of handle)	27
	~		28
int MPI	_Startall(int count, M	<pre>PI_Request array_of_requests[])</pre>	29
MPI_Sta	rtall(count, array_of_:	requests, ierror)	30
	EGER, INTENT(IN) :: c	-	31
TYP	E(MPI_Request), INTENT	<pre>(INOUT) :: array_of_requests(count)</pre>	32
INT	EGER, OPTIONAL, INTENT	(OUT) :: ierror	33 34
MPT STA	RTALL(COUNT, ARRAY_OF_	REQUESTS, TERROR)	34 35
	EGER COUNT, ARRAY_OF_R		36
			37
Star	t all communications asso	ociated with requests in array_of_requests. A call to	20

Start all communications associated with requests in array\_of\_requests. A call to MPI\_STARTALL(count, array\_of\_requests) has the same effect as calls to MPI\_START (&array\_of\_requests[i]), executed for i=0,..., count-1, in some arbitrary order.

A communication started with a call to MPI\_START or MPI\_STARTALL is completed by a call to MPI\_WAIT, MPI\_TEST, or one of the derived functions described in Section 3.7.5. The request becomes inactive after successful completion of such call. The request is not deallocated and it can be activated anew by an MPI\_START or MPI\_STARTALL call.

A persistent request is deallocated by a call to MPI\_REQUEST\_FREE (Section 3.7.3). The call to MPI\_REQUEST\_FREE can occur at any point in the program after the persistent request was created. However, the request will be deallocated only after it becomes

#### Unofficial Draft for Comment Only

38

39

40

41

42

43

44

45

46

47

inactive. Active receive requests should not be freed. Otherwise, it will not be possible
 to check that the receive has completed. It is preferable, in general, to free requests when
 they are inactive. If this rule is followed, then the functions described in this section will
 be invoked in a sequence of the form,

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

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

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

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

### 3.10 Send-Receive

The send-receive operations combine in one call the sending of a message to one desti-nation and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the com-munication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 7 in order to perform shifts on var-ious logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.  $^{31}$ 

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.

by a join of these two threads.

MPI_SENI	DRECV(sendbuf, sendcount, se source, recvtag, comm, s	ndtype, dest, sendtag, recvbuf, recvcount, recvtype, tatus)	1 2
IN	sendbuf	initial address of send buffer (choice)	3
IN	sendcount	number of elements in send buffer (non-negative inte- ger)	4 5 6
IN	sendtype	type of elements in send buffer (handle)	7
IN	dest	rank of destination (integer)	8 9
IN	sendtag	send tag (integer)	9 10
OUT	recvbuf	initial address of receive buffer (choice)	11
IN	recvcount	number of elements in receive buffer (non-negative in- teger)	12 13 14
IN	recvtype	type of elements in receive buffer (handle)	14
IN	source	rank of source or $MPI_ANY_SOURCE$ (integer)	16
IN	recvtag	receive tag or MPI_ANY_TAG (integer)	17
IN	comm	communicator (handle)	18 19
OUT	status	status object (Status)	20
			21
int MPI_S		buf, int sendcount, MPI_Datatype sendtype,	22 23
		g, void *recvbuf, int recvcount,	23 24
	MPI_Datatype recvtyp MPI_Status *status)	e, int source, int recvtag, MPI_Comm comm,	25 26
MPI_Sendi	cecv(sendbuf, sendcount,	sendtype, dest, sendtag, recvbuf,	27
	• -	source, recvtag, comm, status, ierror)	28
	(*), DIMENSION(), INTEN (*), DIMENSION() :: r	Γ(IN) :: sendbuf ecvbuf	29
		ount, dest, sendtag, recvcount, source,	30 31
	recvtag	,,,,,,,	32
TYPE	(MPI_Datatype), INTENT(IN	) :: sendtype, recvtype	33
	(MPI_Comm), INTENT(IN) ::	comm	34
	(MPI_Status) :: status	) iernen	35
	ER, OPTIONAL, INTENT(OUT		36 37
MPI_SENDF		SENDTYPE, DEST, SENDTAG, RECVBUF,	38
(two	RECVCUUNT, RECVTYPE, >> SENDBUF(*), RECVBUF(*)	SOURCE, RECVTAG, COMM, STATUS, IERROR)	39
• 1		DEST, SENDTAG, RECVCOUNT, RECVTYPE,	40
		ATUS(MPI_STATUS_SIZE), IERROR	41
Exect	ite a blocking send and receiv	ve operation. Both send and receive use the same	42 43
	0	ags. The send buffer and receive buffers must be	44

disjoint, and may have different lengths and datatypes. The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed

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

45

46

47

```
1
     MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm,
\mathbf{2}
                     status)
3
       INOUT
                 buf
                                              initial address of send and receive buffer (choice)
4
       IN
                                              number of elements in send and receive buffer (non-
                 count
5
                                              negative integer)
6
7
       IN
                 datatype
                                              type of elements in send and receive buffer (handle)
8
       IN
                 dest
                                              rank of destination (integer)
9
       IN
                 sendtag
                                              send message tag (integer)
10
11
       IN
                 source
                                              rank of source or MPI_ANY_SOURCE (integer)
12
       IN
                 recvtag
                                              receive message tag or MPI_ANY_TAG (integer)
13
       IN
                                              communicator (handle)
                 comm
14
15
       OUT
                 status
                                              status object (Status)
16
17
     int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
18
                     int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
19
                     MPI_Status *status)
20
     MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
21
                     comm, status, ierror)
22
          TYPE(*), DIMENSION(..) :: buf
23
          INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
24
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Comm), INTENT(IN) :: comm
26
          TYPE(MPI_Status) :: status
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
30
                     COMM, STATUS, IERROR)
31
          <type> BUF(*)
32
          INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
33
               STATUS(MPI_STATUS_SIZE), IERROR
34
          Execute a blocking send and receive. The same buffer is used both for the send and
35
     for the receive, so that the message sent is replaced by the message received.
36
37
           Advice to implementors. Additional intermediate buffering is needed for the "replace"
38
           variant. (End of advice to implementors.)
39
40
41
             Null Processes
     3.11
42
43
     In many instances, it is convenient to specify a "dummy" source or destination for commu-
44
     nication. This simplifies the code that is needed for dealing with boundaries, for example,
45
     in the case of a non-circular shift done with calls to send-receive.
46
          The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a
```

<sup>40</sup> The special value MPI\_PROC\_NULL can be used instead of a rank wherever a source or a
 <sup>47</sup> destination argument is required in a call. A communication with process MPI\_PROC\_NULL
 <sup>48</sup> has no effect. A send to MPI\_PROC\_NULL succeeds and returns as soon as possible. A receive

from MPI\_PROC\_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with source = MPI\_PROC\_NULL is executed then the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG and count = 0. A probe or matching probe with source = MPI\_PROC\_NULL succeeds and returns as soon as possible, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG and count = 0. A matching probe (cf. Section 3.8.2) with MPI\_PROC\_NULL as source returns flag = true, message = MPI\_MESSAGE\_NO\_PROC, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0.

 $\mathbf{2}$ 

# Chapter 4

# Datatypes

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

#### 4.1**Derived** Datatypes

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

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shapes and sizes. It is not assumed that the MPI library is cognizant of 34 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 36 that mimics the definition of the structure or array section in question. These facilities can 37 be used by library designers to define communication functions that can transfer objects defined in the host language — by decoding their definitions as available in a symbol table or a dope vector. Such higher-level communication functions are not part of MPI.

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

• •

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

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

7 8 9

15

16

17

18 19

2021

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

35

38

39

40

41

42

43

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

6 7

1

 $\mathbf{2}$ 

3

4

5

8 9

10

1112

13

14

15

18

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

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

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

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

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

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

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

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

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

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

then

38 39 40

32

33

34 35

36

37

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

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

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

41 42

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

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

Let

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

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

#### 4.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI\_TYPE\_CREATE\_HVECTOR, MPI\_TYPE\_CREATE\_HINDEXED, MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK, MPI\_TYPE\_CREATE\_STRUCT, and MPI\_GET\_ADDRESS accept arguments of type INTEGER(KIND=MPI\_ADDRESS\_KIND), wherever arguments of type MPI\_Aint are used in C. On Fortran 77 systems that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs are 32 bits, these arguments will be of type INTEGER\*8.

#### 4.1.2 Datatype Constructors

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

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

IN	count	replication count (non-negative integer)	34
IN	oldtyra	ald datatuma (bandla)	35
IIN	oldtype	old datatype (handle)	36
OUT	newtype	new datatype (handle)	37
			38
int MPI_T	<pre>ype_contiguous(int count,</pre>	MPI_Datatype oldtype,	39
	MPI_Datatype *newtype	e)	40
			41
• 1	contiguous(count, oldtype	e, newtype, lerror)	42
	ER, INTENT(IN) :: count		43
	<pre>MPI_Datatype), INTENT(IN)</pre>	V 1	44
	<pre>MPI_Datatype), INTENT(OUT</pre>		45
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	46
MDT TVDF (	CONTIGUOUS(COUNT, OLDTYPE	NEWTVDE TERROR)	47
	ER COUNT, OLDTYPE, NEWTYP		48
TNIEG	IN COOMI, OLDIIPE, NEWIIP	E, IERRUR	

#### Unofficial Draft for Comment Only

1 2

3

4

5 6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16 17

18 19

20

21

22

23

24

25

26 27

28 29

30

31 32

1 2			by concatenating <b>count</b> copies of <i>extent</i> as the size of the concatenated copies.		
3 4 5	<b>Example 4.2</b> Let oldtype have type map $\{(double, 0), (char, 8)\}$ , with extent 16, and let count = 3. The type map of the datatype returned by newtype is				
6	{(do	(char, 8), (double, 10)	(char, 24), (double, 32), (char, 40);		
7 8	i.e., altern	ating double and char eleme	nts, with displacements $0, 8, 16, 24, 32, 40$ .		
9	In gei	neral, assume that the type m	ap of <b>oldtype</b> is		
10	$\{(ty)$	$pe_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$_{n-1})\},$		
11 12	with exter	nt $ex$ . Then newtype has a type	be map with $count \cdot n$ entries defined by:		
13	$\{(type)\}$	$(1, disp_0), \ldots, (type_{n-1}, disp_{n-1})$	( $type_0, disp_0 + ex$ ),, ( $type_{n-1}, disp_{n-1} + ex$ ),		
14 15	$\ldots, (ty$	$ppe_0, disp_0 + ex \cdot (count - 1)),$	$\ldots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}.$		
16 17 18 19 20 21	cation of obtained l	a datatype into locations that	FOR is a more general constructor that allows repli- at consist of equally spaced blocks. Each block is umber of copies of the old datatype. The spacing nt of the old datatype.		
22 23	ΜΡΙ_ΤΥΡ	E_VECTOR(count, blocklength	n, stride, oldtype, newtype)		
24	IN	count	number of blocks (non-negative integer)		
25 26	IN	blocklength	number of elements in each block (non-negative integer)		
27 28 29	IN	stride	number of elements between start of each block (integer)		
30	IN	oldtype	old datatype (handle)		
31 32	OUT	newtype	new datatype (handle)		
33 34 35	int MPI_1		t blocklength, int stride, e, MPI_Datatype *newtype)		
36 37 38 39 40	INTE TYPE TYPE	_vector(count, blocklengt GER, INTENT(IN) :: count (MPI_Datatype), INTENT(IN (MPI_Datatype), INTENT(OU GER, OPTIONAL, INTENT(OUT	) :: oldtype T) :: newtype		
41 42 43			H, STRIDE, OLDTYPE, NEWTYPE, IERROR) TRIDE, OLDTYPE, NEWTYPE, IERROR		
44 45 46 47	-	A call to MPI_TYPE_VECTO	dtype has type map $\{(double, 0), (char, 8)\}$ , with PR(2, 3, 4, oldtype, newtype) will create the datatype		
48	$\{(do$	uble, 0), (char, 8), (double, 16)	5), (char, 24), (double, 32), (char, 40),		

"heterogeneous").

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

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

#### Unofficial Draft for Comment Only

36

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

MPI_TY	PE_INDEXED(count, array_of_ newtype)	_blocklengths, array_of_displacements, oldtype,	$\frac{1}{2}$
IN	count	number of blocks — also number of entries in array_of_displacements and array_of_blocklengths (non- negative integer)	3 4 5 6
IN	array_of_blocklengths	number of elements per block (array of non-negative integers)	7 8
IN	array_of_displacements	displacement for each block, in multiples of oldtype extent (array of integer)	9 10
IN	oldtype	old datatype (handle)	11
OUT	newtype	new datatype (handle)	12 13
int MPI		<pre>const int array_of_blocklengths[], displacements[], MPI_Datatype oldtype, rpe)</pre>	14 15 16 17
INT TYP TYP	oldtype, newtype, i	t, array_of_blocklengths(count), ount) N) :: oldtype UT) :: newtype	18 19 20 21 22 23 24 25
	OLDTYPE, NEWTYPE, J	KLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),	26 27 28 29 30
and let	oldtype have type map {(down a state of the	ble, 0), (char, 8), with extent 16. Let $B = (3, 1)YPE_INDEXED(2, B, D, oldtype, newtype) returns a$	31 32 33 34 35
{(c	double, 64), (char, 72), (double)	(a, 80), (char, 88), (double, 96), (char, 104),	36 37
(da	$\texttt{puble}, 0), (\texttt{char}, 8) \}.$		38
displace		carting at displacement 64, and one copy starting at as type map.	39 40 41 42
	$type_0, disp_0), \dots, (type_{n-1}, disp_n)$		43
			44 45
array_of	displacements argument. The	blocklengths argument and D be the newly created datatype has $n \cdot \sum_{i=0}^{\text{count}-1} B[i]$ entries:	46
		$ppe_{n-1}, disp_{n-1} + D[0] \cdot ex), \dots,$	47 48

1	(ty)	$pe_0, disp_0 + (D[0] + B[0] - 1) \cdot$	$ex), \ldots, (type_{n-1}, disp_{n-1} + (D[0] + B[0] - 1) \cdot ex), \ldots,$
2 3	(ty)	$pe_0, disp_0 + D[count-1] \cdot ex), \dots$	$., (type_{n-1}, disp_{n-1} + D[count-1] \cdot ex), \ldots,$
4 5	(ty)	$be_0, disp_0 + (D[count-1] + B[count-1]]$	$punt-1] - 1) \cdot ex), \dots,$
6 7	(typ)	$pe_{n-1}, disp_{n-1} + (D[count-1] + $	$-B[count-1] - 1) \cdot ex)\}.$
8 9 10			unt, blocklength, stride, oldtype, newtype) is equivalent nt, B, D, oldtype, newtype) where
11	D[j]	$= j \cdot stride, \ j = 0, \dots, count$	-1,
12 13	and		
1415	B[j]	= blocklength, $j = 0, \dots, cou$	nt-1.
16 17 18 19 20	ified in b	PE_INDEXED, except that blo ytes, rather than in multiples	
21 22	MPI_TYF	<sup>2</sup> E_CREATE_HINDEXED(cour oldtype, newtype)	nt, array_of_blocklengths, array_of_displacements,
23 24 25 26	IN	count	number of blocks — also number of entries in array_of_displacements and array_of_blocklengths (non- negative integer)
27 28	IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)
29	IN	array_of_displacements	byte displacement of each block (array of integer)
30 31	IN	oldtype	old datatype (handle)
32	OUT	newtype	new datatype (handle)
33 34 35 36	int MPI_	• 1	<pre>count, const int array_of_blocklengths[], uy_of_displacements[], MPI_Datatype oldtype, rpe)</pre>
37 38	MPI_Type	e_create_hindexed(count,	•
39 40		EGER, INTENT(IN) :: coun	ents, oldtype, newtype, ierror) t, array_of_blocklengths(count)
41		GER(KIND=MPI_ADDRESS_KIN array_of_displacements(c	
42 43	TYPE	E(MPI_Datatype), INTENT(I	
44		E(MPI_Datatype), INTENT(O	• •
45	INTE	EGER, OPTIONAL, INTENT(OU	T) :: ierror
46	MPI_TYPE	E_CREATE_HINDEXED(COUNT,	
47 48	- <b>.</b>		NTS, OLDTYPE, NEWTYPE, IERROR)
	TNLF	GER COUNT, AKKAY_OF_BLUC	KLENGTHS(*), OLDTYPE, NEWTYPE, IERROR

INT	TEGER(KIND=MPI_ADDRESS_KINI	<pre>D) ARRAY_OF_DISPLACEMENTS(*)</pre>	1
Ass	sume that <b>oldtype</b> has type map	,	2 3
{(	$type_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$(n-1)\},$	4
	with extent $ex$ . Let B be the array_of_blocklengths argument and D be the		
	-	e newly created datatype has a type map with $n \cdot$	6
	$^{-1}$ B[i] entries:		7
{(	$type_0, disp_0 + D[0]), \dots, (type_{n-1})$	$-1, disp_{n-1} + D[0]), \ldots,$	8 9
(t)	$ype_0, disp_0 + D[0] + (B[0] - 1)$ .	ex)	10
	$(type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1) \cdot ex), \dots,$		11
			12
(t)	$ype_0, disp_0 + D[count-1]), \dots, (t)$	$ype_{n-1}, disp_{n-1} + D[count-1]), \dots,$	13 14
(t)	$ype_0, disp_0 + D[count-1] + (B[cont-1])$	$unt-1]-1)\cdot ex),\ldots,$	14
(t)	$ype_{n-1}, disp_{n-1} + D[count-1] + D[count-1]$	$(B[count-1] - 1) \cdot ex)\}.$	16
			17
		me as $MPI_TYPE_INDEXED$ except that the block-	18
0		te are many codes using indirect addressing arising	19
		ocksize is always 1 (gather/scatter). The following at blocksize and arbitrary displacements.	20 21
convenio	ence function anows for constan	a bioexsize and arbitrary displacements.	22
			23
MPI_IY		K(count, blocklength, array_of_displacements, oldtype,	24
	newtype)		25 26
IN	count	length of array of displacements (non-negative integer)	20 27
IN	blocklength	size of block (non-negative integer)	28
IN	array_of_displacements	array of displacements (array of integer)	29
IN	oldtype	old datatype (handle)	30
OUT	newtype	new datatype (handle)	31 32
			32
int MP1	01	x(int count, int blocklength,	34
	const int array_oi_o MPI_Datatype *newty	<pre>displacements[], MPI_Datatype oldtype,</pre>	35
		•	36
MPI_Typ		<pre>int, blocklength, array_of_displacements,</pre>	37
TN	oldtype, newtype, i TEGER, INTENT(IN) :: count		38 39
TU	array_of_displacements(co	6	40
TYI	PE(MPI_Datatype), INTENT(IN		41
	PE(MPI_Datatype), INTENT(OU	• -	42
INT	TEGER, OPTIONAL, INTENT(OUT	[) :: ierror	43
MPI_TY	PE_CREATE_INDEXED_BLOCK(COU	JNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	44
	OLDTYPE, NEWTYPE, I		$45 \\ 46$
INT		ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,	47
	NEWTYPE, IERROR		48

CHAPTER 4. DATATYPES

12			YPE_CREATE_HINDEXED_BLOCK is identical to CK, except that block displacements in	
3	array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.			
4				
5 6 7	MPI_TYP	PE_CREATE_HINDEXED_BLO oldtype, newtype)	DCK(count, blocklength, array_of_displacements,	
8	IN	count	length of array of displacements (non-negative integer)	
9	IN	blocklength	size of block (non-negative integer)	
10 11	IN	array_of_displacements	byte displacement of each block (array of integer)	
12	IN	oldtype	old datatype (handle)	
13 14	OUT	newtype	new datatype (handle)	
15 16 17 18	int MPI_		ock(int count, int blocklength, ay_of_displacements[], MPI_Datatype oldtype, ype)	
<ol> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> <li>24</li> <li>25</li> <li>26</li> <li>27</li> <li>28</li> <li>29</li> <li>30</li> <li>31</li> <li>32</li> <li>33</li> <li>34</li> <li>35</li> </ol>	INTE INTE TYPE TYPE INTE MPI_TYPE INTE INTE Struct M generalize	oldtype, newtype, i GER, INTENT(IN) :: coun GER(KIND=MPI_ADDRESS_KIN array_of_displacements(c (MPI_Datatype), INTENT(I (MPI_Datatype), INTENT(O GER, OPTIONAL, INTENT(OU C_CREATE_HINDEXED_BLOCK(C OLDTYPE, NEWTYPE, I GER COUNT, BLOCKLENGTH, GER(KIND=MPI_ADDRESS_KIN MPI_TYPE_CREATE_STRUCT es MPI_TYPE_CREATE_HIND	<pre>ht, blocklength hD), INTENT(IN) :: count) N) :: oldtype NUT) :: newtype T) :: ierror COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,</pre>	
36 37 38 39 40 41 42 43 44 45 46 47 48	cations of	f different datatypes.		

MPI_TYF		array_of_blocklengths, array_of_displacements,	1 2
	array_of_types, newtyp	e)	3
IN	count	number of blocks (non-negative integer) — also num-	4
		ber of entries in arrays array_of_types,	5
		$array_of_displacements and array_of_blocklengths$	6
IN	array_of_blocklength	number of elements in each block (array of non-negative	7
		integer)	8
IN	array_of_displacements	byte displacement of each block (array of integer)	9
IN	array_of_types	type of elements in each block (array of handles to datatype objects)	10 11 12
OUT	newtype	new datatype (handle)	12
			14
int MPI_		<pre>ount, const int array_of_blocklengths[], </pre>	15 16
		<pre>ny_of_displacements[], array_of_types[], MPI_Datatype *newtype)</pre>	10
	const mri_Datatype	array_or_types[], Mrr_Datatype *newtype)	18
MPI_Type	_create_struct(count, ar	ray_of_blocklengths,	19
		ents, array_of_types, newtype, ierror)	20
		t, array_of_blocklengths(count)	21
LNTE	GER(KIND=MPI_ADDRESS_KIN		22
TVDE	array_of_displacements(c		23
	TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count) TYPE(MPI_Datatype), INTENT(OUT) :: newtype		
	GER, OPTIONAL, INTENT(OU		25
			26
MPI_TYPE	C_CREATE_STRUCT(COUNT, AR		27
тмтг		<pre>INTS, ARRAY_OF_TYPES, NEWTYPE, IERROR) KLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,</pre>	28 29
	IERROR	RLENGINS(*), ARRAI_OF_IIFES(*), NEWIIFE,	30
TNTF		D) ARRAY_OF_DISPLACEMENTS(*)	31
1111			32
Enomal	<b>46</b> Lot transl house trans a mo		33
Ехатріє	e 4.6 Let type1 have type m	ap,	34
{(do	$\texttt{ouble}, 0), (\texttt{char}, 8)\},$		35
with out o	r = 16 Let P (2, 1, 2) D (4)	16 26) and T (MPL FLOAT trans1 MPL CHAP)	36
	, , , ,	D, 16, 26), and $T = (MPI_FLOAT, type1, MPI_CHAR)$ . TRUCT(3, B, D, T, newtype) returns a datatype with	37
type map		Thoe (3, b, b, 1, newtype) returns a datatype with	38 39
∫( <b>f</b>	(1)	$16$ ), (char, 24), (char, 26), (char, 27), (char, 28)}.	40
<u> </u>		(0), (0), (0), (0), (0), (0), (0), (0),	41
,	ed by three copies of MPI_CH	rting at 0, followed by one copy of type1 starting at AR, starting at 26. (We assume that a float occupies	42 43 44
-	·	ypes argument, where T[i] is a handle to,	45
			46

$$typemap_i = \{(type_0^i, disp_0^i), \dots, (type_{n_i-1}^i, disp_{n_i-1}^i)\},\$$

47

 $^{48}$ 

1 2 3	array_of_o		y_of_blocklength argument and D be the et c be the count argument. Then the newly created $B[i] \cdot n_i$ entries:
4 5	$\{(ty$	$pe_0^0, disp_0^0 + D[0]), \dots, (typ)$	$e_{n_0}^0, disp_{n_0}^0 + D[0]), \dots,$
6 7	(typ	$e_0^0, disp_0^0 + D[0] + (B[0] - 1)$	$(1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0]-1) \cdot ex_0), \dots,$
8 9	(typ	$e_0^{C-1}, disp_0^{C-1} + D[c-1]), \dots$	$, (type_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1]), \dots,$
10 11	(typ	$e_0^{C-1}, disp_0^{C-1} + D[c-1] + (E)$	$B[\mathbf{c}-1]-1) \cdot ex_{\mathbf{c}-1}), \ldots,$
12 13	(typ	$e_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1]$	1] + (B[c-1]-1) · $ex_{c-1}$ )}.
14 15 16 17		o MPI_TYPE_CREATE_ST	HINDEXED(count, B, D, oldtype, newtype) is equivalent RUCT(count, B, D, T, newtype), where each entry of T
18 19 20	4.1.3 Sı	ıbarray Datatype Construc	tor
21 22 23	MPI_TYP	E_CREATE_SUBARRAY(no order, oldtype, newty	dims, array_of_sizes, array_of_subsizes, array_of_starts, pe)
24	IN	ndims	number of array dimensions (positive integer)
25 26	IN	array_of_sizes	number of elements of type oldtype in each dimension of the full array (array of positive integers)
27 28 29	IN	array_of_subsizes	number of elements of type oldtype in each dimension of the subarray (array of positive integers)
30 31	IN	array_of_starts	starting coordinates of the subarray in each dimension (array of non-negative integers)
32	IN	order	array storage order flag (state)
33	IN	oldtype	array element datatype (handle)
34 35	OUT	newtype	new datatype (handle)
36 37 38 39	int MPI_	const int array_o	nt ndims, const int array_of_sizes[], f_subsizes[], const int array_of_starts[], tatype oldtype, MPI_Datatype *newtype)
40 41 42 43 44 45 46 47	INTE TYPE INTE	array_of_starts, GER, INTENT(IN) :: nd array_of_subsizes(ndim (MPI_Datatype), INTENT (MPI_Datatype), INTENT GER, OPTIONAL, INTENT(	(OUT) :: newtype DUT) :: ierror
48	MPI_TYPE	_CREATE_SUBARRAY(NDIMS	, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,

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

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

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

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

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

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

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

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

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

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

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

newtype =	Subarray( $ndims$ , { $size_0, size_1, \ldots, size_{ndims-1}$ },	36
	$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$	37
		38
	$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	39

Let the typemap of oldtype have the form:

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

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

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

1  $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

2223

 $^{24}$ 

25

2627

28

29

30 31

32 33

34

35

38 39 40

41

4243

44

45

46

47

1	
2 3	$Subsympt(1 \mid size) \mid subsize) \mid start) \tag{4.2}$
4	Subarray $(1, \{size_0\}, \{subsize_0\}, \{start_0\}, $ $\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\})$ (4.2)
5	
6	$= \{(lb_marker, 0), \\ (lb_marker, 0), \\ ($
7	$(type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex),$
8	$(type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1},$
9	$disp_{n-1} + (start_0 + 1) \times ex), \dots$
10 11	$(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \dots,$
12	$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$
13	$(ub\_marker, size_0  imes ex) \}$
14	
15	Subarray( $ndims$ , { $size_0, size_1, \dots, size_{ndims-1}$ }, (4.3)
16	$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$
17 18	$\{start_0, start_1, \ldots, start_{ndims-1}\}, oldtype\}$
19	= Subarray( $ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\},\$
20	$\{subsize_1, subsize_2, \dots, subsize_{ndims-1}\},\$
21	$\{start_1, start_2, \dots, start_{ndims-1}\},\$
22	$(start_1, start_2, \dots, start_{naims=1}),$ Subarray $(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$
23	$Subarray(1, \{size0\}, \{subsize0\}, \{subrid(i)\}, (outype))$
24	Subsymptot(redimes(sites))))))))))))))))))))))))))))))))))))
25 26	Subarray( $ndims$ , { $size_0, size_1, \dots, size_{ndims-1}$ }, (4.4)
27	$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$
28	$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$
29	$= \text{Subarray}(ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\},\$
30	$\{subsize_0, subsize_1, \ldots, subsize_{ndims-2}\},\$
31	$\{start_0, start_1, \dots, start_{ndims-2}\},\$
32 33	$Subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype))$
33 34	For an anomaly use of MDI TYPE CREATE SURAPRAY in the context of L/O see See
35	For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 13.9.2.
36	
37 38	4.1.4 Distributed Array Datatype Constructor
39	The distributed array type constructor supports HPF-like [42] data distributions. However,
40	unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.
41	
42	Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary flotunes are greated by having around a follows.
43	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
44 45	set appropriately). These filetypes (along with identical disp and etype) are then used
45 46	to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 13.1.1
47	and Section 13.3. Using this view, a collective data access operation (with identical
48	offsets) will yield an HPF-like distribution pattern. (End of advice to users.)

MPI_		rank, ndims, array_of_gsizes, array_of_distribs, of_psizes, order, oldtype, newtype)	1 2
IN	size	size of process group (positive integer)	3
IN	rank	rank in process group (non-negative integer)	4 5
IN	ndims	number of array dimensions as well as process grid dimensions (positive integer)	6 7
IN	array_of_gsizes	number of elements of type oldtype in each dimension of global array (array of positive integers)	8 9
IN	array_of_distribs	distribution of array in each dimension (array of state)	10 11
IN	array_of_dargs	distribution argument in each dimension (array of pos- itive integers)	12 13
IN	array_of_psizes	size of process grid in each dimension (array of positive integers)	14 15
IN	order	array storage order flag (state)	16 17
IN	oldtype	old datatype (handle)	18
OUT	Г newtype	new datatype (handle)	19 20
int M	const int array_of	size, int rank, int ndims, _gsizes[], const int array_of_distribs[], _dargs[], const int array_of_psizes[], atype oldtype, MPI_Datatype *newtype)	22 23 24 25
I T T	array_of_distribs, oldtype, newtype, NTEGER, INTENT(IN) :: siz	e, rank, ndims, array_of_gsizes(ndims), ), array_of_dargs(ndims), order IN) :: oldtype DUT) :: newtype	26 27 28 29 30 31 32 33 34
I	ARRAY_OF_DISTRIBS, OLDTYPE, NEWTYPE, INTEGER SIZE, RANK, NDIMS, ARRAY_OF_DARGS(*), ARRAY IERROR	NK, NDIMS, ARRAY_OF_GSIZES, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER, IERROR) ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*), Y_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, can be used to generate the datatypes corresponding	35 36 37 38 39 40 41 42
		nsional array of oldtype elements onto an	43

ndims-dimensional grid of logical processes. Unused dimensions of array\_of\_psizes should be set to 1. (See Example 4.7.) For a call to MPI\_TYPE\_CREATE\_DARRAY to be correct, the equation  $\prod_{i=0}^{ndims-1} array_of_psizes[i] = size$  must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies.

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

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

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

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

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

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

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

#### Address and Size Functions 4.1.5

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

The address of a location in memory can be found by invoking the function MPI\_GET\_ADDRESS. The relative displacement between two absolute addresses can be calculated with the function MPI\_AINT\_DIFF. A new absolute address as sum of an absolute base address and a relative displacement can be calculated with the function MPI\_AINT\_ADD. To ensure portability, arithmetic on absolute addresses should not be performed with the intrinsic operators "-" and "+". See also Sections 2.5.6 and 4.1.12 on pages 16 and 117.

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

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

1

 $\mathbf{2}$ 

3

4

56

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

2627

28 29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

4546

47

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

			1
	NT_ADD(base,		2
IN	base	base address (integer)	3
IN	disp	displacement (integer)	4
			5
MPI_Ain	t MPI_Aint_a	dd(MPI_Aint base, MPI_Aint disp)	6
TNTEGER	(KIND=MPT AD	DRESS_KIND) MPI_Aint_add(base, disp)	7
		I_ADDRESS_KIND), INTENT(IN) :: base, disp	8
			9
		DRESS_KIND) MPI_AINT_ADD(BASE, DISP) I_ADDRESS_KIND) BASE, DISP	10 11
MD		produces a new MPI_Aint value that is equivalent to the sum of	12
	-	ments, where base represents a base address returned by a call to	13
	. –	and disp represents a signed integer displacement. The resulting ad-	14
		he process that generated <b>base</b> , and it must correspond to a location	15 16
	e	renced by base, as described in Section 4.1.12. The addition is per-	16
formed i	n a manner tha	t results in the correct MPI_Aint representation of the output address,	18
as if the	process that o	riginally produced <b>base</b> had called:	19
MDT Cot	addrogg((ch	ar *) base + disp, &result);	20
III I_Get		ar */ base + disp, &result/,	21
			22
		2ddr()	23
	NT_DIFF(addr1		24
IN	addr1	minuend address (integer)	25
IN	addr2	subtrahend address (integer)	26
			27
MPI_Ain	t MPI_Aint_d	iff(MPI_Aint addr1, MPI_Aint addr2)	28 29
INTEGER	.(KIND=MPI_AD	DRESS_KIND) MPI_Aint_diff(addr1, addr2)	30
INT	EGER(KIND=MP	I_ADDRESS_KIND), INTENT(IN) :: addr1, addr2	31
тмтесер	(KIND-MDT AD	DRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)	32
		I_ADDRESS_KIND) ADDR1, ADDR2	33
			34
	-	produces a new MPI_Aint value that is equivalent to the difference	35
		Ir2 arguments, where addr1 and addr2 represent addresses returned	36 37
e e		ADDRESS. The resulting address is valid only at the process that ddr2, and addr1 and addr2 must correspond to locations in the same	38
-		cess, as described in Section 4.1.12. The difference is calculated in	39
	-	in the signed difference from addr1 to addr2, as if the process that	40
		e addresses had called (char *) addr1 - (char *) addr2 on the	41
		ed to MPI_GET_ADDRESS.	42
		liary functions provide useful information on derived datatypes.	43
			44
			45
			46 47

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

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

37 38

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

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

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

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

In general, if

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

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

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

#### Unofficial Draft for Comment Only

 $\mathbf{5}$ 

 $^{31}$ 

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

Example: Instead of

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

one should define

END TYPE

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

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

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

- 4.1.7 Extent and Bounds of Datatypes

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

44	IN	datatype	data type to get information on (handle)
45 46	OUT	lb	lower bound of datatype (integer)
47	OUT	extent	extent of datatype (integer)
48			

### 4.1. DERIVED DATATYPES

int	MPI_Type_get_extent(MPI_D MPI_Aint *extent	atatype datatype, MPI_Aint *1b, ;)	1 $2$
MPI_	Type_get_extent(datatype, TYPE(MPI_Datatype), INTEN INTEGER(KIND=MPI_ADDRESS_ INTEGER, OPTIONAL, INTENT	T(IN) :: datatype KIND), INTENT(OUT) :: lb, extent	3 4 5 6 7
MPI_	TYPE_GET_EXTENT(DATATYPE, INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_		8 9 10 11 12
MPI.	_TYPE_GET_EXTENT_X(data	type, lb, extent)	13
IN	datatype	datatype to get information on (handle)	14 15
οι		lower bound of datatype (integer)	16
OL		extent of datatype (integer)	17 18
int	MPI_Type_get_extent_x(MPI MPI_Count *exter	_Datatype datatype, MPI_Count *1b, nt)	19 20 21
MPI_	Type_get_extent_x(datatyp	e, lb, extent, ierror)	22
_	TYPE(MPI_Datatype), INTEN		23
		ND), INTENT(OUT) :: lb, extent	24
	INTEGER, OPTIONAL, INTENT	(OUT) :: ierror	25 26
MPI_	TYPE_GET_EXTENT_X(DATATYP	E, LB, EXTENT, IERROR)	20
	INTEGER DATATYPE, IERROR		28
	INTEGER(KIND=MPI_COUNT_KI	ND) LB, EXTENT	29
	For both functions, if either C if the parameter is too small	the extent of datatype (as defined in Equation 4.1). OUT parameter cannot express the value to be returned to hold the output value), it is set to MPI_UNDEFINED. extent of a datatype, using lower bound and upper bound	30 31 32 33
marl		er the stride of successive datatypes that are replicated	34
by d	atatype constructors, or are re	plicated by the <b>count</b> argument in a send or receive call.	35
			$\frac{36}{37}$
MPI.	TYPE_CREATE_RESIZED(old	ltype, lb, extent, newtype)	38
IN	oldtype	input datatype (handle)	39
IN	lb	new lower bound of datatype (integer)	40
IN	extent	new extent of datatype (integer)	41 42
			42
οι	IT newtype	output datatype (handle)	44
in+	MPT Tune create resized (M	PI_Datatype oldtype, MPI_Aint lb,	45
1110		MPI_Datatype *newtype)	46
MPI_		pe, lb, extent, newtype, ierror)	47 48

1	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
2	TYPE(MPI_Datatype), INTENT(IN) :: oldtype
3	TYPE(MPI_Datatype), INTENT(OUT) :: newtype
4	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5	
6	MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
7	INTEGER OLDTYPE, NEWTYPE, IERROR
8	INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
9	Returns in <b>newtype</b> a handle to a new datatype that is identical to <b>oldtype</b> , except that
10	the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb
11	+ extent. Any previous lb and ub markers are erased, and a new pair of lower bound and
12	upper bound markers are put in the positions indicated by the lb and extent arguments.
13	
14	
15	count > 1, and when used in the construction of new derived datatypes.
16	410 True Federal of Detation of
17	4.1.8 True Extent of Datatypes
18	Suppose we implement gather (see also Section $5.5$ ) as a spanning tree implemented on
19	top of point-to-point routines. Since the receive buffer is only valid on the root pro-
20	
21	, , , , , , , , , , , , , , , , , , , ,
14 15 16 17 18 19 20	<ul> <li>This affects the behavior of the datatype when used in communication operations, with count &gt; 1, and when used in the construction of new derived datatypes.</li> <li>4.1.8 True Extent of Datatypes</li> <li>Suppose we implement gather (see also Section 5.5) as a spanning tree implemented or</li> </ul>

ate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent, for example by using MPI\_TYPE\_CREATE\_RESIZED. The functions MPI\_TYPE\_GET\_TRUE\_EXTENT 23and MPI\_TYPE\_GET\_TRUE\_EXTENT\_X are provided which return the true extent of the datatype.

```
MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)
```

```
IN
                datatype
                                            datatype to get information on (handle)
29
30
       OUT
                true_lb
                                            true lower bound of datatype (integer)
^{31}
       OUT
                true_extent
                                            true size of datatype (integer)
32
33
     int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
34
                    MPI_Aint *true_extent)
35
36
     MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
41
         INTEGER DATATYPE, IERROR
42
         INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
43
44
45
46
47
48
```

22

24

252627

MPI_TYF	PE_GET_TRUE_EXTEN	NT_X(datatype, true_lb, true_extent)
IN	datatype	datatype to get information on (handle)
OUT	true_lb	true lower bound of datatype (integer)
OUT	- true_extent	true size of datatype (integer)
001		er de bize er datadype (integer)
int MPI_	Type_get_true_exter	nt_x(MPI_Datatype datatype, MPI_Count *true_lb,
_	MPI_Count *tr	
MPT Type	get true extent x	(datatype, true_lb, true_extent, ierror)
	-	<pre>IENT(IN) :: datatype</pre>
INTE	EGER(KIND=MPI_COUNT_	_KIND), INTENT(OUT) :: true_lb, true_extent
INTE	EGER, OPTIONAL, INTH	ENT(OUT) :: ierror
MPI_TYPE	C_GET_TRUE_EXTENT_X	(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
INTE	EGER DATATYPE, IERRO	)R
INTE	EGER(KIND=MPI_COUNT_	_KIND) TRUE_LB, TRUE_EXTENT
true_	lb returns the offset of	the lowest unit of store which is addressed by the datatype,
i.e., the l	ower bound of the cor	responding typemap, ignoring explicit lower bound mark-
		ie size of the datatype, i.e., the extent of the correspond-
0 0 *	, o o i	ower bound and upper bound markers, and performing no
rounding	for alignment. If the t	ypemap associated with datatype is
Typ	$pemap = \{(type_0, disp_0)\}$	$),\ldots,(type_{n-1},disp_{n-1})\}$
Then		
tru	$e\_lb(Typemap) = min_{j}$	$_{j}\{disp_{j} : type_{j} \neq lb_{marker}, ub_{marker}\},$
tru	$e \ ub(Tupemap) = max$	$x_i \{ disp_i + sizeof(type_i) : type_i \neq lb_marker, ub_marker \},$
		$\int \left( \cos \beta \right) + \sin \left( -\frac{\beta}{\beta} + \frac{\beta}{\beta} \right) + \frac{\beta}{\beta} + \frac$
and		
tru	e extent(Tupemap) =	$true\_ub(Typemap) - true\_lb(typemap).$
•	_	ith the definitions in Section $4.1.6$ and Section $4.1.7$ , which
	the function MPI_TYP	nimum number of bytes of memory necessary to hold a
	, uncompressed.	minum number of bytes of memory necessary to note a
° .		er OUT parameter cannot express the value to be returned
		all to hold the output value), it is set to MPI_UNDEFINED.
4.1.9 C	ommit and Free	
A datatv	pe object has to be <b>co</b>	<b>committed</b> before it can be used in a communication. As
-		uctors, uncommitted and also committed datatypes can be
used. Th	ere is no need to comm	nit basic datatypes. They are "pre-committed."

 $^{48}$ 

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

## MPI\_TYPE\_FREE(DATATYPE, IERROR) INTEGER DATATYPE, IERROR

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

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather then copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (End of advice to implementors.)

# 4.1.10 Duplicating a Datatype

MPI\_TYPE\_DUP(oldtype, newtype)

IN	oldtype	datatype (handle)
OUT	newtype	copy of $oldtype\xspace$ (handle)

<pre>int MPI_Type_dup(MPI_Datatype oldtype,</pre>	MPI_Datatype *newtype)
<pre>MPI_Type_dup(oldtype, newtype, ierror)     Type_f(vpi_p)     Type_f(vpi_p)</pre>	
TYPE(MPI_Datatype), INTENT(IN) ::	oldtype
TYPE(MPI_Datatype), INTENT(OUT) ::	newtype

MPI\_TYPE\_DUP(OLDTYPE, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR

INTEGER, OPTIONAL, INTENT(OUT) ::

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

ierror

# 4.1.11 Use of General Datatypes in Communication

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

1

 $\mathbf{2}$ 

3

4

5

6

7

8 9

10

11

12

13

14

15 16

17 18 19

30

31

32 33

34

35

36

37

38

39

40

41 42

43

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

Each of the sends matches any of the receives.

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

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

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

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

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

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

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

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

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

- MPI\_GET\_ELEMENTS\_X(STATUS, DATATYPE, COUNT, IERROR)

Unofficial Draft for Comment Only

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

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

# 4.1.12 Correct Use of Addresses

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

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

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

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

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

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

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

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $^{31}$ 

distinction is required, addresses are recognized as expressions that involve MPI\_BOTTOM. (*End of advice to implementors.*)

4.1.13 Decoding a Datatype

MPI datatype objects allow users to specify an arbitrary layout of data in memory. There 6 are several cases where accessing the layout information in opaque datatype objects would 7 be useful. The opaque datatype object has found a number of uses outside MPI. Further-8 more, a number of tools wish to display internal information about a datatype. To achieve 9 this, datatype decoding functions are provided. The two functions in this section are used 10 together to decode datatypes to recreate the calling sequence used in their initial defini-11 tion. These can be used to allow a user to determine the type map and type signature of a 12datatype. 13

 $14 \\ 15$ 

16

1

 $\mathbf{2}$ 

3 4

5

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

```
17
       IN
                  datatype
                                                datatype to access (handle)
18
19
        OUT
                  num_integers
                                                number of input integers used in the call constructing
20
                                                combiner (non-negative integer)
21
        OUT
                  num_addresses
                                                number of input addresses used in the call construct-
22
                                                ing combiner (non-negative integer)
23
        OUT
                  num_datatypes
                                                number of input datatypes used in the call construct-
^{24}
                                                ing combiner (non-negative integer)
25
26
       OUT
                  combiner
                                                combiner (state)
27
28
      int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
29
                      int *num_addresses, int *num_datatypes, int *combiner)
30
     MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
^{31}
```

MPI\_Type\_get\_envelope(datatype, num\_integers, num\_addresses, num\_datatypes
 combiner, ierror)
 TYPE(MPI\_Datatype), INTENT(IN) :: datatype
 INTEGER, INTENT(OUT) :: num\_integers, num\_addresses, num\_datatypes,

```
combiner
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

 MPI\_TYPE\_GET\_ENVELOPE(DATATYPE, NUM\_INTEGERS, NUM\_ADDRESSES, NUM\_DATATYPES, COMBINER, IERROR)
 INTEGER DATATYPE NUM INTEGERS NUM ADDRESSES NUM DATATYPES COMBINER

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

For the given datatype, MPI\_TYPE\_GET\_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-ofarguments 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.

48

35

36

40

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

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

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

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

Table 4.1: combiner values returned from MPI\_TYPE\_GET\_ENVELOPE

If combiner is MPI\_COMBINER\_NAMED then datatype is a named predefined datatype. The actual arguments used in the creation call for a datatype can be obtained using MPI\_TYPE\_GET\_CONTENTS.  $\overline{7}$ 

		ray_of_addresses, array_of_datatypes)
IN	datatype	datatype to access (handle)
IN	max_integers	number of elements in array_of_integers (non-negative integer)
IN	max_addresses	number of elements in array_of_addresses (non-neg integer)
IN	max_datatypes	number of elements in array_of_datatypes (non-neg integer)
OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)
OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)
OUT	array_of_datatypes	contains datatype arguments used in constructin datatype (array of handles)
INTE INTE INTE	GER, INTENT(OUT) :: a: GER(KIND=MPI_ADDRESS_K	<pre>x_integers, max_addresses, max_datatypes rray_of_integers(max_integers) IND), INTENT(OUT) ::</pre>
TYPE	uy_of_addresses(max_add: C(MPI_Datatype), INTENT CGER, OPTIONAL, INTENT(	(OUT) :: array_of_datatypes(max_datatypes
MPI_TYPE		, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYP , ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
ARRA	Y_OF_INTEGERS(*), ARRA	GERS, MAX_ADDRESSES, MAX_DATATYPES, Y_OF_DATATYPES(*), IERROR IND) ARRAY_OF_ADDRESSES(*)
datatype The	is a predefined named data values given for max_integer	s, max_addresses, and max_datatypes $must$ be at least
0		tegers, num_addresses, and num_datatypes, respective OPE for the same datatype argument.
	<i>ionale.</i> The arguments $ma$ or checking in the call. ( <i>En</i>	x_integers, max_addresses, and max_datatypes allow

CHAPTER 4. DATATYPES

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

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

Note that MPI\_TYPE\_GET\_CONTENTS can be invoked with a datatype argument that was constructed using MPI\_TYPE\_CREATE\_F90\_REAL, MPI\_TYPE\_CREATE\_F90\_INTEGER, or MPI\_TYPE\_CREATE\_F90\_COMPLEX (an unnamed predefined datatype). In such a case, an empty array\_of\_datatypes is returned.

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

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

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

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

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

The following defines what values are placed in each entry of the returned arrays <sup>45</sup> depending on the datatype constructor used for datatype. It also specifies the size of the <sup>46</sup> arrays needed which is the values returned by MPI\_TYPE\_GET\_ENVELOPE. In Fortran, <sup>47</sup> the following calls were made: <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 18

19

20

21

22

23 24

25 26

27

28 29

30

 $^{31}$ 

32

33

34

35

36

37 38

39

40

41

42

43

1	PARAMETER	(LARGE = 1000)				
2	INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR					
3		IND=MPI_ADDRESS_KIND)				
4						
		DATATYPE TYPE (NOT SHO				
5		YPE_GET_ENVELOPE(TYPE				
6	IF ((NI .G	T. LARGE) .OR. (NA .G	Γ. LAF	AGE) .OR. (ND .G.	Γ. LARGE)) THEN	
7	WRITE (*	*, *) "NI, NA, OR ND =	", NI	, NA, ND, &		
8		ED BY MPI_TYPE_GET_EN			IABGE = "IABGE	
9					LARGE - , LARGE	
		_ABORT(MPI_COMM_WORLD	, 99,	IERROR)		
10	ENDIF					
11	CALL MPI_T	YPE_GET_CONTENTS(TYPE	, NI,	NA, ND, I, A, D	, IERROR)	
12						
13	or in C the analogo	ous calls of:				
14						
15	#define LARGE 10					
	int ni, na, nd,	<pre>combiner, i[LARGE];</pre>				
16	MPI_Aint a[LARGE	:];				
17	MPI_Datatype typ	e. d[I_ARGE]:				
18		atype type (not shown)	) */			
19		01 01		0 1 • \		
20		elope(type, ∋, &na,				
		(na > LARGE)    (1				
21	fprintf(stde	err, "ni, na, or nd = 🕻	%d %d	%d returned by '	', ni, na, nd);	
22	fprintf(stde	err, "MPI_Type_get_enve	elope	is larger than I	LARGE = $d \in $	
23	-		-	0		
24			LARGE);			
	MPI_Abort(MPI_COMM_WORLD, 99);					
25		I_COMM_WORLD, 99);				
	};					
26	};	tents(type, ni, na, no	1, i,	a, d);		
26 27	}; MPI_Type_get_com	tents(type, ni, na, no			: J	
26	<pre>}; MPI_Type_get_com In the descript</pre>	tents(type, ni, na, no	case 1	name of arguments		
26 27	<pre>}; MPI_Type_get_com In the descript If combiner is</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED	case 1	name of arguments		
26 27 28	<pre>}; MPI_Type_get_com In the descript</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED	case 1	name of arguments		
26 27 28 29	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED	then i	name of arguments		
26 27 28 29 30	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS.	then i	name of arguments		
26 27 28 29 30 31 32	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS.	then i	name of arguments		
26 27 28 29 30 31 32 33	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument	then i	name of arguments t is erroneous to o Fortran location		
26 27 28 29 30 31 32	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then	then i	name of arguments t is erroneous to o		
26 27 28 29 30 31 32 33	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument oldtype	then i	name of arguments t is erroneous to o Fortran location		
26 27 28 29 30 31 32 33 33 34	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument oldtype , nd = 1.	$\frac{c \text{ case } c}{c}$	name of arguments t is erroneous to o Fortran location D(1)		
26 27 28 29 30 31 32 33 34 35	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument oldtype	$\frac{c \text{ case } f}{c}$	name of arguments t is erroneous to o Fortran location D(1)		
26 27 28 29 30 31 32 33 34 35 36	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0</pre>	<pre>itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU</pre>	$\frac{c \text{ case } f}{c}$	name of arguments t is erroneous to o Fortran location D(1)		
26 27 28 29 30 31 32 33 34 35 36 37 38	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0</pre>	itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP ther Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU Constructor argument	$\frac{c \text{ case } r}{c}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location		
26 27 28 29 30 31 32 33 34 35 36 37 38 39	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0</pre>	itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU	$\frac{c \text{ case } r}{d \log t}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1)		
26 27 28 30 31 32 33 34 35 36 37 38 39 40	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0</pre>	itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP ther Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU Constructor argument	$\frac{c \text{ case } r}{c}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location		
26 27 28 29 30 31 32 33 34 35 36 37 38 39	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0 If combiner is</pre>	itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU Constructor argument count oldtype	$\frac{c \text{ case } r}{d \log t}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1)		
26 27 28 30 31 32 33 34 35 36 37 38 39 40	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0 If combiner is</pre>	tents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then $\overline{Constructor argument}$ $\overline{oldtype}$ , nd = 1. MPI_COMBINER_CONTIGU $\overline{Constructor argument}$ $\overline{count}$ oldtype , nd = 1.	$\frac{c}{d[0]}$ $\frac{c}{d[0]}$ $\frac{c}{i[0]}$ $\frac{c}{d[0]}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1)		
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0 If combiner is</pre>	itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU Constructor argument count oldtype	$\frac{c}{d[0]}$ $\frac{c}{d[0]}$ $\frac{c}{i[0]}$ $\frac{c}{d[0]}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1)		
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0 If combiner is</pre>	<pre>Atents(type, ni, na, no bions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP ther Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU Constructor argument count oldtype , nd = 1. MPI_COMBINER_VECTOR</pre>	$\frac{c \text{ case } f}{d (0)}$ $\frac{C}{d [0]}$ $\frac{C}{d [0]}$ $\frac{C}{d [0]}$ $\frac{C}{d [0]}$ $\frac{C}{d [0]}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1) D(1)		
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0 If combiner is</pre>	<pre>itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP ther Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU Constructor argument count oldtype , nd = 1. MPI_COMBINER_VECTOR Constructor argument</pre>	$\frac{C}{d[0]}$ $\frac{C}{d[0]}$ $\frac{C}{d[0]}$ $\frac{C}{d[0]}$ $\frac{C}{d[0]}$ $\frac{C}{d[0]}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1) D(1) Fortran location		
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0 If combiner is</pre>	itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU Constructor argument count oldtype , nd = 1. MPI_COMBINER_VECTOR Constructor argument count count	$\frac{c}{d[0]}$ $\frac{c}{d[0]}$ $\frac{c}{i[0]}$ $\frac{c}{i[0]}$ $\frac{c}{i[0]}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1) D(1) Fortran location I(1)		
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0 If combiner is</pre>	<pre>itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP ther Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU Constructor argument count oldtype , nd = 1. MPI_COMBINER_VECTOR Constructor argument</pre>	$\frac{C}{d[0]}$ $\frac{C}{d[0]}$ $\frac{C}{d[0]}$ $\frac{C}{d[0]}$ $\frac{C}{d[0]}$ $\frac{C}{d[0]}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1) D(1) Fortran location		
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0 If combiner is</pre>	itents(type, ni, na, no ions that follow, the lower MPI_COMBINER_NAMED CONTENTS. MPI_COMBINER_DUP then Constructor argument oldtype , nd = 1. MPI_COMBINER_CONTIGU Constructor argument count oldtype , nd = 1. MPI_COMBINER_VECTOR Constructor argument count count	$\frac{c \text{ case } i}{d [0]}$ $\frac{C}{d[0]}$ $\frac{C}{i[0]}$ $\frac{C}{i[0]}$ $\frac{C}{i[0]}$ $\frac{C}{i[1]}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1) D(1) Fortran location I(1)		
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	<pre>}; MPI_Type_get_com In the descript If combiner is MPI_TYPE_GET_C If combiner is and ni = 0, na = 0 If combiner is</pre>	attents(type, ni, na, noattents(type, ni, na, noattents(type, ni, na, noMPI_COMBINER_NAMEDCONTENTS.MPI_COMBINER_DUP then $constructor argument$ oldtypeattentsattentsConstructor argumentcountoldtypeattentsattentsConstructor argumentcountoldtypeattentsattentsConstructor argumentcountoldtypeattentscountblocklength	$\frac{c}{d[0]}$ $\frac{c}{d[0]}$ $\frac{c}{i[0]}$ $\frac{c}{i[0]}$ $\frac{c}{i[0]}$	name of arguments t is erroneous to o Fortran location D(1) nen Fortran location I(1) D(1) Fortran location I(1) I(1) I(2)		

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

If combiner is MPI\_COMBINER\_HVECTOR then

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

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

If combiner is  $\mathsf{MPI\_COMBINER\_INDEXED}$  then

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

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

If combiner is MPI\_COMBINER\_HINDEXED then

Constructor argument	С	Fortran location
count	i[0]	I(1)
$array_{of_blocklengths}$	i[1] to i[i[0]]	I(2) to $I(I(1)+1)$
$array_of_displacements$	a[0] to $a[i[0]-1]$	A(1) to $A(I(1))$
oldtype	d[0]	$\mathrm{D}(1)$

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

If combiner is MPI\_COMBINER\_INDEXED\_BLOCK then

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

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

If combiner is  $MPI_COMBINER_HINDEXED_BLOCK$  then

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

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

If combiner is  $\mathsf{MPI\_COMBINER\_STRUCT}$  then

 $\mathbf{2}$ 

 $^{31}$ 

			C		· ·
	Constructor arg	Jument	C ;[0]	Fortran loca	01011
	ount rray of blockl	ongtha	i[0]	I(1) I(2) to I(I(1))	1)
	rray_of_blockle rray_of_displace		[1] to $i[i[0]]$		,
	° *		0] to $a[i[0]-i]$	• • • •	. , ,
a	rray_of_types	α	0 to d[i[0]-	1  D(1)  to  D(1)	(1))
and $ni = count +$	-1, na = count, is MPI_COMBIN				
-	ctor argument	(		Fortran lo	ocation
ndims		i[(		I(1)	
array_of		i[1] to		I(2) to $I(I)$	· · ·
•	_subsizes		o i[2*i[0]]		, ,
array_of	_starts			I(2*I(1)+2) to	
order		i[3*i[(		I(3*I(1))	-
oldtype		d[	0]	D(1)	)
	is MPI_COMBIN	NER_DARRAY			1
	or argument	(		Fortran	
size		i[(	-	I	,
rank		i[]	-	I(2	,
ndims		i[2	-	I(:	,
array_of_	5	i[3] to i		I(4) to $I(4)$	
array_of_o		i[i[2]+3] to			
orror of	dargs	i 2*i 2 +3  te		I(2*I(3)+4) to	o I(3*I(3)+3
array_of_o					
array_of_]		i[3*i[2]+3] to			I(4*I(3)+3)
array_of_] order		$i[3^*i[2]+3]$ to $i[4^*i[2]$	2]+3]	I(4*I(	I(4*I(3)+3) (3)+4)
array_of_]		i[3*i[2]+3] to	2]+3]		I(4*I(3)+3) (3)+4)
$array_of_j$ order oldtype and ni = 4*ndim	psizes ns+4, $na = 0$ ,	i[3*i[2]+3] to i[4*i[2]+3] to define the set of the s	$\frac{2[+3]}{2}$ AL then $\frac{t C Fc}{i[0]}$	I(4*I( D( rtran location I(1)	I(4*I(3)+3) (3)+4)
$array_of_j$ order oldtype and ni = 4*ndim	psizes ns+4, $na = 0$ , r is MPI_COMBIN Construc	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA	2]+3] D] AL then t C Fc	I(4*I(; D(	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = 4*ndim If combiner	psizes ns+4, na = 0, r is MPI_COMBIN $ \frac{Construc}{p} \\ r $	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA	$\frac{2[+3]}{2}$ AL then $\frac{t C Fc}{i[0]}$	I(4*I( D( rtran location I(1)	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = 4*ndim If combiner and ni = 2, na =	psizes ns+4, na = 0, r is MPI_COMBIN $ \frac{Construc}{p} \\ r $	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA tor argumen	$\frac{2[+3]}{\text{AL then}}$ $\frac{\text{t C Fc}}{\text{i[0]}}$ $\frac{\text{i[1]}}{\text{i[1]}}$	I(4*I(5))     D(7)     I(1)     I(2)     I(1)     I(2)     I(2)	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = $4$ *ndim If combiner and ni = 2, na =	psizes ns+4, $na = 0$ , $ris MPI_COMBINConstrucpr= 0, nd = 0.is MPI_COMBIN$	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA tor argumen	$\frac{2]+3]}{AL \text{ then}}$ $\frac{t  C  Fc}{i[0]}$ $i[1]$ $MPLEX \text{ then}$	I(4*I(3)) $I(4*I(3))$ $I(1)$ $I(1)$ $I(2)$ $I(2)$	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = 4*ndim If combiner and ni = 2, na =	psizes ns+4, na = 0, r is MPI_COMBIN r r r r r r r r	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA tor argumen	$\frac{2[+3]}{\text{ML then}}$	I(4*I(; D( rtran location I(1) I(2)	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = $4$ *ndim If combiner and ni = 2, na =	psizes ns+4, na = 0, r is MPI_COMBIN r r = 0, nd = 0. is MPI_COMBIN r r r r	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA tor argumen	$\frac{2[+3]}{M}$ AL then $\frac{t C Fc}{i[0]}$ MPLEX then $\frac{t C Fc}{i[0]}$	I(4*I(3)) $I(1)$ $I(1)$ $I(2)$ $I(1)$ $I(1)$ $I(1)$ $I(1)$	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = $4$ *ndim If combiner and ni = 2, na =	psizes ns+4, na = 0, r is MPI_COMBIN r r r r r r r r	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA tor argumen	$\frac{2[+3]}{\text{ML then}}$	I(4*I(; D( rtran location I(1) I(2)	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = 4*ndim If combiner and ni = 2, na = If combiner	psizes ns+4, na = 0, r is MPI_COMBIN r = 0, nd = 0. is MPI_COMBIN r r	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA tor argumen	$\frac{2[+3]}{M}$ AL then $\frac{t C Fc}{i[0]}$ MPLEX then $\frac{t C Fc}{i[0]}$	I(4*I(3)) $I(1)$ $I(1)$ $I(2)$ $I(1)$ $I(1)$ $I(1)$ $I(1)$	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = 4*ndim If combiner and ni = 2, na = If combiner	psizes ns+4, na = 0, r is MPI_COMBIN $\overline{\frac{Construc}{p}}$ r = 0, nd = 0. is MPI_COMBIN $\overline{\frac{Construc}{p}}$ r = 0, nd = 0.	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA tor argumen	$\frac{2[+3]}{M}$ AL then $\frac{t C Fc}{i[0]}$ $\frac{t C Fc}{i[1]}$ $\frac{t C Fc}{i[0]}$ $\frac{t C Fc}{i[1]}$	I(4*I(3)) $I(1)$ $I(1)$ $I(2)$ $I(1)$ $I(1)$ $I(1)$ $I(1)$	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = 4*ndim If combiner and ni = 2, na = If combiner	psizes ns+4, na = 0, r is MPI_COMBIN r = 0, nd = 0. is MPI_COMBIN r r	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA tor argumen	$\frac{2[+3]}{M}$ AL then $\frac{t C Fc}{i[0]}$ $\frac{t C Fc}{i[1]}$ $\frac{t C Fc}{i[0]}$ $\frac{t C Fc}{i[1]}$	I(4*I(3)) $I(1)$ $I(1)$ $I(2)$ $I(1)$ $I(1)$ $I(1)$ $I(1)$	I(4*I(3)+3) (3)+4)
array_of_j order oldtype and ni = 4*ndim If combiner and ni = 2, na = If combiner	psizes ns+4, na = 0, r is MPI_COMBIN r r r r r r r r	i[3*i[2]+3] to i[4*i[2 d[ nd = 1. NER_F90_REA tor argumen	$\frac{2[+3]}{\text{ML then}}$ $\frac{\text{t C Fo}}{i[0]}$ $\frac{\text{MPLEX ther}}{i[0]}$ $\frac{\text{t C Fo}}{i[0]}$ $i[1]$ $\text{EGER then}$	I(4*I(3)) $I(1)$ $I(1)$ $I(2)$ $I(1)$ $I(1)$ $I(1)$ $I(1)$	I(4*I(3)+3) (3)+4)

	i = 1, na = 0, f combiner is N	nd = 0. IPI_COMBINER_RESIZED	then		1 2
		Constructor engineert	С	Fortran location	3 4
		Constructor argument lb	$\frac{C}{a[0]}$	$\frac{1}{A(1)}$	5
		extent	a[0] a[1]	A(1) A(2)	6
		oldtype	d[0]	$\mathbf{D}(1)$	7
		olutype	սլՍյ		8
and n	i = 0, na = 2,	nd = 1.			9
4.1.14	Examples				10 11
	·	bles illustrate the use of o	lerived	l datatypes.	12
	0 1				13 14
Exan	nple 4.13 Sen	d and receive a section o	of a 3D	) array.	14
					16
		100,100), e(9,9,9)			10
		slice, twoslice, three			18
		ND=MPI_ADDRESS_KIND)	lb, s	sizeofreal	19
	INTEGER sta	tus(MPI_STATUS_SIZE)			20
~					21
C		section a(1:17:2, 3:	:11, 2	2:10)	22
С	and store 1	t in e(:,:,:).			23
	CALL MDT CO	MM DANK/MDT COMM LIODI	· D		24
	CALL MPI_CU	MM_RANK(MPI_COMM_WORI	JD, my	fank, leff)	25
	CALL MDT TV	PE_GET_EXTENT(MPI_REA	י די	sizeofreel ierr)	26
	CALL MFI_II		чш, шк	, Sizeoffeat, leff)	27
С	create data	type for a 1D section	<b>.</b>		28
0		PE_VECTOR(9, 1, 2, MF		N. oneslice ierr)	29
		<u>11_v101010(0, 1, 2, 11</u>	T_10D1		30
С	create data	type for a 2D section	h		31
U		• -		00*sizeofreal, oneslice,	32
	0			e, ierr)	33
				,,	34
С	create data	type for the entire a	sectio	n	35
		• •		00*100*sizeofreal, twoslice,	36
	_			lce, ierr)	37
					38
	CALL MPI_TY	PE_COMMIT(threeslice,	ierr	r)	39
				slice, myrank, O, e, 9*9*9,	40
	_			), MPI_COMM_WORLD, status, ierr)	41
		_ , ,	•		42
_	_	- /			43
Exan	nple 4.14 Cop	by the (strictly) lower tri	angula	ar part of a matrix.	44
					45
					46
					47
					48

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

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

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

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

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

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

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

}

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

$\frac{1}{2}$	4.2 Pa	ck and Unpack	
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	tiguous da it, and uny described i The user s library dir for compar otherwise the receive Another u thus overr	ta. In these, the user explicitly packs it from a contiguous buf in Section 4.1, allow one, in m specifies the layout of the dat rectly accesses a noncontiguous tibility with previous libraries available in MPI. For instance e operation done on a later p se is that outgoing messages iding the system buffering po	provide pack/unpack functions for sending noncon- v packs data into a contiguous buffer before sending fer after receiving it. Derived datatypes, which are ost cases, to avoid explicit packing and unpacking. ta to be sent or received, and the communication is buffer. The pack/unpack routines are provided . Also, they provide some functionality that is not , a message can be received in several parts, where part may depend on the content of a former part. may be explicitly buffered in user supplied space, plicy. Finally, the availability of pack and unpack f additional communication libraries layered on top
17 18	MPI_PAC	۲(inbuf, incount, datatype, out	ouf, outsize, position, comm)
19	IN	inbuf	input buffer start (choice)
20	IN	incount	number of input data items (non-negative integer)
21 22	IN	datatype	datatype of each input data item (handle)
23	OUT	outbuf	output buffer start (choice)
24	IN	outsize	output buffer size, in bytes (non-negative integer)
25 26	INOUT	position	current position in buffer, in bytes (integer)
27 28	IN	comm	communicator for packed message (handle)
29 30	int MPI_F		nt incount, MPI_Datatype datatype, tsize, int *position, MPI_Comm comm)
31 32 33 34 35 36 37 38 39	TYPE( TYPE) INTEC TYPE( INTEC TYPE)	<pre>(inbuf, incount, datatype) (*), DIMENSION(), INTENT (*), DIMENSION() :: ou GER, INTENT(IN) :: incoun (MPI_Datatype), INTENT(IN) GER, INTENT(INOUT) :: pos (MPI_Comm), INTENT(IN) :: GER, OPTIONAL, INTENT(OUT)</pre>	<pre>itbuf nt, outsize ) :: datatype sition     comm</pre>
40 41 42	<type< td=""><td><pre>&gt; INBUF(*), OUTBUF(*)</pre></td><td>, OUTBUF, OUTSIZE, POSITION, COMM, IERROR) ISIZE, POSITION, COMM, IERROR</td></type<>	<pre>&gt; INBUF(*), OUTBUF(*)</pre>	, OUTBUF, OUTSIZE, POSITION, COMM, IERROR) ISIZE, POSITION, COMM, IERROR
43 44 45 46 47 48	space spec allowed in bytes, star	ified by <b>outbuf</b> and <b>outsize</b> . T MPI_SEND. The output buff	r specified by inbuf, incount, datatype into the buffer. The input buffer can be any communication buffer fer is a contiguous storage area containing outsize ngth is counted in <i>bytes</i> , not elements, as if it were f type MPI_PACKED).

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

MPI\_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm) inbuf IN input buffer start (choice) IN insize size of input buffer, in bytes (non-negative integer) INOUT position current position in bytes (integer) OUT outbuf output buffer start (choice) IN outcount number of items to be unpacked (integer) IN datatype datatype of each output data item (handle) IN communicator for packed message (handle) comm

MPI\_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm, ierror) TYPE(\*), DIMENSION(..), INTENT(IN) :: inbuf TYPE(\*), DIMENSION(..) :: outbuf INTEGER, INTENT(IN) :: insize, outcount INTEGER, INTENT(INOUT) :: position TYPE(MPI\_Datatype), INTENT(IN) :: datatype TYPE(MPI\_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 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 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

41

42

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

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

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

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

A message sent with any type (including MPI\_PACKED) can be received using the type
 MPI\_PACKED. Such a message can then be unpacked by calls to MPI\_UNPACK.

A packing unit (or a message created by a regular, "typed" send) can be unpacked into several successive messages. This is effected by several successive related calls to

MPI\_UNPACK, where the first call provides position = 0, and each successive call inputs the
 value of position that was output by the previous call, and the same values for inbuf, insize
 and comm.

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

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

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

46

38

39

40

41

42 43

- 40 47
- 48

1

2

3

4

5

6

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

MPI_PACK	_SIZE(incount, datatype, comr	n, size)	1
IN	incount	count argument to packing call (non-negative integer)	2 3
IN	datatype	datatype argument to packing call (handle)	4
IN	comm	communicator argument to packing call (handle)	5
OUT	size	upper bound on size of packed message, in bytes (non- negative integer)	6 7 8
int MPI_P	ack_size(int incount, MPI int *size)	_Datatype datatype, MPI_Comm comm,	9 10 11
MPI Pack	size(incount, datatype, c	comm. size. ierror)	12
	ER, INTENT(IN) :: incoun		13
TYPE(	MPI_Datatype), INTENT(IN)	:: datatype	14
TYPE(	MPI_Comm), INTENT(IN) ::	comm	15
INTEG	ER, INTENT(OUT) :: size		16
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	17
MPI PACK	SIZE(INCOUNT, DATATYPE, C	COMM. SIZE. IERROR)	18
	ER INCOUNT, DATATYPE, COM		19 20
			20 21
	•	atatype, comm, size) returns in size an upper bound ed by a call to MPI_PACK(inbuf, incount, datatype,	21
	-	e packed size of the datatype cannot be expressed	23
		SIZE sets the value of size to MPI_UNDEFINED.	24
			25
	-	per bound, rather than an exact bound, since the	26
		ack the message may depend on the context (e.g.,	27
first 1	message packed in a packing u	unit may take more space). (End of rationale.)	28
			29
Example	4.21 An example using MPI_	PACK.	30
int	<pre>position, i, j, a[2];</pre>		31
char	buff[1000];		32 33
onar	Surr[1000],		34
MPI_Comm_	rank(MPI_COMM_WORLD, &myr	ank);	35
	k == 0)		36
{			37
/* SE	NDER CODE */		38
			39
nosit	ion = 0		10

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

**Example 4.22** An elaborate example.

1

40

41

42

43

44

45

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

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

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

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

1 2

3

4

5 6 7

8

9 10 11

12

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

#### Canonical MPI PACK and MPI UNPACK 4.3

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

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

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

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

**Unofficial Draft for Comment Only** 

1

2

3

4

5

6

7 8

9

11

18

21

22 23

24

25

26

2728

29

30

 $^{31}$ 

32 33

34

35

36

MPI_PACK	_EXTERNAL(datarep, inbuf, i	ncount, datatype, outbuf, outsize, position)	1	
IN	datarep	data representation (string)	2 3	
IN	inbuf	input buffer start (choice)	4	
IN	incount	number of input data items (integer)	5	
IN	datatype	datatype of each input data item (handle)	6	
OUT	outbuf	output buffer start (choice)	7	
IN	outsize	output buffer size, in bytes (integer)	9	
INOUT	position	current position in buffer, in bytes (integer)	10 11	
int MPI_P		<pre>latarep[], const void *inbuf, int incount, e, void *outbuf, MPI_Aint outsize,</pre>	12 13 14 15	
MPI_Pack_	external(datarep, inbuf,	incount, datatype, outbuf, outsize,	16	
	position, ierror)		17 18	
	CHARACTER(LEN=*), INTENT(IN) :: datarep			
	TYPE(*), DIMENSION(), INTENT(IN) :: inbuf TYPE(*), DIMENSION() :: outbuf			
	INTEGER, INTENT(IN) :: incount			
	TYPE(MPI_Datatype), INTENT(IN) :: datatype			
	GER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize			
	GER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position GER, OPTIONAL, INTENT(OUT) :: ierror			
			26	
MPI_PACK_	POSITION, IERROR)	INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	27	
INTEG	ER INCOUNT, DATATYPE, IEF	ROR	28 29	
	INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION			
CHARACTER*(*) DATAREP			30 31	
<type> INBUF(*), OUTBUF(*)</type>				
			33	
MPI_UNPA	CK_EXTERNAL(datarep, inbu	f, insize, position, outbuf, outsize, position)	34 35	
IN	datarep	data representation (string)	36	
IN	inbuf	input buffer start (choice)	37	
IN	insize	input buffer size, in bytes (integer)	38 39	
INOUT	position	current position in buffer, in bytes (integer)	40	
OUT	outbuf	output buffer start (choice)	41	
IN	outcount	number of output data items (integer)	42	
IN	datatype	datatype of output data item (handle)	43 44	
			45	
int MPI_U	npack_external(const char	datarep[], const void *inbuf,	46	
		_Aint *position, void *outbuf,	47	
	int outcount, MPI_Datatype datatype)			

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

# Chapter 5

# **Collective Communication**

# 5.1 Introduction and Overview

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

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

1 2

3

4

• MPI\_SCAN, MPI\_ISCAN, MPI\_EXSCAN, MPI\_IEXSCAN: Scan across all members of a group (also called prefix) (Section 5.11, Section 5.11.2, Section 5.12.11, and Section 5.12.12).

One of the key arguments in a call to a collective routine is a communicator that 5defines the group or groups of participating processes and provides a context for the oper-6 ation. This is discussed further in Section 5.2. The syntax and semantics of the collective 7 operations are defined to be consistent with the syntax and semantics of the point-to-point 8 operations. Thus, general datatypes are allowed and must match between sending and re-9 ceiving processes as specified in Chapter 4. Several collective routines such as broadcast 10 and gather have a single originating or receiving process. Such a process is called the *root*. 11 Some arguments in the collective functions are specified as "significant only at root," and 12are ignored for all participants except the root. The reader is referred to Chapter 4 for 13 information concerning communication buffers, general datatypes and type matching rules, 14and to Chapter 6 for information on how to define groups and create communicators. 15

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

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

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

35 36

37

38

39

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

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

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

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

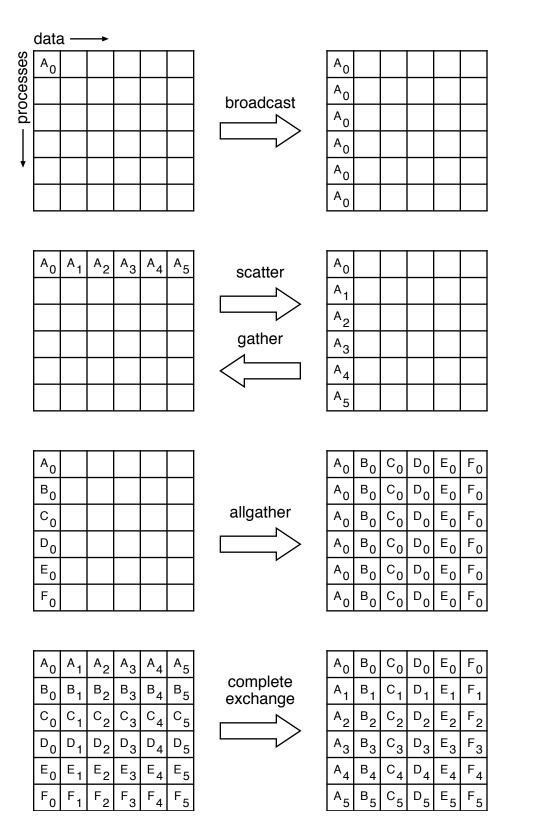


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data  $A_0$ , but after the broadcast all processes contain it.

 $\mathbf{2}$ 

 $^{31}$ 

 $^{41}$ 

 $44 \\ 45$ 

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

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

<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.

5.2 Communicator Argument

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

5.2.1 Specifics for Intracommunicator Collective Operations

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

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

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

6 7 8

9

10

11

12

17 18

19

28

29

1

2

3

4

5.2.2 Applying Collective Operations to Intercommunicators <sup>1</sup>
To understand how collective operations apply to intercommunicators, we can view most $^{2}_{3}$ MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [56]): 5
All-To-All All processes contribute to the result. All processes receive the result.
• MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV
• MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW
• MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER
• MPI_BARRIER, MPI_IBARRIER
All-To-One All processes contribute to the result. One process receives the result.
MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV     MPI_REDUCE, MPI_IREDUCE
One-To-All One process contributes to the result. All processes receive the result.
<ul> <li>MPI_BCAST, MPI_IBCAST</li> <li>MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV</li> <li>23</li> <li>24</li> <li>24</li> <li>25</li> </ul>
<b>Other</b> Collective operations that do not fit into one of the above categories.
• MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN 28
The data movement patterns of MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, and MPI_IEXSCAN do not fit this taxonomy. The application of collective communication to intercommunicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 5.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 5.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior. The following collective operations also apply to intercommunicators:
• MPI_BARRIER, MPI_IBARRIER 43
• MPI_BCAST, MPI_IBCAST 45
• MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV,
• MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV,

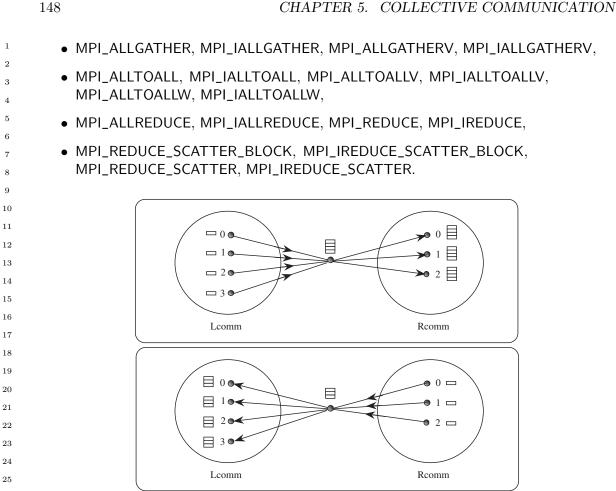


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

2930  $^{31}$ 

32

2627

28

#### 5.2.3 Specifics for Intercommunicator Collective Operations

All processes in both groups identified by the intercommunicator must call the collective 33 routine.

34 Note that the "in place" option for intracommunicators does not apply to intercom-35 municators since in the intercommunicator case there is no communication from a process 36 to itself. 37

For intercommunicator 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 45transfer 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

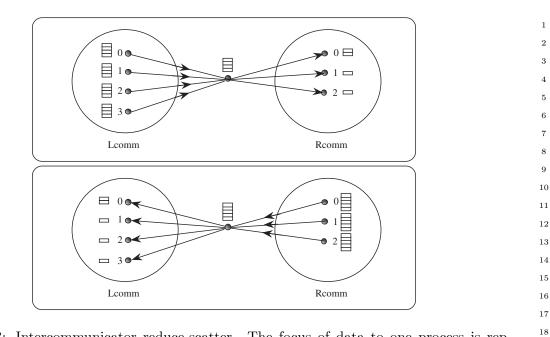


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

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

communicator (handle)

5.3	Barrier	Sync	hronization	

```
MPI_BARRIER(comm)
```

int MPI\_Barrier(MPI\_Comm comm)

```
MPI_Barrier(comm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR
```

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

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

CHAPTER 5. COLLECTIVE COMMUNICATION

54 Broadcast 1  $\mathbf{2}$ 3 4 MPI\_BCAST(buffer, count, datatype, root, comm) 5INOUT buffer 6 starting address of buffer (choice) 7 IN number of entries in buffer (non-negative integer) count 8 IN datatype data type of buffer (handle) 9 rank of broadcast root (integer) 10 IN root 11 IN comm communicator (handle) 1213 int MPI\_Bcast(void\* buffer, int count, MPI\_Datatype datatype, int root, 14 MPI\_Comm comm) 1516MPI\_Bcast(buffer, count, datatype, root, comm, ierror) TYPE(\*), DIMENSION(..) :: buffer 17 INTEGER, INTENT(IN) :: count, root 18 19TYPE(MPI\_Datatype), INTENT(IN) :: datatype TYPE(MPI\_Comm), INTENT(IN) :: comm 20INTEGER, OPTIONAL, INTENT(OUT) :: 21ierror 22MPI\_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR) 23<type> BUFFER(\*) 24INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR 2526If comm is an intracommunicator, MPI\_BCAST broadcasts a message from the process 27with rank root to all processes of the group, itself included. It is called by all members of 28the group using the same arguments for comm and root. On return, the content of root's 29buffer is copied to all other processes. 30 General, derived datatypes are allowed for datatype. The type signature of count,  $^{31}$ datatype on any process must be equal to the type signature of count, datatype at the root. 32 This implies that the amount of data sent must be equal to the amount received, pairwise 33 between each process and the root. MPI\_BCAST and all other data-movement collective 34routines make this restriction. Distinct type maps between sender and receiver are still 35 allowed. 36 The "in place" option is not meaningful here. 37 If comm is an intercommunicator, then the call involves all processes in the intercom-

<sup>38</sup> municator, but with one group (group A) defining the root processes in the intercom-<sup>39</sup> other group (group B) pass the same value in argument root, which is the rank of the root <sup>40</sup> in group A. The root passes the value MPI\_ROOT in root. All other processes in group A <sup>41</sup> pass the value MPI\_PROC\_NULL in root. Data is broadcast from the root to all processes <sup>42</sup> in group B. The buffer arguments of the processes in group B must be consistent with the <sup>43</sup> buffer argument of the root.

44

<sup>45</sup> 5.4.1 Example using MPI\_BCAST

47 The examples in this section use intracommunicators.

### Example 5.1

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

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

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

# 5.5 Gather

			16
MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)			
IN	sendbuf	starting address of send buffer (choice)	18
IN	sendcount	number of elements in send buffer (non-negative inte- ger)	19 20 21
IN	sendtype	data type of send buffer elements (handle)	22
OUT	recvbuf	address of receive buffer (choice, significant only at root)	23 24
IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)	25 26 27
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	28 29
IN	root	rank of receiving process (integer)	30
IN	comm	communicator (handle)	31 32
			32
int MPI_Ga	ther(const void* sendbuf	, int sendcount, MPI_Datatype sendtype,	34
	void* recvbuf, int re MPI_Comm comm)	ecvcount, MPI_Datatype recvtype, int root,	35 36
MPT Gather	(sendbuf, sendcount, sen	dtype, recvbuf, recvcount, recvtype,	37
	root, comm, ierror)	augpo, 10000ai, 100000aio, 10000jpo,	38
TYPE(*	), DIMENSION(), INTENT	(IN) :: sendbuf	39
TYPE(*	), DIMENSION() :: re	cvbuf	40
	R, INTENT(IN) :: sendco		41 42
	PI_Datatype), INTENT(IN)	V1 V1	42
	PI_Comm), INTENT(IN) ::		44
INTEGE	R, OPTIONAL, INTENT(OUT)	:: lerror	45
MPI_GATHER	MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,		
	ROOT, COMM, IERROR)		47
<type></type>	<type> SENDBUF(*), RECVBUF(*) 4</type>		

1 2

3

4

5

6

7

8 9

10

11 12 13

 $14 \\ 15$ 

1	
2	
3	

5

6

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

If comm is an intracommunicator, each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is  $as \ if$  each of the n processes in the group (including the root process) had executed a call to

7 8 9

> 10 11

12

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

and the root had executed  $\boldsymbol{n}$  calls to

MPI\_Recv(recvbuf+i· recvcount· extent(recvtype), recvcount, recvtype, i,...),

<sup>13</sup> where extent(recvtype) is the type extent obtained from a call to MPI\_Type\_get\_extent.

<sup>14</sup> An alternative description is that the n messages sent by the processes in the group <sup>15</sup> are concatenated in rank order, and the resulting message is received by the root as if by a <sup>16</sup> call to MPI\_RECV(recvbuf, recvcount·n, recvtype, ...).

17

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

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

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

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

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

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

<sup>34</sup> If comm is an intercommunicator, then the call involves all processes in the intercom-<sup>35</sup> municator, but with one group (group A) defining the root process. All processes in the <sup>36</sup> other group (group B) pass the same value in argument root, which is the rank of the root <sup>37</sup> in group A. The root passes the value MPI\_ROOT in root. All other processes in group A <sup>38</sup> pass the value MPI\_PROC\_NULL in root. Data is gathered from all processes in group B to <sup>39</sup> the root. The send buffer arguments of the processes in group B must be consistent with <sup>40</sup> the receive buffer argument of the root.

- 41
- 42 43
- 44
- 45
- 46
- 47
- 48

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, <sup>1</sup> comm) <sup>2</sup>			
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements in send buffer (non-negative inte- ger)	4 5 6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	address of receive buffer (choice, significant only at root)	8 9 10
IN	recvcounts	non-negative integer array (of length group size) con- taining the number of elements that are received from each process (significant only at root)	11 12 13
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to <b>recvbuf</b> at which to place the incoming data from process i (significant only at root)	14 15 16 17
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	18 19 20
IN	root	rank of receiving process (integer)	21
IN	comm	communicator (handle)	22
<pre>int MPI_Gatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm)</pre>			24 25 26
TYPE(* TYPE(* INTEGE TYPE(N TYPE(N	<pre>recvtype, root, comm, ), DIMENSION(), INTENT ), DIMENSION() :: re</pre>	<pre>(IN) :: sendbuf ecvbuf ount, recvcounts(*), displs(*), root     :: sendtype, recvtype     comm</pre>	27 28 29 30 31 32 33 34 35
<pre>MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,</pre>			38 39
MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count42of data from each process, since recvcounts is now an array. It also allows more flexibility43as to where the data is placed on the root, by providing the new argument, displs.44If comm is an intracommunicator, the outcome is as if each process, including the root45			42 43 44

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

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

 $46 \\ 47$ 

1 2	and the root executes $n$ receives,
3	MPI_Recv(recvbuf+displs[j]· extent(recvtype), recvcounts[j], recvtype, i,).
4 5	The data received from process j is placed into recvbuf of the root process beginning at
6	offset displs[j] elements (in terms of the recvtype).
7	The receive buffer is ignored for all non-root processes.
8	The type signature implied by sendcount, sendtype on process i must be equal to the
9	type signature implied by recvcounts[i], recvtype at the root. This implies that the amount
10	of data sent must be equal to the amount of data received, pairwise between each process
11	and the root. Distinct type maps between sender and receiver are still allowed, as illustrated
12	in Example 5.6.
13	All arguments to the function are significant on process root, while on other processes,
14	only arguments $sendbuf, sendcount, sendtype, root, \mathrm{and} comm \ \mathrm{are} \ \mathrm{significant}.$ The arguments
15	root and comm must have identical values on all processes.
16	The specification of counts, types, and displacements should not cause any location on
17	the root to be written more than once. Such a call is erroneous.
18	The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as
19	the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and
20	the contribution of the root to the gathered vector is assumed to be already in the correct
21	place in the receive buffer.
22	If comm is an intercommunicator, then the call involves all processes in the intercom-
23	municator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root
24	in group A. The root passes the value MPI_ROOT in root. All other processes in group A
25	pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to
26	the root. The send buffer arguments of the processes in group B must be consistent with
27 28	the receive buffer argument of the root.
29	
30	5.5.1 Examples using MPI_GATHER, MPI_GATHERV
31 32	The examples in this section use intracommunicators.
33	Example 5.2
34	Gather 100 ints from every process in group to root. See Figure 5.4.
35	Gather 100 mus nom every process in group to root. See Figure 0.4.
36	MPI_Comm comm;
37	int gsize, sendarray[100];
38	int root, *rbuf;
39	
40	<pre>MPI_Comm_size(comm, &amp;gsize);</pre>
41	<pre>rbuf = (int *)malloc(gsize*100*sizeof(int));</pre>
42	<pre>MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);</pre>
43	
44	
45	Example 5.3
46	Previous example modified — only the root allocates memory for the receive buffer.
47	
48	

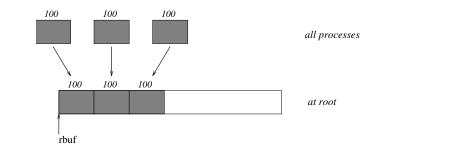


Figure 5.4: The root process gathers 100 ints from each process in the group.

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

### Example 5.4

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

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

### Example 5.5

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

 $^{24}$ 

```
100
                                   100
                                           100
1
\mathbf{2}
                                                                  all processes
3
4
                             100
                                    100
                                           100
5
                                                                  at root
6
7
                                     stride
                            rbuf
8
9
      Figure 5.5: The root process gathers 100 ints from each process in the group, each set is
10
      placed stride ints apart.
11
12
          MPI_Comm comm;
13
          int gsize,sendarray[100];
14
          int root, *rbuf, stride;
15
          int *displs,i,*rcounts;
16
17
          . . .
18
19
          MPI_Comm_size(comm, &gsize);
20
          rbuf = (int *)malloc(gsize*stride*sizeof(int));
21
          displs = (int *)malloc(gsize*sizeof(int));
22
          rcounts = (int *)malloc(gsize*sizeof(int));
23
          for (i=0; i<gsize; ++i) {</pre>
24
               displs[i] = i*stride;
25
               rcounts[i] = 100;
26
          }
27
          MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
28
                                                                                root, comm);
29
30
          Note that the program is erroneous if stride < 100.
^{31}
32
      Example 5.6
33
          Same as Example 5.5 on the receiving side, but send the 100 ints from the 0th column
34
      of a 100 \times 150 int array, in C. See Figure 5.6.
35
36
          MPI_Comm comm;
37
          int gsize, sendarray[100][150];
38
          int root, *rbuf, stride;
39
          MPI_Datatype stype;
40
          int *displs,i,*rcounts;
41
42
          . . .
43
44
          MPI_Comm_size(comm, &gsize);
45
          rbuf = (int *)malloc(gsize*stride*sizeof(int));
46
          displs = (int *)malloc(gsize*sizeof(int));
47
          rcounts = (int *)malloc(gsize*sizeof(int));
48
          for (i=0; i<gsize; ++i) {</pre>
```

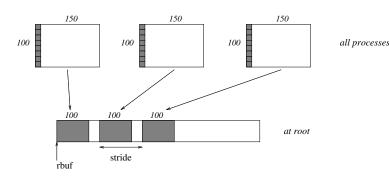


Figure 5.6: The root process gathers column 0 of a  $100 \times 150$  C array, and each set is placed stride ints apart.

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

### Example 5.7

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

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

1

 $\mathbf{2}$ 

3 4 5

6

7

9 10

11

12 13

14

15

16

17

18

19

20

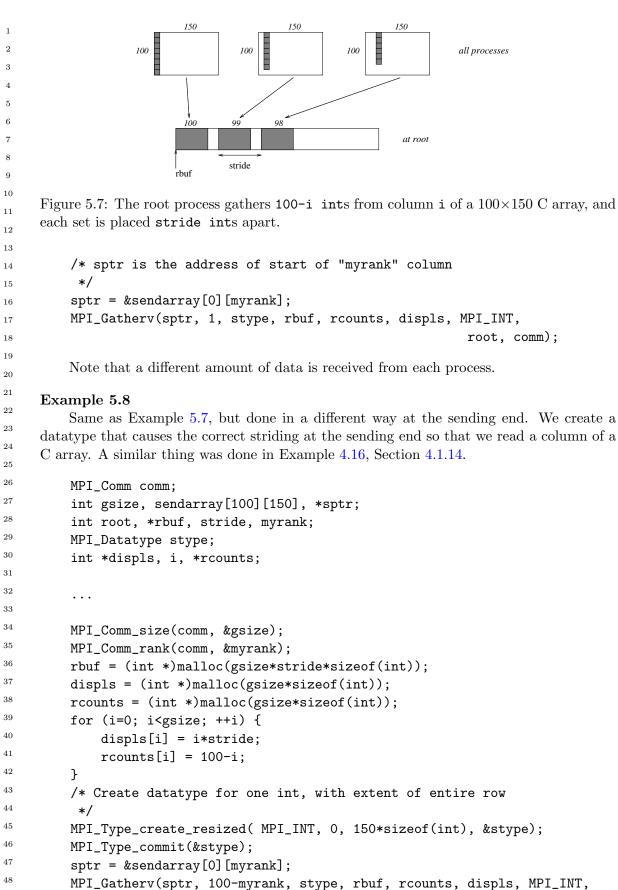
21

22 23 24

25

26

```
CHAPTER 5. COLLECTIVE COMMUNICATION
```



2 3

4

5

6 7

46

47 48

root, comm);

```
Example 5.9
```

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

```
MPI_Comm comm;
                                                                                  8
int gsize,sendarray[100][150],*sptr;
                                                                                  9
int root, *rbuf, *stride, myrank, bufsize;
                                                                                  10
MPI_Datatype stype;
                                                                                  11
int *displs,i,*rcounts,offset;
                                                                                 12
                                                                                 13
. . .
                                                                                 14
                                                                                  15
MPI_Comm_size(comm, &gsize);
                                                                                  16
MPI_Comm_rank(comm, &myrank);
                                                                                  17
                                                                                 18
stride = (int *)malloc(gsize*sizeof(int));
                                                                                 19
. . .
                                                                                 20
/* stride[i] for i = 0 to gsize-1 is set somehow
                                                                                 21
 */
                                                                                 22
                                                                                 23
/* set up displs and rcounts vectors first
                                                                                 ^{24}
 */
                                                                                 25
displs = (int *)malloc(gsize*sizeof(int));
                                                                                 26
rcounts = (int *)malloc(gsize*sizeof(int));
                                                                                 27
offset = 0;
                                                                                 28
for (i=0; i<gsize; ++i) {</pre>
                                                                                 29
    displs[i] = offset;
                                                                                 30
    offset += stride[i];
                                                                                 31
    rcounts[i] = 100-i;
                                                                                 32
}
                                                                                 33
/* the required buffer size for rbuf is now easily obtained
                                                                                 34
 */
                                                                                 35
bufsize = displs[gsize-1]+rcounts[gsize-1];
                                                                                 36
rbuf = (int *)malloc(bufsize*sizeof(int));
                                                                                 37
/* Create datatype for the column we are sending
                                                                                 38
 */
                                                                                 39
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
                                                                                  40
MPI_Type_commit(&stype);
                                                                                 41
sptr = &sendarray[0][myrank];
                                                                                 42
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                                                 43
                                                        root, comm);
                                                                                 44
                                                                                  45
```

Example 5.10

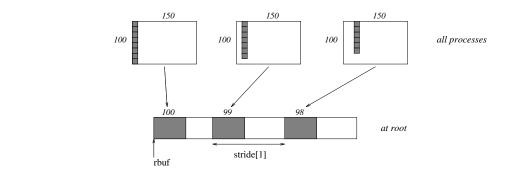


Figure 5.8: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride[i] ints apart (a varying stride).

18

1

2

7

8

9 10

11

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

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

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

3 4 162

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

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

5 6

17

18

19

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

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

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

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

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

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

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

- 34
- 35 36
- 37
- 38
- 39 40
- 41
- 42

- 44 45
- 46
- 47
- 48

0.01 /0 0111			
MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, <sup>1</sup> comm) <sup>2</sup>			
IN	sendbuf	address of send buffer (choice, significant only at root)	3 4
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	4 5 6
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i	7 8 9
IN	sendtype	data type of send buffer elements (handle)	10 11
OUT	recvbuf	address of receive buffer (choice)	12
IN	recvcount	number of elements in receive buffer (non-negative in-teger)	13 14
IN	recvtype	data type of receive buffer elements (handle)	15
IN	root	rank of sending process (integer)	16 17
IN	comm	communicator (handle)	18
			19
int MPI_So	catterv(const void* send)	ouf, const int sendcounts[],	20
	const int displs[], 1	MPI_Datatype sendtype, void* recvbuf,	21
	int recvcount, MPI_D	atatype recvtype, int root, MPI_Comm comm)	22 23
MPI_Scatte	erv(sendbuf, sendcounts,	displs, sendtype, recvbuf, recvcount,	23 24
-	recvtype, root, comm, ierror)		
TYPE(*	<pre> ), DIMENSION(), INTENT</pre>	Γ(IN) :: sendbuf	26
	-	ecvbuf	27
	•	<pre>ounts(*), displs(*), recvcount, root</pre>	28
	<pre>MPI_Datatype), INTENT(IN)</pre>		29
	MPI_Comm), INTENT(IN) ::		30
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: lerror	31
MPI_SCATTE	ERV(SENDBUF, SENDCOUNTS,	DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,	32 33
	RECVTYPE, ROOT, COMM	, IERROR)	33 34
• 1	> SENDBUF(*), RECVBUF(*)		35
		(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,	36
COMM,	IERROR		37
MPI_S	CATTERV is the inverse oper	ation to MPI_GATHERV.	38
		ionality of MPI_SCATTER by allowing a varying	39
	-	s, since sendcounts is now an array. It also allows	40
	more flexibility as to where the data is taken from on the root, by providing an additional <sup>41</sup>		
argument, displs. 42			
If comm is an intracommunicator, the outcome is as if the root executed n send oper- ations, 44			

MPI\_Send(sendbuf+displs[i]· extent(sendtype), sendcounts[i], sendtype, i,...),

and each process executed a receive,

45

 $46 \\ 47$ 

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

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

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

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

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

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

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

26 27

28

30

 $^{31}$ 

32

33

48

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

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

### Example 5.11

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

34MPI\_Comm comm; 35 int gsize,\*sendbuf; 36 int root, rbuf[100]; 37 . . . 38 MPI\_Comm\_size(comm, &gsize); 39 sendbuf = (int \*)malloc(gsize\*100\*sizeof(int)); 40 . . . 41 MPI\_Scatter(sendbuf, 100, MPI\_INT, rbuf, 100, MPI\_INT, root, comm); 4243

### <sup>44</sup> Example 5.12

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

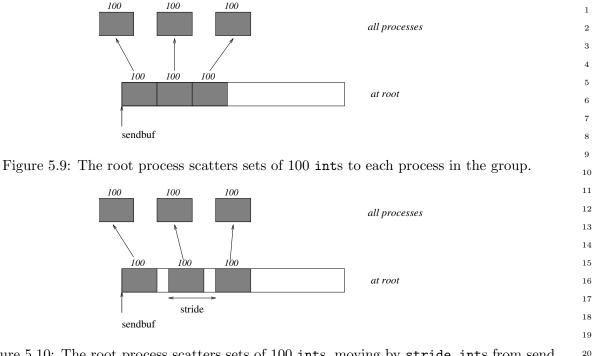


Figure 5.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

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

# ;

21 22 23

 $^{24}$ 

25

26 27

28 29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40 41

42

43

44

 $45 \\ 46$ 

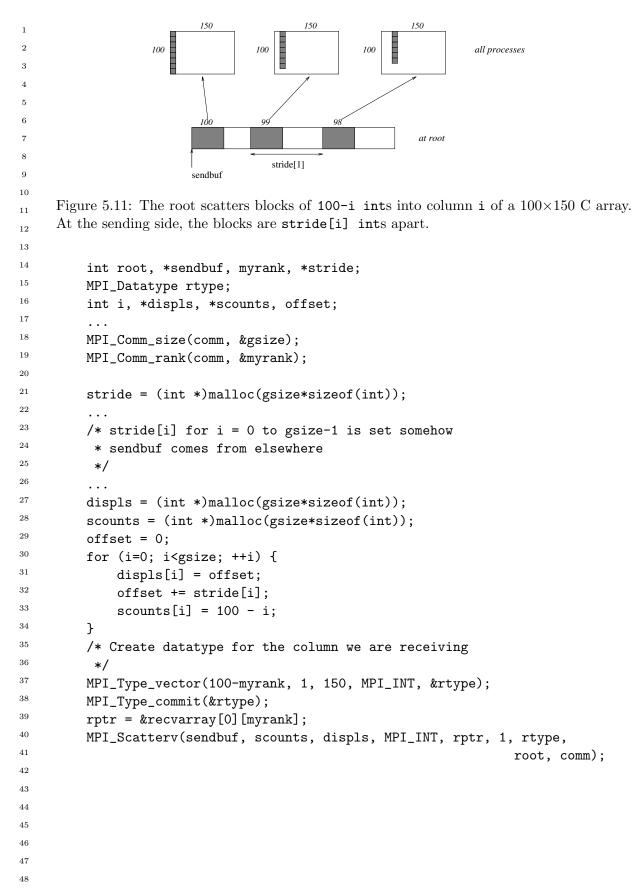
47

48

## Example 5.13

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

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



# 5.7 Gather-to-all

IN	sendbuf	ount, sendtype, recvbuf, recvcount, recvtype, comm) starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
nt MPI	MPI_Datatype s	d* sendbuf, int sendcount, endtype, void* recvbuf, int recvcount, ecvtype, MPI_Comm comm)
TYP TYP INT TYP TYP	<pre>comm, ierror) E(*), DIMENSION(), E(*), DIMENSION() EGER, INTENT(IN) ::</pre>	sendcount, recvcount ENT(IN) :: sendtype, recvtype IN) :: comm
<ty]< td=""><td>COMM, IERROR) pe&gt; SENDBUF(*), RECVI</td><td>COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, BUF(*) TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR</td></ty]<>	COMM, IERROR) pe> SENDBUF(*), RECVI	COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, BUF(*) TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
he resul by every The he type If cc	t, instead of just the roo process and placed in the type signature associated signature associated with	ought of as MPI_GATHER, but where all processes receive t. The block of data sent from the j-th process is received he j-th block of the buffer recvbuf. ed with sendcount, sendtype, at a process must be equal to h recvcount, recvtype at any other process. cator, the outcome of a call to MPI_ALLGATHER() is as
MPI_0	Gather(sendbuf,sendco	ount,sendtype,recvbuf,recvcount, recvtype,root,comm)
	= 0 ,, n-1. The r corresponding rules for	ules for correct usage of MPI_ALLGATHER are easily found MPI_GATHER
		tracommunicators is specified by passing the value

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

# Unofficial Draft for Comment Only

Then the input data of each process is assumed to be in the area where that process would
 receive its own contribution to the receive buffer.

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

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

17 18 19

20 21

8 9

10

11

12

13

14

15

16

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

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

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

5

CHAPTER 5. COLLECTIVE COMMUNICATION

## 5.8 All-to-All Scatter/Gather

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

```
IN
                  sendbuf
6
                                               starting address of send buffer (choice)
\overline{7}
        IN
                  sendcount
                                               number of elements sent to each process (non-negative
8
                                               integer)
9
        IN
                  sendtype
                                               data type of send buffer elements (handle)
10
        OUT
                  recvbuf
                                               address of receive buffer (choice)
11
12
        IN
                                               number of elements received from any process (non-
                  recvcount
13
                                               negative integer)
14
        IN
                  recvtype
                                               data type of receive buffer elements (handle)
15
        IN
                  comm
                                               communicator (handle)
16
17
18
      int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
19
                     void* recvbuf, int recvcount, MPI_Datatype recvtype,
20
                     MPI_Comm comm)
21
     MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
22
                     comm, ierror)
23
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
24
          TYPE(*), DIMENSION(...)
                                      :: recvbuf
25
          INTEGER, INTENT(IN) :: sendcount, recvcount
26
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                    sendtype, recvtype
27
          TYPE(MPI_Comm), INTENT(IN) :: comm
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                    ierror
29
30
     MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
^{31}
                     COMM, IERROR)
32
          <type> SENDBUF(*), RECVBUF(*)
33
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
34
          MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process
35
      sends distinct data to each of the receivers. The j-th block sent from process i is received
36
      by process j and is placed in the i-th block of recvbuf.
37
          The type signature associated with sendcount, sendtype, at a process must be equal to
38
      the type signature associated with recvcount, recvtype at any other process. This implies
39
      that the amount of data sent must be equal to the amount of data received, pairwise between
40
      every pair of processes. As usual, however, the type maps may be different.
41
          If comm is an intracommunicator, the outcome is as if each process executed a send to
42
      each process (itself included) with a call to,
43
44
         MPI_Send(sendbuf+i \cdot sendcount \cdot extent(sendtype), sendcount, sendtype, i, ...),
45
46
     and a receive from every other process with a call to,
47
48
         MPI_Recv(recvbuf+i \cdot recvcount \cdot extent(recvtype), recvcount, recvtype, i,...).
```

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

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

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

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

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

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

Unofficial Draft for Comment Only

 $\mathbf{2}$ 

1 2	MPI_ALLT(	DALLV(sendbuf, sendcounts, server s	displs, sendtype, recvbuf, recvcounts, rdispls,
3 4	IN	sendbuf	starting address of send buffer (choice)
5 6	IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank
7 8 9	IN	sdispls	integer array (of length group size). Entry $j$ specifies the displacement (relative to ${\sf sendbuf})$ from which to take the outgoing data destined for process $j$
10 11	IN	sendtype	data type of send buffer elements (handle)
12	OUT	recvbuf	address of receive buffer (choice)
13 14 15	IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each rank
16 17 18 19	IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
20	IN	recvtype	data type of receive buffer elements (handle)
21	IN	comm	communicator (handle)
24 25 26 27 28 29 30 31 32 33 34 35 36	TYPE(* TYPE(* INTEGH rdisp] TYPE(N TYPE(N	<pre>const int recvcounts MPI_Datatype recvtyp allv(sendbuf, sendcounts rdispls, recvtype, c *), DIMENSION(), INTENT *), DIMENSION() :: re ER, INTENT(IN) :: sendco</pre>	<pre>, sdispls, sendtype, recvbuf, recvcounts, omm, ierror) T(IN) :: sendbuf ecvbuf ounts(*), sdispls(*), recvcounts(*), ) :: sendtype, recvtype comm</pre>
37 38 39 40 41 42 43 44 45 46 47 48	<type> INTEGH RECVTY MPI_A the send is side is speci If comr</type>	RDISPLS, RECVTYPE, C > SENDBUF(*), RECVBUF(*) ER SENDCOUNTS(*), SDISPLS (PE, COMM, IERROR LLTOALLV adds flexibility to specified by sdispls and the lo ified by rdispls. m is an intracommunicator, the	, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, OMM, IERROR) S(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), o MPI_ALLTOALL in that the location of data for ocation of the placement of the data on the receive men the j-th block sent from process i is received by of recvbuf. These blocks need not all have the same

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

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

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

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

MPI\_Recv(recvbuf+rdispls[i] extent(recvtype), recvcounts[i], recvtype, i,...).

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

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

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

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

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

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

 $^{24}$ 

 $^{31}$ 

1 2	MPI_ALLT	OALLW(sendbuf, sendcounts, s recvtypes, comm)	sdispls, sendtypes, recvbuf, recvcounts, rdispls,
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6	IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank
7 8 9 10	IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
11 12 13 14	IN	sendtypes	array of datatypes (of length group size). Entry j spec- ifies the type of data to send to process j (array of handles)
14	OUT	recvbuf	address of receive buffer (choice)
16 17 18	IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each rank
19 20 21 22	IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
23 24 25 26	IN	recvtypes	array of datatypes (of length group size). Entry i spec- ifies the type of data received from process i (array of handles)
27	IN	comm	communicator (handle)
28 29 30 31 32 33	int MPI_A	<pre>const int sdispls[], void* recvbuf, const</pre>	<pre>dbuf, const int sendcounts[], const MPI_Datatype sendtypes[], int recvcounts[], const int rdispls[], ecvtypes[], MPI_Comm comm)</pre>
34 35 36 37 38 39 40 41 42	TYPE( TYPE( INTEG rdisp TYPE( TYPE(	<pre>rdispls, recvtypes, *), DIMENSION(), INTEN *), DIMENSION() :: re</pre>	<pre>F(IN) :: sendbuf ecvbuf punts(*), sdispls(*), recvcounts(*), ) :: sendtypes(*) ) :: recvtypes(*)</pre>
43 44			, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
45 46 47 48	INTEG	RDISPLS, RECVTYPES, > SENDBUF(*), RECVBUF(*) ER SENDCOUNTS(*), SDISPLS PLS(*), RECVTYPES(*), COM	S(*), SENDTYPES(*), RECVCOUNTS(*),

#### 5.9. GLOBAL REDUCTION OPERATIONS

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

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

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

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

MPI\_Send(sendbuf+sdispls[i],sendcounts[i],sendtypes[i],i,...),

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

MPI\_Recv(recvbuf+rdispls[i],recvcounts[i],recvtypes[i],i,...).

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

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

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

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

## 5.9 Global Reduction Operations

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

 $\mathbf{2}$ 

 $^{24}$ 

 $^{31}$ 

```
176
                                           CHAPTER 5. COLLECTIVE COMMUNICATION
1
      5.9.1
            Reduce
\mathbf{2}
3
4
      MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
5
       IN
                  sendbuf
                                              address of send buffer (choice)
6
       OUT
7
                  recvbuf
                                              address of receive buffer (choice, significant only at
8
                                              root)
9
       IN
                  count
                                              number of elements in send buffer (non-negative inte-
10
                                              ger)
11
                                              data type of elements of send buffer (handle)
       IN
                  datatype
12
       IN
                                              reduce operation (handle)
13
                  ор
14
       IN
                                              rank of root process (integer)
                  root
15
       IN
                  comm
                                              communicator (handle)
16
17
      int MPI_Reduce(const void* sendbuf, void* recvbuf, int count,
18
                     MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
19
20
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
21
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
22
          TYPE(*), DIMENSION(..) :: recvbuf
23
          INTEGER, INTENT(IN) :: count, root
^{24}
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Op), INTENT(IN) :: op
26
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
29
          <type> SENDBUF(*), RECVBUF(*)
30
          INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
^{31}
32
          If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the
33
     input buffer of each process in the group, using the operation op, and returns the combined
34
      value in the output buffer of the process with rank root. The input buffer is defined by
35
      the arguments sendbuf, count and datatype; the output buffer is defined by the arguments
36
      recvbuf, count and datatype; both have the same number of elements, with the same type.
37
      The routine is called by all group members using the same arguments for count, datatype, op,
38
      root and comm. Thus, all processes provide input buffers of the same length, with elements
39
      of the same type as the output buffer at the root. Each process can provide one element, or a
40
      sequence of elements, in which case the combine operation is executed element-wise on each
41
      entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains
42
      two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then
43
      recvbuf(1) = global \max(sendbuf(1)) and recvbuf(2) = global \max(sendbuf(2)).
44
          Section 5.9.2, lists the set of predefined operations provided by MPI. That section also
45
      enumerates the datatypes to which each operation can be applied.
46
          In addition, users may define their own operations that can be overloaded to operate
47
      on several datatypes, either basic or derived. This is further explained in Section 5.9.5.
48
```

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

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

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

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

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

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

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

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

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI\_ROOT in root. All other processes in group A pass the value MPI\_PROC\_NULL in root. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root. 43 44 44 45 46 46 47 88

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10

11

12

13 14

15

16

17

18

19

20

21

22

23 24

25

26

27

28

29

30

31

32

33 34

35

36

37

38

39

40 41

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

		MPI_TYPE_CREATE_F90_REAL,	1
		and if available: MPI_REAL2,	2
		MPI_REAL4, MPI_REAL8, MPI_REAL16	3
	Logical:	MPI_LOGICAL,MPI_C_BOOL,	4
	-	MPI_CXX_BOOL	5
	Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	6
		MPI_C_FLOAT_COMPLEX (as synonym),	7
		MPI_C_DOUBLE_COMPLEX,	8
		MPI_C_LONG_DOUBLE_COMPLEX,	9
		MPI_CXX_FLOAT_COMPLEX,	10
		MPI_CXX_DOUBLE_COMPLEX,	11
		MPI_CXX_LONG_DOUBLE_COMPLEX,	12
		and handles returned from	13
		MPI_TYPE_CREATE_F90_COMPLEX,	14
		and if available: MPI_DOUBLE_COMPLEX,	15
		MPI_COMPLEX4, MPI_COMPLEX8,	16
		MPI_COMPLEX16, MPI_COMPLEX32	10
	Byte:	MPI_BYTE	18
	Multi-language types:	MPI_AINT, MPI_OFFSET, MPI_COUNT	
	Now, the valid datatypes for each oper	ration are specified below	19
	Now, the valid datatypes for each open	ration are specified below.	20
			21
	Ор	Allowed Types	22
	θÞ	Anowed Types	23
	MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point,	24
		Multi-language types	25
	MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex,	26
		Multi-language types	27
	MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical	28
	MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte, Multi-language types	29
			30
		ed datatypes are valid in all supported program-	31
n	ning languages, see also Reduce Operation	as in Section $18.2.6$ .	32
	The following examples use intracomm	nunicators.	33
			34
E	Example 5.15		35
		luct of two vectors that are distributed across a	36
g	roup of processes and returns the answer	at node zero.	37
			38
			39
			40
			41
			42
			43
			44
			45
			46
			47
			48

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

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

<sup>48</sup> MPI\_WCHAR, and MPI\_CHARACTER will be translated so as to preserve the printable

character, whereas MPI\_SIGNED\_CHAR and MPI\_UNSIGNED\_CHAR will be translated so as to preserve the integer value.

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

### 5.9.4 MINLOC and MAXLOC

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

The operation that defines MPI\_MAXLOC is:

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

where

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

and

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

MPI\_MINLOC is defined similarly:

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

where

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

and

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

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

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

1	according to the first component of	each pair, and ties are resolved according to the second	
2	component.		
3	The reduce operation is defined to operate on arguments that consist of a pair: value		
4	and index. For both Fortran and C, types are provided to describe the pair. The potentially		
5	mixed-type nature of such argumen	ts is a problem in Fortran. The problem is circumvented,	
6	for Fortran, by having the MPI-pro	ovided type consist of a pair of the same type as value,	
7	,	e also. In C, the MPI-provided pair type has distinct	
8	types and the index is an int.		
9	<i>U</i> <b>1</b>	d MPI_MAXLOC in a reduce operation, one must provide	
10		nts a pair (value and index). MPI provides nine such	
11		ions MPI_MAXLOC and MPI_MINLOC can be used with	
12	each of the following datatypes.		
13	call of the following datatypes.		
14	Fortran:		
15	Name	Description	
16	MPI_2REAL	pair of REALs	
17	MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables	
18	MPI_2INTEGER	pair of INTEGERS	
19		Part of	
20	<u> </u>		
21	C:		
22		Description	
23	MPI_FLOAT_INT	float and int	
24	MPI_DOUBLE_INT	double and int	
25	MPI_LONG_INT	long and int	
26	MPI_2INT	pair of int	
27	MPI_SHORT_INT	short and int	
28	MPI_LONG_DOUBLE_INT	long double and int	
29	The datatype $MPI_2REAL$ is as	if defined by the following (see Section $4.1$ ).	
30			
31	MPI_TYPE_CONTIGUOUS(2, MPI_RE	AL, MPI_2REAL)	
32			
33	Similar statements apply for M	$PI\_2INTEGER, MPI\_2DOUBLE\_PRECISION, \mathrm{and} \; MPI\_2INT.$	
34	The datatype $MPI_FLOAT_INT$	is as if defined by the following sequence of instructions.	
35			
36	type[0] = MPI_FLOAT		
37	type[1] = MPI_INT		
38	disp[0] = 0		
39	disp[1] = sizeof(float)		
40	block[0] = 1		
41	block[1] = 1		
42	MPI_TYPE_CREATE_STRUCT(2, blo	ck, disp, type, MPI_FLOAT_INT)	
43	Similar statements apply for MPI_L	ONIC INT and MPL DOUBLE INT	
44	The following examples use int		
45	The following examples use in		
46	Example 5.17		
47	-	30 doubles, in C. For each of the 30 locations, compute	
	Laon process has an array of e	o as as the second seco	

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

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

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

*Rationale.* The definition of MPI\_MINLOC and MPI\_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of MPI\_MAXLOC and MPI\_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

#### 5.9.5 User-Defined Reduction Operations

LOGICAL COMMUTE

INTEGER OP, IERROR

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

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

The argument user\_fn is the user-defined function, which must have the following four arguments: invec, inoutvec, len, and datatype. The ISO C prototype for the function is the following. two defined MDL large function (argidit investore and data the second sec

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

The Fortran declarations of the user-defined function user\_fn appear below. ABSTRACT INTERFACE SUBROUTINE MPI\_User\_function(invec, inoutvec, len, datatype)

USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR

1

 $\mathbf{2}$ 

3

4

5

7 8

9 10 11

12

13 14

15

16 17

18 19

20

21

22

23

24

25

26

27

28 29

30

31

32

33

34

35

36

37

38

43

44

45

46

47

1	TYDE (C DTD) VALUE invog incutvog
2	TYPE(C_PTR), VALUE :: invec, inoutvec INTEGER :: len
3	
4	TYPE(MPI_Datatype) :: datatype
5	SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
6	<type> INVEC(LEN), INOUTVEC(LEN)</type>
7	INTEGER LEN, DATATYPE
8	The detet we comment is a bardle to the data time that may mean distantly all to
9	The datatype argument is a handle to the data type that was passed into the call to
10	MPI_REDUCE. The user reduce function should be written such that the following holds: Let $u[0], \ldots, u[len-1]$ be the len elements in the communication buffer described by the
11	arguments invec, len and datatype when the function is invoked; let $v[0],\ldots$ , $v[\text{len-1}]$ be len
12	elements in the communication buffer described by the arguments inoutvec, len and datatype
13	when the function is invoked; let $w[0], \ldots, w[len-1]$ be len elements in the communication
14	buffer described by the arguments inoutvec, len and datatype when the function returns;
15	then $w[i] = u[i] \circ v[i]$ , for i=0,, len-1, where $\circ$ is the reduce operation that the function
16	computes.
17	Informally, we can think of invec and inoutvec as arrays of len elements that user_fn
18	is combining. The result of the reduction overwrites values in inoutvec, hence the name.
19	Each invocation of the function results in the pointwise evaluation of the reduce operator
20	on len elements: i.e., the function returns in $inoutvec[i]$ the value $invec[i] \circ inoutvec[i]$ , for
21	i=0,, count-1, where $\circ$ is the combining operation computed by the function.
22	
23	<i>Rationale.</i> The len argument allows MPI_REDUCE to avoid calling the function for
24	each element in the input buffer. Rather, the system can choose to apply the function to church of input. In $C$ , it is passed in as a reference for passeng of compatibility
25	to chunks of input. In C, it is passed in as a reference for reasons of compatibility
26	with Fortran.
27	By internally comparing the value of the datatype argument to known, global handles,
28	it is possible to overload the use of a single user-defined function for several, different
29 30	data types. (End of rationale.)
31	Consuldatations need to the user function. However, use of datations that
32	General datatypes may be passed to the user function. However, use of datatypes that
33	are not contiguous is likely to lead to inefficiencies. No MPI communication function may be called inside the user function. MPI_ABORT
34	may be called inside the function in case of an error.
35	may be called inside the function in case of an error.
36	Advice to users. Suppose one defines a library of user-defined reduce functions that
37	are overloaded: the datatype argument is used to select the right execution path at each
38	invocation, according to the types of the operands. The user-defined reduce function
39	cannot "decode" the datatype argument that it is passed, and cannot identify, by itself,
40	the correspondence between the datatype handles and the datatype they represent.
41	This correspondence was established when the datatypes were created. Before the
42	library is used, a library initialization preamble must be executed. This preamble
43	code will define the datatypes that are used by the library, and store handles to these
44	datatypes in global, static variables that are shared by the user code and the library
45	code.
46	The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using
47	the Fortran calling conventions and will pass a Fortran-type datatype argument; the
48	
-1	C version will use C calling convention and the C representation of a datatype handle.

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

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

```
MPI_Comm_size(comm, &groupsize);
MPI_Comm_rank(comm, &rank);
if (rank > 0) {
    MPI_Recv(tempbuf, count, datatype, rank-1,...);
    User_reduce(tempbuf, sendbuf, count, datatype);
}
if (rank < groupsize-1) {
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
}
/* answer now resides in process groupsize-1 ... now send to root
 */
if (rank == root) {
    MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
}
if (rank == groupsize-1) {
    MPI_Send(sendbuf, count, datatype, root, ...);
}
if (rank == root) {
    MPI_Wait(&req, &status);
}
```

The reduction computation proceeds, sequentially, from process 0 to process groupsize-1. This order is chosen so as to respect the order of a possibly noncommutative operator defined by the function User\_reduce(). A more efficient implementation is achieved by taking advantage of associativity and using a logarithmic tree reduction. Commutativity can be used to advantage, for those cases in which the commute argument to MPI\_OP\_CREATE is true. Also, the amount of temporary buffer required can be reduced, and communication can be pipelined with computation, by transferring and reducing the elements in chunks of size len <count.

The predefined reduce operations can be implemented as a library of user-defined operations. However, better performance might be achieved if MPI\_REDUCE handles these functions as a special case. (*End of advice to implementors.*)

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

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

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

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

## 5.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

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

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

MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm) MPI\_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)

TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf

## Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $46 \\ 47$ 

1	TYPE(*), DIMENSION() :: recvbuf
2	INTEGER, INTENT(IN) :: count
3	TYPE(MPI_Datatype), INTENT(IN) :: datatype
4	TYPE(MPI_Op), INTENT(IN) :: op
5	TYPE(MPI_Comm), INTENT(IN) :: comm
6	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7	MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
8	<type> SENDBUF(*), RECVBUF(*)</type>
9	INTEGER COUNT, DATATYPE, OP, COMM, IERROR
10 11	
12	If comm is an intracommunicator, MPI_ALLREDUCE behaves the same as
12	MPI_REDUCE except that the result appears in the receive buffer of all the group members.
14	Advice to implementors. The all-reduce operations can be implemented as a re-
15	duce, followed by a broadcast. However, a direct implementation can lead to better
16	performance. (End of advice to implementors.)
17	For a first ( - the of an of a first of a fi
18	The "in place" option for intracommunicators is specified by passing the value
19	MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is
20	taken at each process from the receive buffer, where it will be replaced by the output data.
21	If comm is an intercommunicator, then the result of the reduction of the data provided
22	by processes in group A is stored at each process in group B, and vice versa. Both groups
23	should provide <b>count</b> and <b>datatype</b> arguments that specify the same type signature.
24	The following example uses an intracommunicator.
25	Example 5.22
25 26	<b>Example 5.22</b> A routine that computes the product of a vector and an array that are distributed
	A routine that computes the product of a vector and an array that are distributed
26	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16).
26 27	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
26 27 28 29 30	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array
26 27 28 29 30 31	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result
26 27 28 29 30 31 32	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n)
26 27 28 29 30 31 32 33	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result
26 27 28 29 30 31 32 33 34	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr
26 27 28 29 30 31 32 33 34 35	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum
26 27 28 29 30 31 32 33 34 35 36	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n
26 27 28 29 30 31 32 33 34 35 36 37	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n sum(j) = 0.0
26 27 28 29 30 31 32 33 34 35 36 37 38	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n sum(j) = 0.0 DO i = 1, m
26 27 28 29 30 31 32 33 34 35 36 37 38 39	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n sum(j) = 0.0 DO i = 1, m sum(j) = sum(j) + a(i)*b(i,j)
26 27 28 29 30 31 32 33 34 35 36 37 38	<pre>A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16).</pre> SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum D0 j = 1, n sum(j) = 0.0 D0 i = 1, m sum(j) = sum(j) + a(i)*b(i,j) END D0
26 27 28 30 31 32 33 34 35 36 37 38 39 40	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n sum(j) = 0.0 DO i = 1, m sum(j) = sum(j) + a(i)*b(i,j)
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j = 1, n sum(j) = 0.0 DO i = 1, m sum(j) = sum(j) + a(i)*b(i,j) END DO END DO
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	<pre>A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j = 1, n sum(j) = 0.0 DO i = 1, m sum(j) = sum(j) + a(i)*b(i,j) END DO END DO ! global sum</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j = 1, n sum(j) = 0.0 DO i = 1, m sum(j) = sum(j) + a(i)*b(i,j) END DO END DO
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	<pre>A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n sum(j) = 0.0 DO i = 1, m sum(j) = sum(j) + a(i)*b(i,j) END DO END DO ! global sum CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n sum(j) = 0.0 DO i = 1, m sum(j) = sum(j) + a(i)*b(i,j) END DO END DO ! global sum CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr) ! return result at all nodes</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	<pre>A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16). SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm) REAL a(m), b(m,n) ! local slice of array REAL c(n) ! result REAL sum(n) INTEGER n, comm, i, j, ierr ! local sum DO j= 1, n sum(j) = 0.0 DO i = 1, m sum(j) = sum(j) + a(i)*b(i,j) END DO END DO ! global sum CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)</pre>

0.01 0.20				
5.9.7 Pro	ocess-Local Reduction		1	
implement	The functions in this section are of importance to library implementors who may want to implement special reduction patterns that are otherwise not easily covered by the standard MPI operations.			
The fo	ollowing function applies a rec	luction operator to local arguments.	5 6	
			7 8	
	JCE_LOCAL( inbuf, inoutbuf, o		9	
IN	inbuf	input buffer (choice)	10	
INOUT	inoutbuf	combined input and output buffer (choice)	11	
IN	count	number of elements in inbuf and inoutbuf buffers (non-negative integer) $% \left( {{\left[ {{{\left[ {{\left[ {{\left[ {{\left[ {{\left[ {{\left[ $	12 13	
IN	datatype	data type of elements of inbuf and inoutbuf buffers (handle)	14 15 16	
IN	ор	operation (handle)	17	
· ·			18 19	
int MPI_R	<pre>Leduce_local(const void* : MPI_Datatype datatyp</pre>	inbuf, void* inoutbuf, int count,	20	
			21	
		count, datatype, op, ierror)	22	
	<pre>(*), DIMENSION(), INTEN (*), DIMENSION() :: in </pre>	Γ(IN) :: inbuf noutbuf	23	
	ER, INTENT(IN) :: count	loutbul	24	
	[MPI_Datatype), INTENT(IN]	) :: datatype	25 26	
	<pre>MPI_Op), INTENT(IN) :: 0</pre>		20	
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	28	
MPT REDUC	E LOCAL(INBUE, INOUTBUE,	COUNT, DATATYPE, OP, IERROR)	29	
	<pre>&gt; INBUF(*), INOUTBUF(*)</pre>		30	
• -	ER COUNT, DATATYPE, OP,	IERROR	31	
The f	unction applies the operation	given by op element-wise to the elements of inbuf	32	
		nent-wise in inoutbuf, as explained for user-defined	33 34	
		f and inoutbuf (input as well as result) have the	35	
same num	ber of elements given by cour	and the same datatype given by datatype. The	36	
	ACE option is not allowed.		37	
Reduc	ction operations can be querie	d for their commutativity.	38	
			39	
MPI_OP_O	COMMUTATIVE( op, commute	2)	40	
IN	ор	operation (handle)	41 42	
	-	,	42	
OUT	commute	true if op is commutative, false otherwise (logical)	44	
int MDT (	p_commutative(MPI_Op op,	int *commute)	45	
			46	
-	mmutative(op, commute, i		47	
TYPE(	TYPE(MPI_Op), INTENT(IN) :: op48			

1 2		AL, INTENT(OUT) :: commu ER, OPTIONAL, INTENT(OUT)	
3 4 5 6 7	LOGIC	MMUTATIVE(OP, COMMUTE, I AL COMMUTE ER OP, IERROR	ERROR)
8 9	5.10 Re	educe-Scatter	
10 11 12 13	in a group	-	ations where the result is scattered to all processes rs equal-sized blocks to all processes, while another size for each process.
14 15 16	5.10.1 M	IPI_REDUCE_SCATTER_BL	ОСК
17 18	MPI REDI	ICE SCATTER BLOCK( send	buf, recvbuf, recvcount, datatype, op, comm)
19	IN	sendbuf	starting address of send buffer (choice)
20	OUT	recvbuf	starting address of receive buffer (choice)
21 22	IN	recvcount	element count per block (non-negative integer)
23 24	IN	datatype	data type of elements of send and receive buffers (han- dle)
25 26	IN	ор	operation (handle)
27	IN	comm	communicator (handle)
28 29 30 31 32	int MPI_R		t void* sendbuf, void* recvbuf, atatype datatype, MPI_Op op,
33	MPI_Reduc		recvbuf, recvcount, datatype, op, comm,
34	TYPE(	ierror) (*), DIMENSION(), INTEN	Γ(IN) :: sendbuf
35 36		(*), DIMENSION() :: re	
37		ER, INTENT(IN) :: recvco	
38		<pre>MPI_Datatype), INTENT(IN) MPI_Op), INTENT(IN) :: </pre>	v -
39 40		MPI_Comm), INTENT(IN) ::	-
41	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror
42	MPI_REDUC	E_SCATTER_BLOCK(SENDBUF,	RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
43 44	<i>.</i>	IERROR)	
45	• 1	> SENDBUF(*), RECVBUF(*) ER RECVCOUNT, DATATYPE, (	IP. COMM. TERROR
46			MPI_REDUCE_SCATTER_BLOCK first performs a
47 48		,	$r of count = n^{*}recvcount$ elements in the send buffers
10	- /		

defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcount, datatype, op and comm. The resulting vector is treated as n consecutive blocks of recvcount elements that are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcount, and datatype.

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

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

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

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

# 5.10.2 MPI\_REDUCE\_SCATTER

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

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

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

 $\mathbf{2}$ 

 $^{24}$ 

```
1
     int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf,
2
                     const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
3
                     MPI_Comm comm)
4
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
5
                     ierror)
6
          TYPE(*), DIMENSION(...), INTENT(IN) ::
                                                        sendbuf
7
          TYPE(*), DIMENSION(..) :: recvbuf
8
          INTEGER, INTENT(IN) :: recvcounts(*)
9
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
          TYPE(MPI_Op), INTENT(IN) :: op
11
          TYPE(MPI_Comm), INTENT(IN) :: comm
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
15
                     IERROR)
16
          <type> SENDBUF(*), RECVBUF(*)
17
          INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
18
          If comm is an intracommunicator, MPI_REDUCE_SCATTER first performs a global,
19
     element-wise reduction on vectors of count = \sum_{i=0}^{n-1} recvcounts[i] elements in the send buffers
20
     defined by sendbuf, count and datatype, using the operation op, where n is the number of
21
     processes in the group of comm. The routine is called by all group members using the
22
     same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as
23
     n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The
24
     blocks are scattered to the processes of the group. The i-th block is sent to process i and
25
     stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.
26
27
           Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equiv-
28
           alent to: an MPI_REDUCE collective operation with count equal to the sum of
29
           recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. How-
30
           ever, a direct implementation may run faster. (End of advice to implementors.)
31
32
          The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in
33
     the sendbuf argument. In this case, the input data is taken from the receive buffer. It is
34
     not required to specify the "in place" option on all processes, since the processes for which
35
     recvcounts[i] == 0 may not have allocated a receive buffer.
36
          If comm is an intercommunicator, then the result of the reduction of the data provided
37
     by processes in one group (group A) is scattered among processes in the other group (group
38
     B), and vice versa. Within each group, all processes provide the same recvcounts argument,
39
     and provide input vectors of count = \sum_{i=0}^{n-1} recvcounts[i] elements stored in the send buffers,
40
     where n is the size of the group. The resulting vector from the other group is scattered in
41
     blocks of recvcounts[i] elements among the processes in the group. The number of elements
42
     count must be the same for the two groups.
43
44
           Rationale. The last restriction is needed so that the length of the send buffer can be
45
           determined by the sum of the local recvcounts entries. Otherwise, a communication
46
           is needed to figure out how many elements are reduced. (End of rationale.)
```

47 48

5.11. S	CAN	:	195
5.11	Scan		1
5.11.1	Inclusive Scan		2 3
			4
			5
MPI_SC	AN(sendbuf, recvbuf, count, dat	atype, op, comm)	6
IN	sendbuf	starting address of send buffer (choice)	7
OUT	recvbuf	starting address of receive buffer (choice)	9
IN	count	number of elements in input buffer (non-negative teger)	in- <sup>10</sup> 11
IN	datatype	data type of elements of input buffer (handle)	12
IN	ор	operation (handle)	13 14
IN	•		14 15
IIN	comm	communicator (handle)	16
int MP1	[ Scan(const void* sendbuf.	void* recvbuf, int count,	17
	-	pe, MPI_Op op, MPI_Comm comm)	18
MPT Sca	an (sendbuf recubuf count	datatype, op, comm, ierror)	19 20
	PE(*), DIMENSION(), INTEN		20 21
	PE(*), DIMENSION() :: r		22
	TEGER, INTENT(IN) :: count		23
	PE(MPI_Datatype), INTENT(IN		24
	PE(MPI_Op), INTENT(IN) :: PE(MPI_Comm), INTENT(IN) ::	-	25
	TEGER, OPTIONAL, INTENT(OUT		26 27
			28
	AN(SENDBUF, RECVBUF, CUUNT, /pe> SENDBUF(*), RECVBUF(*)	DATATYPE, OP, COMM, IERROR)	29
•	TEGER COUNT, DATATYPE, OP,		30
			31
	,	MPI_SCAN is used to perform a prefix reduction	
		operation returns, in the receive buffer of the proc s in the send buffers of processes with ranks 0,.	
		group members using the same arguments for cou	
		or user-defined operations, the same rules apply	
		ons supported, their semantics, and the constrai	
	and receive buffers are as for ${\sf N}$		38
		municators is specified by passing MPI_IN_PLACE	
	-	he input data is taken from the receive buffer, a	
-	l by the output data. is operation is invalid for interce	ommunicators	41
111	is operation is invalid for interce	Jiiiiumcators.	42 43
			44

```
196
                                          CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.11.2 Exclusive Scan
\mathbf{2}
3
4
     MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
\overline{7}
                 recvbuf
                                              starting address of receive buffer (choice)
8
       IN
                 count
                                              number of elements in input buffer (non-negative in-
9
                                              teger)
10
       IN
                                              data type of elements of input buffer (handle)
                 datatype
11
       IN
                                              operation (handle)
12
                 op
13
       IN
                                             intracommunicator (handle)
                 comm
14
15
     int MPI_Exscan(const void* sendbuf, void* recvbuf, int count,
16
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
17
18
     MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
19
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
          TYPE(*), DIMENSION(..) :: recvbuf
20
21
          INTEGER, INTENT(IN) ::
                                     count
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
23
          TYPE(MPI_Op), INTENT(IN) :: op
^{24}
          TYPE(MPI_Comm), INTENT(IN) :: comm
25
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
27
          <type> SENDBUF(*), RECVBUF(*)
28
          INTEGER COUNT, DATATYPE, OP, COMM, IERROR
29
30
         If comm is an intracommunicator, MPI_EXSCAN is used to perform a prefix reduction
^{31}
     on data distributed across the group. The value in recvbuf on the process with rank 0 is
32
     undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process
```

with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks  $0, \ldots, i-1$  (inclusive). The routine is called by all group members using the same arguments for count, datatype, op and comm, except that for user-defined operations, the same rules apply as for MPI\_REDUCE. The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI\_REDUCE.

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

44

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

### 5.11.3 Example using MPI\_SCAN

The example in this section uses an intracommunicator.

#### Example 5.23

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

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

The operator that produces this effect is

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

where

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

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

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

1 2

3 4

5

6

7

18 19

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

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

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

# 5.12 Nonblocking Collective Operations

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

37 The nonblocking collective communication model is similar to the model used for non-38blocking point-to-point communication. A nonblocking call initiates a collective operation, 39 which must be completed in a separate completion call. Once initiated, the operation 40may progress independently of any computation or other communication at participating 41 processes. In this manner, nonblocking collective operations can mitigate possible synchro-42nizing effects of collective operations by running them in the "background." In addition to 43enabling communication-computation overlap, nonblocking collective operations can per-44form collective operations on overlapping communicators, which would lead to deadlocks 45with blocking operations. Their semantic advantages can also be useful in combination with 46 point-to-point communication.

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

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

26 27

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

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

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

Upon returning from a completion call in which a nonblocking collective operation completes, the MPI\_ERROR field in the associated status object is set appropriately, see Section 3.2.5. The values of the MPI\_SOURCE and MPI\_TAG fields are undefined. It is valid to mix different request types (i.e., any combination of collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI\_WAITALL). It is erroneous to call MPI\_REQUEST\_FREE or MPI\_CANCEL for a request associated with a nonblocking collective operation. Nonblocking collective requests are not persistent.

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

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

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

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

#### **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

 $46 \\ 47$ 

	200	CH	HAPTER 5.	COLLECTIVE COMMUNICATION
1 2 3	t	<u> </u>	operations 1	ptimized for minimal time to comple- nay balance time to completion with
4 5 6		The use of tags for collective opera End of rationale.)	ations can pr	event certain hardware optimizations.
7 8 9 10 11	c i	collective operations, then a nonb	olocking colle	e matching blocking and nonblocking ective operation can be initiated and o emulate blocking behavior. ( <i>End of</i>
12 13 14 15 16	as its l tion. L effect a	blocking counterpart for intracom ikewise, upon completion, nonbloc	municators a cking collecti	ollective operation has the same effect and intercommunicators after comple- ve reduction operations have the same restrictions and recommendations on
17 18 19 20	T blockir as botl	he use of the "in place" option is an a collective operations. When usi	ng the "in p	tly as described for the corresponding lace" option, message buffers function l not be modified or accessed until the
21 22 23	Pi	-	-	erations are similar to progression of on $3.7.4$ .
24 25 26 27	l		g nonblocking	e operations can be implemented with g point-to-point communication and a <i>ntors.</i> )
28 29	5.12.1	Nonblocking Barrier Synchroniz	zation	
30 31 32	MPI_I	3ARRIER(comm , request)		
33	IN	comm	communicat	or (handle)
34 35	OUT	request	communicat	ion request (handle)
36 37	int MH	PI_Ibarrier(MPI_Comm comm, MF	PI_Request	*request)
38 39 40 41	TY TY	Darrier(comm, request, ierron YPE(MPI_Comm), INTENT(IN) :: YPE(MPI_Request), INTENT(OUT) WTEGER, OPTIONAL, INTENT(OUT)	comm :: reque	
42 43 44		BARRIER(COMM, REQUEST, IERROF NTEGER COMM, REQUEST, IERROR	1)	
45 46 47	a proc	ess notifies that it has reached th	e barrier. T	BARRIER. By calling MPI_IBARRIER, he call returns immediately, indepen- ARRIER. The usual barrier semantics

 $^{48}$ 

are enforced at the corresponding completion operation (test or wait), which in the intracommunicator case will complete only after all other processes in the communicator have called MPI\_IBARRIER. In the intercommunicator case, it will complete when all processes in the remote group have called MPI\_IBARRIER.

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

#### 5.12.2 Nonblocking Broadcast

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

			16
INOUT	buffer	starting address of buffer (choice)	17
IN	count	number of entries in buffer (non-negative integer)	18
IN	datatype	data type of buffer (handle)	19
IN	root	rank of broadcast root (integer)	20
			21
IN	comm	communicator (handle)	22
OUT	request	communication request (handle)	23 24
			24 25
int MPI_	Ibcast(void* buffer,	<pre>int count, MPI_Datatype datatype, int root,</pre>	26
	MPI_Comm comm, 1	MPI_Request *request)	27
MPI Ibcas	st(buffer, count, dat	tatype, root, comm, request, ierror)	28
		ASYNCHRONOUS :: buffer	29
	GER, INTENT(IN) :: c		30
TYPE	(MPI_Datatype), INTEN	NT(IN) :: datatype	31
TYPE	(MPI_Comm), INTENT(IN	N) :: comm	32
TYPE	(MPI_Request), INTENT	T(OUT) :: request	33
INTE	GER, OPTIONAL, INTENI	C(OUT) :: ierror	34
MPT TRCAS	ST (BUFFER COUNT DAT	TATYPE, ROOT, COMM, REQUEST, IERROR)	35
	e> BUFFER(*)		36
01		ROOT, COMM, REQUEST, IERROR	37
			38 39
This	call starts a nonblocking	variant of MPI_BCAST (see Section 5.4).	39 40
			40
Example u	sing MPI_IBCAST		42
The exam	ple in this section uses a	n intracommunicator.	43

#### Example 5.24

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

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10 11 12

13 14 15

16

44

45

46

1		_Comm comm;					
2	int array1[100], array2[100];						
3	<pre>int root=0;</pre>						
4	MPI_Request req;						
5	•••		· ··· · · ·				
6		•	<pre>O, MPI_INT, root, comm, &amp;req);</pre>				
7		<pre>pute(array2, 100);</pre>					
8 9	MPI	_Wait(&req, MPI_ST	ATUS_IGNURE);				
10	F 10 0						
11	5.12.3	Nonblocking Gather					
12							
13							
14 15	MPI_IGA	request)	ount, sendtype, recvbuf, recvcount, recvtype, root, comm,				
16	IN	sendbuf	starting address of send buffer (choice)				
17 18	IN	sendcount	number of elements in send buffer (non-negative integer)				
19	IN	sendtype	data type of send buffer elements (handle)				
20 21	OUT	recvbuf	address of receive buffer (choice, significant only at				
21			root)				
23 24	IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)				
25 26 27	IN	recvtype	data type of recv buffer elements (significant only at root) (handle)				
28	IN	root	rank of receiving process (integer)				
29	IN	comm	communicator (handle)				
30 31	OUT	request	communication request (handle)				
32	int MDT	T					
33	int MPI	-	d* sendbuf, int sendcount, MPI_Datatype sendtype,				
34			<pre>f, int recvcount, MPI_Datatype recvtype, int root,     MDI_Beguart traggest)</pre>				
35		MP1_Comm Com	m, MPI_Request *request)				
36	MPI_Iga	ther(sendbuf, send	count, sendtype, recvbuf, recvcount, recvtype,				
37			request, ierror)				
38			), INTENT(IN), ASYNCHRONOUS :: sendbuf				
39			), ASYNCHRONOUS :: recvbuf				
40 41			: sendcount, recvcount, root				
41		• -	NTENT(IN) :: sendtype, recvtype				
42		E(MPI_Comm), INTEN					
44		-	TENT(OUT) :: request				
45	TNT	EGER, UPIIUNAL, IN	<pre>IENT(OUT) :: ierror</pre>				
46	MPI_IGA	THER(SENDBUF, SEND	COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,				
47			REQUEST, IERROR)				
48	<ty< th=""><th>pe&gt; SENDBUF(*), REG</th><th>CVBUF(*)</th></ty<>	pe> SENDBUF(*), REG	CVBUF(*)				

INTH IERH		E, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,	1 2
This	call starts a nonblocking va	ariant of $MPI_GATHER$ (see Section 5.5).	$\frac{3}{4}$
			5
MPI_IGA	THERV(sendbuf, sendcount, comm, request)	, sendtype, recvbuf, recvcounts, displs, recvtype, root,	6 7
IN	sendbuf	starting address of send buffer (choice)	8 9
IN	sendcount	number of elements in send buffer (non-negative integer)	10 11
IN	sendtype	data type of send buffer elements (handle)	12
OUT	recvbuf	address of receive buffer (choice, significant only at root)	13 14 15
IN	recvcounts	non-negative integer array (of length group size) con- taining the number of elements that are received from each process (significant only at root)	16 17 18
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to <b>recvbuf</b> at which to place the incoming data from process i (significant only at root)	19 20 21 22 23
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	23 24 25
IN	root	rank of receiving process (integer)	26
IN	comm	communicator (handle)	27
OUT	request	communication request (handle)	28 29
int MPI_	void* recvbuf, co	endbuf, int sendcount, MPI_Datatype sendtype, nst int recvcounts[], const int displs[], type, int root, MPI_Comm comm, est)	30 31 32 33 34
TYPH TYPH INTH INTH TYPH TYPH INTH	recvtype, root, c E(*), DIMENSION(), IN E(*), DIMENSION(), AS EGER, INTENT(IN) :: ser EGER, INTENT(IN), ASYNCH E(MPI_Datatype), INTENT E(MPI_Comm), INTENT(IN) E(MPI_Comm), INTENT(IN) E(MPI_Request), INTENT(I EGER, OPTIONAL, INTENT(I)	ndcount, root HRONOUS :: recvcounts(*), displs(*) (IN) :: sendtype, recvtype :: comm DUT) :: request	35 36 37 38 39 40 41 42 43 44 45 46
		OMM, REQUEST, IERROR)	47 48

```
204
                                          CHAPTER 5. COLLECTIVE COMMUNICATION
1
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
\mathbf{2}
          COMM, REQUEST, IERROR
3
         This call starts a nonblocking variant of MPI_GATHERV (see Section 5.5).
4
5
     5.12.4 Nonblocking Scatter
6
7
8
9
     MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
                     request)
10
11
       IN
                  sendbuf
                                             address of send buffer (choice, significant only at root)
12
       IN
                  sendcount
                                             number of elements sent to each process (non-negative
13
                                             integer, significant only at root)
14
15
       IN
                  sendtype
                                             data type of send buffer elements (significant only at
16
                                             root) (handle)
17
       OUT
                  recvbuf
                                             address of receive buffer (choice)
18
       IN
                                             number of elements in receive buffer (non-negative in-
                  recvcount
19
                                             teger)
20
       IN
                                             data type of receive buffer elements (handle)
21
                  recvtype
22
       IN
                  root
                                             rank of sending process (integer)
23
       IN
                  comm
                                             communicator (handle)
24
       OUT
25
                 request
                                             communication request (handle)
26
27
     int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
28
                     void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
29
                    MPI_Comm comm, MPI_Request *request)
30
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
^{31}
                    root, comm, request, ierror)
32
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
33
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
34
          INTEGER, INTENT(IN) :: sendcount, recvcount, root
35
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
36
          TYPE(MPI_Comm), INTENT(IN) :: comm
37
          TYPE(MPI_Request), INTENT(OUT) :: request
38
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
41
                    ROOT, COMM, REQUEST, IERROR)
42
          <type> SENDBUF(*), RECVBUF(*)
43
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
44
          IERROR
45
         This call starts a nonblocking variant of MPI_SCATTER (see Section 5.6).
46
47
48
```

	comm, request)	ints, displs, sendtype, recvbuf, recvcount, recvtype, root,
IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative in-teger)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)
	-	- X /
int MPI_	Iscatterv(const void*	sendbuf, const int sendcounts[],
	const int displa	s[], MPI_Datatype sendtype, void* recvbuf,
	-	s[], MPI_Datatype sendtype, void* recvbuf, MPI_Datatype recvtype, int root, MPI_Comm comm,
	-	MPI_Datatype recvtype, int root, MPI_Comm comm,
MPI Iscat	int recvcount, M MPI_Request *rec	MPI_Datatype recvtype, int root, MPI_Comm comm, quest)
MPI_Iscat	int recvcount, M MPI_Request *rec tterv(sendbuf, sendco	MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount,
	int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root,	MPI_Datatype recvtype, int root, MPI_Comm comm, quest)
TYPE	int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root,	MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE TYPE	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco     recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A</pre>	MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE TYPE INTE	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco     recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*)</pre>
TYPE TYPE INTE INTE	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN) :: r</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*)</pre>
TYPE TYPE INTE INTE TYPE TYPE	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN)</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf ENTENT(IN), ASYNCHRONOUS :: sendbuf ICHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*) recvcount, root IT(IN) :: sendtype, recvtype I) :: comm</pre>
TYPE TYPE INTE INTE TYPE TYPE TYPE	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN (MPI_Request), INTENT</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*) recvcount, root IT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request</pre>
TYPE TYPE INTE INTE TYPE TYPE TYPE	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN)</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*) recvcount, root IT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request</pre>
TYPE TYPE INTE INTE TYPE TYPE INTE	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN (MPI_Request), INTENT GER, OPTIONAL, INTENT</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*) recvcount, root IT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request C(OUT) :: ierror</pre>
TYPE TYPE INTEC INTEC TYPE TYPE INTEC	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN (MPI_Request), INTENT GER, OPTIONAL, INTENT TTERV(SENDBUF, SENDCO</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) INTENT(IN), ASYNCHRONOUS :: sendbuf INTENT(IN), ASYNCHRONOUS :: sendbuf INTENT(IN), ASYNCHRONOUS :: sendbuf ICHRONOUS :: sendcounts(*), displs(*) recvcount, root IT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request C(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,</pre>
TYPE TYPE INTE INTE TYPE TYPE INTE MPI_ISCAT	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN (MPI_Request), INTENT GER, OPTIONAL, INTENT TTERV(SENDBUF, SENDCO RECVTYPE, ROOT,</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*) recvcount, root IT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request C(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, REQUEST, IERROR)</pre>
TYPE TYPE INTEC INTEC TYPE TYPE INTEC MPI_ISCAT	<pre>int recvcount, M MPI_Request *req tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN (MPI_Request), INTENT GER, OPTIONAL, INTENT GER, OPTIONAL, INTENT TTERV(SENDBUF, SENDCO RECVTYPE, ROOT, e&gt; SENDBUF(*), RECVBU</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf ENTENT(IN), ASYNCHRONOUS :: sendbuf EXYNCHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*) recvcount, root IT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request C(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, REQUEST, IERROR) IF(*)</pre>
TYPE TYPE INTE INTE TYPE TYPE INTE MPI_ISCA <type INTE</type 	<pre>int recvcount, M MPI_Request *req tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN (MPI_Request), INTENT GER, OPTIONAL, INTENT GER, OPTIONAL, INTENT TTERV(SENDBUF, SENDCO RECVTYPE, ROOT, e&gt; SENDBUF(*), RECVBU</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*) recvcount, root IT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request C(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, REQUEST, IERROR)</pre>
TYPE TYPE INTEC TYPE TYPE INTEC MPI_ISCAT	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN (MPI_Comm), INTENT(IN (MPI_Request), INTENT GER, OPTIONAL, INTENT TTERV(SENDBUF, SENDCO RECVTYPE, ROOT, e&gt; SENDBUF(*), RECVBU GER SENDCOUNTS(*), DI , REQUEST, IERROR</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) punts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: recvbuf NCHRONOUS :: sendcounts(*), displs(*) recvcount, root NT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request C(OUT) :: ierror NUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, REQUEST, IERROR) FF(*) SPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,</pre>
TYPE TYPE INTEC TYPE TYPE INTEC MPI_ISCAT	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN (MPI_Comm), INTENT(IN (MPI_Request), INTENT GER, OPTIONAL, INTENT TTERV(SENDBUF, SENDCO RECVTYPE, ROOT, e&gt; SENDBUF(*), RECVBU GER SENDCOUNTS(*), DI , REQUEST, IERROR</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) ounts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf ENTENT(IN), ASYNCHRONOUS :: sendbuf EXYNCHRONOUS :: recvbuf ICHRONOUS :: sendcounts(*), displs(*) recvcount, root IT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request C(OUT) :: ierror DUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, REQUEST, IERROR) IF(*)</pre>
TYPE TYPE INTEC TYPE TYPE INTEC MPI_ISCAT	<pre>int recvcount, M MPI_Request *rec tterv(sendbuf, sendco recvtype, root, (*), DIMENSION(), I (*), DIMENSION(), A GER, INTENT(IN), ASYN GER, INTENT(IN), ASYN GER, INTENT(IN) :: r (MPI_Datatype), INTEN (MPI_Comm), INTENT(IN (MPI_Comm), INTENT(IN (MPI_Request), INTENT GER, OPTIONAL, INTENT TTERV(SENDBUF, SENDCO RECVTYPE, ROOT, e&gt; SENDBUF(*), RECVBU GER SENDCOUNTS(*), DI , REQUEST, IERROR</pre>	<pre>MPI_Datatype recvtype, int root, MPI_Comm comm, quest) punts, displs, sendtype, recvbuf, recvcount, comm, request, ierror) ENTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: recvbuf NCHRONOUS :: sendcounts(*), displs(*) recvcount, root NT(IN) :: sendtype, recvtype I) :: comm C(OUT) :: request C(OUT) :: ierror NUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, COMM, REQUEST, IERROR) FF(*) SPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,</pre>

```
206
                                         CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.12.5
             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
8
       IN
                 sendcount
                                             number of elements in send buffer (non-negative inte-
9
                                             ger)
10
       IN
                 sendtype
                                             data type of send buffer elements (handle)
11
       OUT
                  recvbuf
                                             address of receive buffer (choice)
12
       IN
                                             number of elements received from any process (non-
13
                  recvcount
14
                                             negative integer)
15
       IN
                  recvtype
                                             data type of receive buffer elements (handle)
16
       IN
                                             communicator (handle)
                 comm
17
       OUT
18
                 request
                                             communication request (handle)
19
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
     MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
24
                    comm, request, ierror)
25
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
          INTEGER, INTENT(IN) :: sendcount, recvcount
28
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
          TYPE(MPI_Comm), INTENT(IN) :: comm
30
          TYPE(MPI_Request), INTENT(OUT) ::
                                                  request
^{31}
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
32
33
     MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
34
                    COMM, REQUEST, IERROR)
35
          <type> SENDBUF(*), RECVBUF(*)
36
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
37
         This call starts a nonblocking variant of MPI_ALLGATHER (see Section 5.7).
38
39
40
41
42
43
44
45
46
47
48
```

	sendtype, recvbuf, recvcounts, displs, recvtype, comm,	1 2
sendbuf	starting address of send buffer (choice)	3
sendcount	number of elements in send buffer (non-negative integer)	4 5 6
sendtype	data type of send buffer elements (handle)	7
recvbuf	address of receive buffer (choice)	8 9
recvcounts	non-negative integer array (of length group size) con- taining the number of elements that are received from each process	9 10 11 12
displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	13 14 15
recvtype	data type of receive buffer elements (handle)	16 17
comm	communicator (handle)	18
request	communication request (handle)	19
<pre>MPI_Datatype sendtyp const int displs[], MPI_Request* request atherv(sendbuf, sendcount recvtype, comm, requ *), DIMENSION(), INTENT *), DIMENSION(), ASYNCH ER, INTENT(IN) :: sendco ER, INTENT(IN) :: sendco ER, INTENT(IN), ASYNCHRON MPI_Datatype), INTENT(IN) MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT) ER, OPTIONAL, INTENT(OUT) ATHERV(SENDBUF, SENDCOUNT RECVTYPE, COMM, REQU &gt; SENDBUF(*), RECVBUF(*) ER SENDCOUNT, SENDTYPE, I ST, IERROR</pre>	<pre>e, void* recvbuf, const int recvcounts[], MPI_Datatype recvtype, MPI_Comm comm, ) t, sendtype, recvbuf, recvcounts, displs, est, ierror) T(IN), ASYNCHRONOUS :: sendbuf HRONOUS :: recvbuf ount NOUS :: recvcounts(*), displs(*) ) :: sendtype, recvtype comm ) :: request ) :: ierror T, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, EST, IERROR) RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,</pre>	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
	<pre>request) sendbuf sendcount sendtype recvbuf recvcounts displs  recvtype comm request allgatherv(const void* se MPI_Datatype sendtyp const int displs[], MPI_Request* request atherv(sendbuf, sendcounf recvtype, comm, request atherv(sendbuf, sendcounf recvtype, comm, request atherv(sendbuf, sendcounf recvtype, comm, request), DIMENSION(), INTENT *), DIMENSION(), ASYNCHROF ER, INTENT(IN) :: sendce ER, INTENT(IN) :: sendce ER, INTENT(IN), ASYNCHROF MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT ER, OPTIONAL, INTENT(OUT ATHERV(SENDBUF, SENDCOUNF, RECVTYPE, COMM, REQU &gt; SENDBUF(*), RECVBUF(*) ER SENDCOUNT, SENDTYPE, INTENT(*) ST, IERROR</pre>	<pre>sendbuf starting address of send buffer (choice) sendcount number of elements in send buffer (non-negative inte- ger) sendtype data type of send buffer elements (handle) recvbuf address of receive buffer (choice) recvcounts non-negative integer array (of length group size) con- taining the number of elements that are received from each process displs integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i recvtype data type of receive buffer elements (handle) comm communicator (handle) request communication request (handle) allgatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request* request) atherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror) *), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf eR, INTENT(IN) :: sendtype, recvtype MPI_Comm), INTENT(IN) :: comm MPI_Request), INTENT(IN) :: request ER, OPTIONAL, INTENT(OUT) :: ierror ATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVENUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) &gt; SENDBUF(*), RECVENUF(*) ER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), RECVTYPE, COMM,</pre>

	208		CHAPTER 5. COLLECTIVE COMMUNICATION
1	5.12.6	Nonblocking All-to-All Scat	ter/Gather
2	5.12.0	Nonbioeking / II to / II Seat	
3			
4 5	MPI_IA	LLTOALL(sendbuf, sendcount,	sendtype, recvbuf, recvcount, recvtype, comm, request)
6			
7	IN	sendbuf	starting address of send buffer (choice)
8 9	IN	sendcount	number of elements sent to each process (non-negative integer)
10 11	IN	sendtype	data type of send buffer elements (handle)
11	OUT	recvbuf	address of receive buffer (choice)
13 14	IN	recvcount	number of elements received from any process (non-negative integer)
15	IN	recvtype	data type of receive buffer elements (handle)
16 17	IN	comm	communicator (handle)
18	OUT	request	communication request (handle)
19			
20	int MP	I_Ialltoall(const void* s	sendbuf, int sendcount,

21

22

23

32

```
int MPI_Ialltoall(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

```
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
24
                  comm, request, ierror)
25
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
         INTEGER, INTENT(IN) :: sendcount, recvcount
28
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         TYPE(MPI_Request), INTENT(OUT) :: request
^{31}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
33
     MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
34
                    COMM, REQUEST, IERROR)
35
         <type> SENDBUF(*), RECVBUF(*)
36
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
37
         This call starts a nonblocking variant of MPI_ALLTOALL (see Section 5.8).
38
39
40
41
42
```

MPI_	IALLTOALLV(sendbuf, sendcounts, recvtype, comm, request	sdispls, sendtype, recvbuf, recvcounts, rdispls, )	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	4 5 6
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	7 8 9
IN	sendtype	data type of send buffer elements (handle)	10 11
OU	T recvbuf	address of receive buffer (choice)	12
IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each rank	13 14 15
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	16 17 18 19
IN	recvtype	data type of receive buffer elements (handle)	20
IN	comm	communicator (handle)	21
OU	T request	communication request (handle)	22 23
int 1	<pre>const int sdispls[], const int recvcounts</pre>	ndbuf, const int sendcounts[], MPI_Datatype sendtype, void* recvbuf, s[], const int rdispls[], pe, MPI_Comm comm, MPI_Request *request)	24 25 26 27 28
MPI_	rdispls, recvtype, c TYPE(*), DIMENSION(), INTEN TYPE(*), DIMENSION(), ASYNCH INTEGER, INTENT(IN), ASYNCHRON recvcounts(*), rdispls(*) TYPE(MPI_Datatype), INTENT(IN TYPE(MPI_Comm), INTENT(IN) :: TYPE(MPI_Comm), INTENT(IN) :: TYPE(MPI_Request), INTENT(OUT INTEGER, OPTIONAL, INTENT(OUT IALLTOALLV(SENDBUF, SENDCOUNT RDISPLS, RECVTYPE, C Ktype> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPL RECVTYPE, COMM, REQUEST, IERR	<pre>HRONOUS :: recvbuf NOUS :: sendcounts(*), sdispls(*), ) :: sendtype, recvtype comm ) :: request ) :: ierror S, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, COMM, REQUEST, IERROR) S(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
			48

MPLIALITOALIV(sendbuf sendcounts sdispls sendtype recybuf recycounts rdispls

12	MPI_IALL1	OALLW(sendbuf, sendcounts, recvtypes, comm, request	sdispls, sendtypes, recvbuf, recvcounts, rdispls, )
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6 7	IN	sendcounts	integer array (of length group size) specifying the num- ber of elements to send to each rank (array of non- negative integers)
8 9 10 11	IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
12 13 14 15	IN	sendtypes	array of datatypes (of length group size). Entry j spec- ifies the type of data to send to process j (array of handles)
16	OUT	recvbuf	address of receive buffer (choice)
17 18 19	IN	recvcounts	integer array (of length group size) specifying the num- ber of elements that can be received from each rank (array of non-negative integers)
20 21 22 23 24	IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
25 26 27	IN	recvtypes	array of datatypes (of length group size). Entry i spec- ifies the type of data received from process i (array of handles)
28	IN	comm	communicator (handle)
29 30	OUT	request	communication request (handle)
31 32 33 34 35 36	int MPI_I	<pre>const int sdispls[], void* recvbuf, const</pre>	ndbuf, const int sendcounts[], const MPI_Datatype sendtypes[], int recvcounts[], const int rdispls[], ecvtypes[], MPI_Comm comm, )
36 37	MPI_Iallt		, sdispls, sendtypes, recvbuf,
38	recvcounts, rdispls, recvtypes, comm, request, ierror)		
39 40	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf		
41			
42	<pre>INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*), recvcounts(*), rdispls(*)</pre>		
43			), ASYNCHRONOUS :: sendtypes(*),
44		<pre>ypes(*)</pre>	
45 46		<pre>MPI_Comm), INTENT(IN) :: MDI_Document) INTENT(OUT)</pre>	
47		<pre>MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)</pre>	-
48	111110	The second se	

MPI_IALLTC	-	S, SDISPLS, SENDTYPES, RECVBUF,
		RECVTYPES, COMM, REQUEST, IERROR)
• -	SENDBUF(*), RECVBUF(*)	(*), SENDTYPES(*), RECVCOUNTS(*),
	S(*), RECVTYPES(*), COMM	
This ca	all starts a nonblocking varian	t of MPI_ALLTOALLW (see Section $5.8$ ).
		٤
5.12.7 No	nblocking Reduce	9
		1
	ICE (conditioned wave but count of	1
		atatype, op, root, comm, request) 1
IN	sendbuf	address of send buffer (choice)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	count	number of elements in send buffer (non-negative inte- ger) 1
IN	datatype	data type of elements of send buffer (handle)
IN	ор	reduce operation (handle)
IN	root	rank of root process (integer) 2
IN	comm	communicator (handle) 2
OUT	request	communication request (handle)
		2
int MPI_Ir		f, void* recvbuf, int count, 2
	MPI_Datatype datatype MPI_Request *request)	e, MPI_Op op, int root, MPI_Comm comm, 2 2 2
MPI_Ireduc	e(sendbuf, recvbuf, coun ierror)	t, datatype, op, root, comm, request, $3 \atop 3$
TYPE(*	<pre>(), DIMENSION(), INTENT</pre>	(IN), ASYNCHRONOUS :: sendbuf
	<pre>&gt;, DIMENSION(), ASYNCH</pre>	0
	ER, INTENT(IN) :: count,	2
TYPE(M	<pre>IPI_Datatype), INTENT(IN)</pre>	
	<pre>IPI_Op), INTENT(IN) :: c</pre>	•
	<pre>IPI_Comm), INTENT(IN) ::</pre>	-
	PI_Request), INTENT(OUT)	:: request
INTEGE	ER, OPTIONAL, INTENT(OUT)	:: ierror 4
MPI_IREDUC	CE(SENDBUF, RECVBUF, COUN	T, DATATYPE, OP, ROOT, COMM, REQUEST, 4
	IERROR)	4
• -	<pre>SENDBUF(*), RECVBUF(*) </pre>	
INTEGE	LE CUUNI, DATATYPE, UP, R	COT, COMM, REQUEST, IERROR 4
This ca	ll starts a nonblocking variar	t of MPI_REDUCE (see Section 5.9.1). $4$
	-	plementation is explicitly allowed to use different <sup>4</sup> ocking reduction operations that might change the <sup>4</sup>

```
212
                                          CHAPTER 5. COLLECTIVE COMMUNICATION
1
           order of evaluation of the operations. However, as for MPI_REDUCE, it is strongly
\mathbf{2}
           recommended that MPI_IREDUCE be implemented so that the same result be obtained
3
           whenever the function is applied on the same arguments, appearing in the same order.
4
           Note that this may prevent optimizations that take advantage of the physical location
5
           of processes. (End of advice to implementors.)
6
           Advice to users. For operations which are not truly associative, the result delivered
7
           upon completion of the nonblocking reduction may not exactly equal the result deliv-
8
           ered by the blocking reduction, even when specifying the same arguments in the same
9
           order. (End of advice to users.)
10
11
12
     5.12.8 Nonblocking All-Reduce
13
14
15
     MPI_IALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, request)
16
       IN
                 sendbuf
                                             starting address of send buffer (choice)
17
18
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
19
       IN
                 count
                                             number of elements in send buffer (non-negative inte-
20
                                             ger)
21
       IN
                 datatype
                                             data type of elements of send buffer (handle)
22
23
       IN
                 ор
                                              operation (handle)
24
       IN
                 comm
                                             communicator (handle)
25
       OUT
                                              communication request (handle)
                 request
26
27
     int MPI_Iallreduce(const void* sendbuf, void* recvbuf, int count,
28
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
29
                     MPI_Request *request)
30
^{31}
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
32
                     ierror)
33
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS ::
                                                                       sendbuf
34
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
35
          INTEGER, INTENT(IN) :: count
36
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
          TYPE(MPI_Op), INTENT(IN) :: op
38
          TYPE(MPI_Comm), INTENT(IN) :: comm
39
          TYPE(MPI_Request), INTENT(OUT) ::
                                                  request
40
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
41
     MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
42
                     IERROR)
43
          <type> SENDBUF(*), RECVBUF(*)
44
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
45
46
          This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 5.9.6).
47
48
```

5.12.	NONBLOCKING COLI	LECTIVE OPERATIONS	213
5.12.9	Nonblocking Reduce-	Scatter with Equal Blocks	1
MPI_I	REDUCE_SCATTER_BL	OCK(sendbuf, recvbuf, recvcount, datatype, op, comm, requ	3 Jest) 4 5
IN	sendbuf	starting address of send buffer (choice)	6 7
OUT	recvbuf	starting address of receive buffer (choice)	8
IN	recvcount	element count per block (non-negative integer)	9
IN	datatype	data type of elements of send and receive buffers (adle)	10 han- 11 12
IN	ор	operation (handle)	13
IN	comm	communicator (handle)	14
OUT	request	communication request (handle)	15 16
MPI_I	int recvcoun MPI_Comm com reduce_scatter_block( request, ier		18 19 20 21 22 23
T I T T T	YPE(*), DIMENSION() NTEGER, INTENT(IN) :: YPE(MPI_Datatype), IN YPE(MPI_Op), INTENT(] YPE(MPI_Comm), INTENT	: recvcount NTENT(IN) :: datatype IN) :: op F(IN) :: comm FENT(OUT) :: request	24 25 26 27 28 29 30 31
<	REQUEST, IER type> SENDBUF(*), REC		
	This call starts a nonbloc $.10.1$ ).	king variant of MPI_REDUCE_SCATTER_BLOCK (see a	36 37 38
			39 40
			41 42
			43
			44
			45
			46 47
			48

```
214
                                         CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.12.10
              Nonblocking Reduce-Scatter
\mathbf{2}
3
4
     MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
7
                 recvbuf
                                             starting address of receive buffer (choice)
8
       IN
                 recvcounts
                                             non-negative integer array specifying the number of
9
                                             elements in result distributed to each process. Array
10
                                             must be identical on all calling processes.
11
       IN
                 datatype
                                             data type of elements of input buffer (handle)
12
       IN
                                             operation (handle)
13
                 ор
14
       IN
                 comm
                                             communicator (handle)
15
       OUT
                 request
                                             communication request (handle)
16
17
     int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf,
18
                    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
19
                    MPI_Comm comm, MPI_Request *request)
20
21
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
22
                    request, ierror)
23
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
^{24}
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
25
          INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          TYPE(MPI_Op), INTENT(IN) :: op
28
          TYPE(MPI_Comm), INTENT(IN) :: comm
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
^{31}
     MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
32
                    REQUEST, IERROR)
33
34
          <type> SENDBUF(*), RECVBUF(*)
          INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
35
36
         This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 5.10.2).
37
38
39
40
41
42
43
44
45
46
47
48
```

# 5.12.11 Nonblocking Inclusive Scan

IPI_ISC	AN(sendbut, recvbut, col	unt, datatype, op, comm, request)
IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative in-teger)
IN	datatype	data type of elements of input buffer (handle)
IN	ор	operation (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)
IU MPI		endbuf, void* recvbuf, int count, datatype, MPI_Op op, MPI_Comm comm, request)
TYPI	E(*), DIMENSION(),	<pre>count, datatype, op, comm, request, ierror) INTENT(IN), ASYNCHRONOUS :: sendbuf</pre>
11111	E(*) DIMENSION().	ASYNCHRONOUS :: recybuf
	E(*), DIMENSION(), EGER, INTENT(IN) ::	ASYNCHRONOUS :: recvbuf count
INTI TYPI	EGER, INTENT(IN) :: E(MPI_Datatype), INT	count ENT(IN) :: datatype
INTI TYPI TYPI	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN	count ENT(IN) :: datatype ) :: op
INTI TYPI TYPI TYPI	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT(	count ENT(IN) :: datatype ) :: op IN) :: comm
INTI TYPI TYPI TYPI TYPI	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE	count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request
INTI TYPI TYPI TYPI TYPI INTI	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror</pre>
INTI TYPI TYPI TYPI TYPI INTI INTI	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF,	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm ENT(OUT) :: request ENT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)</pre>
INTI TYPI TYPI TYPI INTI INTI (PI_ISC) <tyj< td=""><td>EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe&gt; SENDBUF(*), RECV</td><td><pre>count ENT(IN) :: datatype ) :: op IN) :: comm ENT(OUT) :: request ENT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)</pre></td></tyj<>	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm ENT(OUT) :: request ENT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)</pre>
INTI TYPI TYPI TYPI INTI IPI_ISCA <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>
INTI TYPI TYPI TYPI INTI INTI (PI_ISC) <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*)</pre>
INTI TYPI TYPI TYPI INTI IPI_ISCA <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>
INTI TYPI TYPI TYPI INTI INTI IPI_ISCA <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>
INTI TYPI TYPI TYPI INTI INTI SPI_ISCA <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>
INTI TYPI TYPI TYPI INTI PI_ISCA <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>
INTI TYPI TYPI TYPI INTI PI_ISCA <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>
INTI TYPI TYPI TYPI INTI PI_ISCA <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>
INTI TYPI TYPI TYPI INTI PI_ISCA <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>
INTI TYPI TYPI TYPI INTI INTI (PI_ISC) <tyj INTI</tyj 	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>
INTI TYPI TYPI TYPI INTI INTI (ty) INTI	EGER, INTENT(IN) :: E(MPI_Datatype), INT E(MPI_Op), INTENT(IN E(MPI_Comm), INTENT( E(MPI_Request), INTE EGER, OPTIONAL, INTE AN(SENDBUF, RECVBUF, pe> SENDBUF(*), RECV EGER COUNT, DATATYPE	<pre>count ENT(IN) :: datatype ) :: op IN) :: comm NT(OUT) :: request NT(OUT) :: ierror COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) BUF(*) ;, OP, COMM, REQUEST, IERROR</pre>

1

```
216
                                          CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.12.12 Nonblocking Exclusive Scan
\mathbf{2}
3
4
     MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
7
                 recvbuf
                                             starting address of receive buffer (choice)
8
       IN
                 count
                                              number of elements in input buffer (non-negative in-
9
                                              teger)
10
       IN
                                              data type of elements of input buffer (handle)
                 datatype
11
       IN
                                              operation (handle)
12
                 op
13
       IN
                 comm
                                             intracommunicator (handle)
14
       OUT
                                             communication request (handle)
                 request
15
16
     int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count,
17
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
18
                     MPI_Request *request)
19
20
     MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
21
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS ::
                                                                       sendbuf
22
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
23
          INTEGER, INTENT(IN) :: count
^{24}
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          TYPE(MPI_Op), INTENT(IN) :: op
26
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
          TYPE(MPI_Request), INTENT(OUT) :: request
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
29
     MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
30
          <type> SENDBUF(*), RECVBUF(*)
^{31}
          INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
32
33
          This call starts a nonblocking variant of MPI_EXSCAN (see Section 5.11.2).
34
35
     5.13
             Correctness
36
37
     A correct, portable program must invoke collective communications so that deadlock will not
38
     occur, whether collective communications are synchronizing or not. The following examples
39
     illustrate dangerous use of collective routines on intracommunicators.
40
41
     Example 5.25
42
          The following is erroneous.
43
44
45
46
47
48
```

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

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

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

#### Example 5.26

The following is erroneous.

```
switch(rank) {
   case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
   case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
   case 2:
        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
   case 3:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
   case 4:
        MPI_Bcast(buf1, count, type, 1, comm1);
        break;
   case 5:
        MPI_Bcast(buf1, count, type, 1, comm1);
   case 5:
   case 5:
   case 5:
   case 5:
   case 5:
   case 5:
   case
```

}

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

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

#### Example 5.27

The following is erroneous.

Unofficial Draft for Comment Only

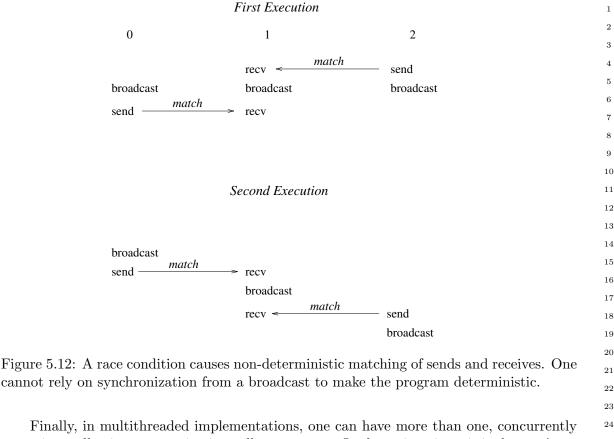
 $^{24}$ 

 $^{31}$ 

 $41 \\ 42$ 

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

<sup>46</sup> example illustrates the fact that one should not rely on collective communication functions
 <sup>47</sup> to have particular synchronization effects. A program that works correctly only when the
 <sup>48</sup> first execution occurs (only when broadcast is synchronizing) is erroneous.



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

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

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

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

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

#### Example 5.29

Blocking and nonblocking collective operations can be interleaved, i.e., a blocking collective operation can be posted even if there is a nonblocking collective operation outstanding.

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

 $^{24}$ 

```
1
     MPI_Request req;
\mathbf{2}
3
     MPI_Ibarrier(comm, &req);
4
     MPI_Bcast(buf1, count, type, 0, comm);
5
     MPI_Wait(&req, MPI_STATUS_IGNORE);
6
          Each process starts a nonblocking barrier operation, participates in a blocking broad-
7
     cast and then waits until every other process started the barrier operation. This ef-
8
     fectively turns the broadcast into a synchronizing broadcast with possible communica-
9
     tion/communication overlap (MPI_Bcast is allowed, but not required to synchronize).
10
11
     Example 5.30
12
          The starting order of collective operations on a particular communicator defines their
13
     matching. The following example shows an erroneous matching of different collective oper-
14
     ations on the same communicator.
15
16
     MPI_Request req;
17
     switch(rank) {
18
          case 0:
19
              /* erroneous matching */
20
              MPI_Ibarrier(comm, &req);
21
              MPI_Bcast(buf1, count, type, 0, comm);
22
              MPI_Wait(&req, MPI_STATUS_IGNORE);
23
              break;
24
          case 1:
25
              /* erroneous matching */
26
              MPI_Bcast(buf1, count, type, 0, comm);
27
              MPI_Ibarrier(comm, &req);
28
              MPI_Wait(&reg, MPI_STATUS_IGNORE);
29
              break;
30
     }
^{31}
32
          This ordering would match MPI_Ibarrier on rank 0 with MPI_Bcast on rank 1 which is
33
     erroneous and the program behavior is undefined. However, if such an order is required, the
34
     user must create different duplicate communicators and perform the operations on them.
35
     If started with two processes, the following program would be correct:
36
37
     MPI_Request req;
38
     MPI_Comm dupcomm;
39
     MPI_Comm_dup(comm, &dupcomm);
40
     switch(rank) {
41
          case 0:
42
              MPI_Ibarrier(comm, &req);
              MPI_Bcast(buf1, count, type, 0, dupcomm);
43
44
              MPI_Wait(&req, MPI_STATUS_IGNORE);
45
              break;
46
          case 1:
47
              MPI_Bcast(buf1, count, type, 0, dupcomm);
48
              MPI_Ibarrier(comm, &req);
```

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

}

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

#### Example 5.31

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

```
MPI_Request req;
```

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

```
}
```

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

### Example 5.32

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

```
MPI_Request req;
```

```
switch(rank) {
                                                                                    38
                                                                                    39
    case 0:
      /* erroneous false matching of Alltoall and Ialltoall */
                                                                                    40
                                                                                    41
      MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
                                                                                    42
      MPI_Wait(&req, MPI_STATUS_IGNORE);
      break;
                                                                                    43
                                                                                    44
    case 1:
      /* erroneous false matching of Alltoall and Ialltoall */
                                                                                    45
      MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
                                                                                    46
                                                                                    47
      break;
                                                                                    48
}
```

 $\mathbf{2}$ 

 $^{24}$ 

```
1
     Example 5.33
\mathbf{2}
          Collective and point-to-point requests can be mixed in functions that enable multiple
3
     completions. If started with two processes, the following program is valid.
4
     MPI_Request reqs[2];
5
6
     switch(rank) {
7
8
          case 0:
            MPI_Ibarrier(comm, &reqs[0]);
9
            MPI_Send(buf, count, dtype, 1, tag, comm);
10
            MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
11
            break;
12
          case 1:
13
            MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
14
            MPI_Ibarrier(comm, &reqs[1]);
15
16
            MPI_Waitall(2, regs, MPI_STATUSES_IGNORE);
17
            break;
     }
18
19
          The MPI_Waitall call returns only after the barrier and the receive completed.
20
21
     Example 5.34
22
          Multiple nonblocking collective operations can be outstanding on a single communicator
23
     and match in order.
24
25
     MPI_Request reqs[3];
26
27
     compute(buf1);
28
     MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
29
     compute(buf2);
30
     MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
31
     compute(buf3);
32
     MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
33
     MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
34
35
           Advice to users. Pipelining and double-buffering techniques can efficiently be used
36
           to overlap computation and communication. However, having too many outstanding
37
           requests might have a negative impact on performance. (End of advice to users.)
38
39
                                      The use of pipelining may generate many outstanding
           Advice to implementors.
40
           requests. A high-quality hardware-supported implementation with limited resources
41
           should be able to fall back to a software implementation if its resources are exhausted.
42
           In this way, the implementation could limit the number of outstanding requests only
43
           by the available memory. (End of advice to implementors.)
44
45
46
     Example 5.35
47
48
```

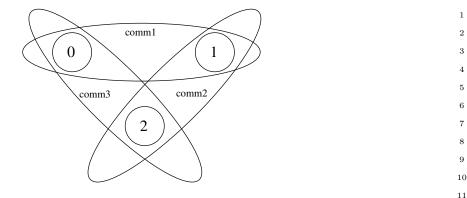


Figure 5.13: Example with overlapping communicators.

exists no deadlock-free order to invoke them. However, nonblocking collective operations

Nonblocking collective operations can also be used to enable simultaneous collective operations on multiple overlapping communicators (see Figure 5.13). The following example is started with three processes and three communicators. The first communicator comm1 includes ranks 0 and 1, comm2 includes ranks 1 and 2, and comm3 spans ranks 0 and 2. It is not possible to perform a blocking collective operation on all communicators because there

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

can easily be used to achieve this task.

```
switch(rank) {
    case 0:
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
                                                                                  27
      break;
                                                                                  28
    case 1:
                                                                                  29
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
                                                                                  30
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[1]);
      break;
                                                                                  33
    case 2:
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[0]);
                                                                                  34
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
                                                                                  35
      break;
                                                                                  36
}
                                                                                  37
MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
                                                                                  38
```

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

#### Example 5.36

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

## **Unofficial Draft for Comment Only**

1213 14

15

16

17

18

19

2021

22 23

> $^{24}$ 25

> 26

31

32

39

40

41

42

43 4445

46

47

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

## Chapter 6

# Groups, Contexts, Communicators, and Caching

## 6.1 Introduction

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

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

#### 6.1.1 Features Needed to Support Libraries

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

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

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

 $^{24}$ 

## 6.1.2 MPI's Support for Libraries

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

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

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

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

24

1

2

3

4 5

6

7 8

9 10

11

12

18

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

31

36

37

38

39

40

41

42

43

44

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

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

45 46 47

• **Groups** define the participants in the communication (see above) of a communicator.

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

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

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

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

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

#### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

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

## 6.2 Basic Concepts

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

6.2.1 Groups

1

 $\mathbf{2}$ 

3 4

5 6

> 7 8

9

20 21

22

23 24

25

26

27

28

34

35

<sup>10</sup> A **group** is an ordered set of process identifiers (henceforth processes); processes are 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.)*

## 6.2.2 Contexts

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

 $41 \\ 42$ 

43

44

45

46

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

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

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

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

#### 6.2.3 Intra-Communicators

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

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

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

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

#### 6.2.4 Predefined Intra-Communicators

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

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

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

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

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11 12

13 14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28 29

30 31

32 33

34

35

MPI\_COMM\_GROUP (see below). MPI does not specify the correspondence between the
 process rank in MPI\_COMM\_WORLD and its (machine-dependent) absolute address. Neither
 does MPI specify the function of the host process, if any. Other implementation-dependent,
 predefined communicators may also be provided.

5 6 7

8

9

10 11

## 6.3 Group Management

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

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

MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2)				
IN	group1	group1 (handle)	2 3	
IN	n	number of ranks in ${\sf ranks1}$ and ${\sf ranks2}$ arrays (integer)	4	
IN	ranks1	array of zero or more valid ranks in group1	5	
IN	group2	group2 (handle)	6	
OUT	ranks2	array of corresponding ranks in group2,	7 8	
		MPI_UNDEFINED when no correspondence exists.	9	
			10	
int MPI_	-	_Group group1, int n, const int ranks1[],	11	
	MPI_Group group2, in	it ranks2[])	12 13	
		n, ranks1, group2, ranks2, ierror)	14	
	(MPI_Group), INTENT(IN) : GER, INTENT(IN) :: n, ra		15	
	GER, INTENT(OUT) :: rank		16	
	GER, OPTIONAL, INTENT(OUT		17 18	
MPI_GROU	P_TRANSLATE_RANKS(GROUP1,	N, RANKS1, GROUP2, RANKS2, IERROR)	19	
		GROUP2, RANKS2(*), IERROR	20	
This	function is important for deter	mining the relative numbering of the same processes	21	
	-	one knows the ranks of certain processes in the group	22 23	
of MPI_COMM_WORLD, one might want to know their ranks in a subset of that group.				
	MPI_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL as the translated rank.			
	T_TROC_NOLL as the translat	icu tank.	26	
MPL GRO	UP_COMPARE(group1, group2	2 result)	27 28	
IN	group1	first group (handle)	29	
	<b>C</b> .	01()	30	
IN	group2	second group (handle)	31 32	
OUT	result	result (integer)	33	
int MPT	Group compare(MPI Group g	roup1.MPI Group group2. int *result)	34	
	<pre>int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)</pre>			
	p_compare(group1, group2, (MPI_Group), INTENT(IN) :		36 37	
	GER, INTENT(OUT) :: resu		38	
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	39	
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)			40	
INTEGER GROUP1, GROUP2, RESULT, IERROR			41 42	
MPI_IDENT results if the group members and group order is exactly the same in both groups.				
		group2 are the same handle. MPI_SIMILAR results if	44	
the group	members are the same but the	e order is different. MPI_UNEQUAL results otherwise.	45	
			46 47	

```
6.3.2 Group Constructors
```

1

 $\mathbf{2}$ 

11

12

13

14

15

16 17

18

19

20

21 22 23

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

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

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

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

```
25
       IN
                 comm
                                            communicator (handle)
26
       OUT
                 group
                                            group corresponding to comm (handle)
27
28
     int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
29
30
     MPI_Comm_group(comm, group, ierror)
^{31}
          TYPE(MPI_Comm), INTENT(IN) :: comm
32
          TYPE(MPI_Group), INTENT(OUT) :: group
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_COMM_GROUP(COMM, GROUP, IERROR)
35
          INTEGER COMM, GROUP, IERROR
36
37
          MPI_COMM_GROUP returns in group a handle to the group of comm.
38
39
40
     MPI_GROUP_UNION(group1, group2, newgroup)
41
       IN
                                            first group (handle)
                 group1
42
       IN
                 group2
                                            second group (handle)
43
44
       OUT
                 newgroup
                                            union group (handle)
45
46
     int MPI_Group_union(MPI_Group group1, MPI_Group group2,
47
                    MPI_Group *newgroup)
48
```

<pre>MPI_Group_union(group1, group2, newgroup, ierror)</pre>				
TYPE(MPI_Group), INTENT(IN) :: group1, group2 TYPE(MPI_Group), INTENT(OUT) :: newgroup				
	GER, OPTIONAL, INTENT(U)	<b>.</b>	3 4	
	JER, UFILUNAL, INIENI(		5	
	MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)			
INTEC	GER GROUP1, GROUP2, NE	WGROUP, IERROR	7	
			8	
			9	
MPI_GRO	UP_INTERSECTION(group	رار), group2, newgroup)	10	
IN	group1	first group (handle)	11	
IN	group2	second group (handle)	12 13	
OUT	newgroup	intersection group (handle)	14	
			15	
int MPI_C	Group_intersection(MPI)	_Group group1, MPI_Group group2,	16	
	MPI_Group *newgro		17	
MDT Crour	intorgostion(group1	group2, newgroup, ierror)	18	
-	(MPI_Group), INTENT(IN		19	
	(MPI_Group), INTENT(OU)		20	
	GER, OPTIONAL, INTENT(	<b>.</b>	21	
MDT ODOUU			22 23	
	GER GROUP1, GROUP2, NE	GROUP2, NEWGROUP, IERROR)	23	
	ER GROOFI, GROOFZ, NE	wondor, IEnnon	25	
			26	
MPL GRO	UP_DIFFERENCE(group1,	group? newgroup)	27	
	<b>,</b>		28	
IN	group1	first group (handle)	29	
IN	group2	second group (handle)	30	
OUT	newgroup	difference group (handle)	31	
			32 33	
int MPI_C	Group_difference(MPI_G	roup group1, MPI_Group group2,	34	
	MPI_Group *newgro	up)	35	
MPI_Group	o_difference(group1, g	roup2, newgroup, ierror)	36	
-	(MPI_Group), INTENT(IN		37	
TYPE	(MPI_Group), INTENT(OU	T) :: newgroup	38	
INTEC	GER, OPTIONAL, INTENT(	OUT) :: ierror	39	
MPI_GROUP	P_DIFFERENCE(GROUP1, G	ROUP2, NEWGROUP, IERROR)	40	
	GER GROUP1, GROUP2, NE		41 42	
The set-like operations are defined as follows:				
	_	up (group1), followed by all elements of second group	45	
(grot	(group2) not in the first group. 46			
intersect all elements of the first group that are also in the second group, ordered as in the first group.				

## Unofficial Draft for Comment Only

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

<pre>int MPI_Group_excl(MPI_Group group, int n, const int ranks[],</pre>			
TYPE INTEC TYPE	<pre>MPI_Group_excl(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</pre>		
MPI_GROUP	P_EXCL(GROUP, N, RANKS, N) GER GROUP, N, RANKS(*), N)	EWGROUP, IERROR)	8 9 10
by deleting processes i must be a	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 must be a valid rank in group and all elements must be distinct; otherwise, the program is erroneous. If $n = 0$ , then newgroup is identical to group.		
MPL GRO	UP_RANGE_INCL(group, n, ra	nges newgroup)	17 18
IN	group	group (handle)	19
IN	n	number of triplets in array ranges (integer)	20 21
			22
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group	23
		of processes to be included in <b>newgroup</b>	24
OUT	newgroup	new group derived from above, in the order defined by ranges (handle)	25 26 27
int MPI_(		p group, int n, int ranges[][3],	28 29
	MPI_Group *newgroup)		30 31
-	p_range_incl(group, n, range_incl(group, n, range)		32
	(MPI_Group), INTENT(IN) : BER, INTENT(IN) :: n, ran		33
	(MPI_Group), INTENT(OUT)		34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			35
			36
	P_RANGE_INCL(GROUP, N, RAN GER GROUP, N, RANGES(3,*)		37 38
		, NEWGROOF, TERROR	39
If ranges consists of the triplets			40
$(first_1, last_1, stride_1), \ldots, (first_n, last_n, stride_n)$			41
			42
then newgroup consists of the sequence of processes in group with ranks			43
$first_1, first_1 + stride_1, \dots, first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, \dots,$			44 45 46
$ last_n - first_n $			47
$first_n, first_n + stride_n, \dots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$			48

## Unofficial Draft for Comment Only

1 Each computed rank must be a valid rank in group and all computed ranks must be  $\mathbf{2}$ distinct, or else the program is erroneous. Note that we may have  $first_i > last_i$ , and  $stride_i$ 3 may be negative, but cannot be zero. 4 The functionality of this routine is specified to be equivalent to expanding the array 5of ranges to an array of the included ranks and passing the resulting array of ranks and 6 other arguments to MPI\_GROUP\_INCL. A call to MPI\_GROUP\_INCL is equivalent to a call 7to MPI\_GROUP\_RANGE\_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the 8 argument ranges. 9 10 MPI\_GROUP\_RANGE\_EXCL(group, n, ranges, newgroup) 11 12IN group (handle) group 13 IN n number of elements in array ranges (integer) 14a one-dimensional array of integer triplets of the form IN ranges 15(first rank, last rank, stride), indicating the ranks in 16group of processes to be excluded from the output 17 group newgroup. 18 19 OUT newgroup new group derived from above, preserving the order 20in group (handle) 2122int MPI\_Group\_range\_excl(MPI\_Group group, int n, int ranges[][3], 23MPI\_Group \*newgroup)  $^{24}$ MPI\_Group\_range\_excl(group, n, ranges, newgroup, ierror) 25TYPE(MPI\_Group), INTENT(IN) :: group 26INTEGER, INTENT(IN) :: n, ranges(3,n) 27TYPE(MPI\_Group), INTENT(OUT) :: newgroup 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 MPI\_GROUP\_RANGE\_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)  $^{31}$ INTEGER GROUP, N, RANGES(3,\*), NEWGROUP, IERROR 32 Each computed rank must be a valid rank in group and all computed ranks must be distinct, 33 34or else the program is erroneous. The functionality of this routine is specified to be equivalent to expanding the array of 35 ranges to an array of the excluded ranks and passing the resulting array of ranks and other 36 arguments to MPI\_GROUP\_EXCL. A call to MPI\_GROUP\_EXCL is equivalent to a call to 37 MPI\_GROUP\_RANGE\_EXCL with each rank i in ranks replaced by the triplet (i,i,1) in the 38 argument ranges. 39 40 The range operations do not explicitly enumerate ranks, and Advice to users. 41

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

42

43

44 45

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.)

### Unofficial Draft for Comment Only

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

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

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

## 6.4 Communicator Management

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

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

incation. (End of dablee to implementors.)		36	
6.4.1 Communicator Accessors			37
0.4.1 C0	minumeator Accessors		38
The follow	ving are all local operations.		39
			40
			41
MPI_COM	M_SIZE(comm, size)		42
IN	comm	communicator (handle)	43
OUT	size	number of processes in the group of comm (integer)	44
001	5120	number of processes in the group of comm (moeger)	45
int NDT Grown sine (NDT Grown some int stains)			46
int MPI_Comm_size(MPI_Comm comm, int *size)			47
MPI_Comm_size(comm, size, ierror)			48

 $\frac{24}{25}$ 

1	TYPE(MPI_Comm), INTENT(IN)	:: comm		
2	INTEGER, INTENT(OUT) :: size			
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
4	MPI_COMM_SIZE(COMM, SIZE, IERROR)			
5	INTEGER COMM, SIZE, IERROR			
6				
7		· 1 / / · · / 1 · · / 1 · · / 1		
8 9		quivalent to accessing the communicator's group with ve), computing the size using MPI_GROUP_SIZE, and		
9 10		bup via MPI_GROUP_FREE. However, this function is		
11	· · · ·	ortcut was introduced. ( <i>End of rationale.</i> )		
12	so commonly about that this sh			
13	Advice to users. This funct	tion indicates the number of processes involved in a		
14		M_WORLD, it indicates the total number of processes		
15		f processes has been changed by using the functions		
16	<b>1</b> /	that the number of processes in MPI_COMM_WORLD		
17	does not change during the life	of an MPI program.		
18		ne next call to determine the amount of concurrency		
19	* 0	or program. The following call, MPI_COMM_RANK		
20 21	_	ss that calls it in the range from $0$ size $-1$ , where size		
21	is the return value of MPI_CO	MM_SIZE.(End of advice to users.)		
23				
24				
25	MPI_COMM_RANK(comm, rank)			
26	IN comm	communicator (handle)		
27	OUT rank	rank of the calling process in group of $comm$ (integer)		
28 29				
30	int MPI_Comm_rank(MPI_Comm comm	n, int *rank)		
31	MPI_Comm_rank(comm, rank, ierro			
32	TYPE(MPI_Comm), INTENT(IN) :: comm			
33	INTEGER, INTENT(OUT) :: ra			
34	INTEGER, OPTIONAL, INTENT(	DUT) :: ierror		
35 36	MPI_COMM_RANK(COMM, RANK, IERROR)			
37	INTEGER COMM, RANK, IERROR			
38				
39	Rationale. This function is e	quivalent to accessing the communicator's group with		
40		we), computing the rank using MPI_GROUP_RANK,		
41		group via MPI_GROUP_FREE. However, this function		
42	is so commonly used that this	shortcut was introduced. (End of rationale.)		
43				
44		gives the rank of the process in the particular commu-		
45	Ŭ I ,	noted above, in conjunction with MPI_COMM_SIZE.		
46		with the master-slave model, where one process (such		
47 48	- /	play a supervisory role, and the other processes will		
-10	serve as compute nodes. In t	his framework, the two preceding calls are useful for		

## Unofficial Draft for Comment Only

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

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

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

#### 6.4.2 Communicator Constructors

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

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

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

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

#### Unofficial Draft for Comment Only

 $^{24}$ 

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

```
5
6
     MPI_COMM_DUP(comm, newcomm)
7
8
       IN
                                          communicator (handle)
                comm
9
       OUT
                newcomm
                                          copy of comm (handle)
10
11
     int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
12
13
    MPI_Comm_dup(comm, newcomm, ierror)
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
16
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                               ierror
17
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
18
         INTEGER COMM, NEWCOMM, IERROR
19
20
```

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

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

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

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

40 41 42

43

48

36

37

38

39

27

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

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

request.

```
1
int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
                                                                                       2
MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
                                                                                       3
    TYPE(MPI_Comm), INTENT(IN) ::
                                      comm
                                                                                       4
    TYPE(MPI_Info), INTENT(IN) ::
                                      info
                                                                                       5
    TYPE(MPI_Comm), INTENT(OUT) ::
                                       newcomm
                                                                                       6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
                                                                                       9
    INTEGER COMM, INFO, NEWCOMM, IERROR
                                                                                       10
    MPI_COMM_DUP_WITH_INFO behaves exactly as MPI_COMM_DUP except that the
                                                                                       11
info hints associated with the communicator comm are not duplicated in newcomm. The
                                                                                       12
hints provided by the argument info are associated with the output communicator newcomm
                                                                                       13
instead.
                                                                                       14
                                                                                       15
     Rationale. It is expected that some hints will only be valid at communicator creation
                                                                                       16
     time. However, for legacy reasons, most communicator creation calls do not provide
                                                                                       17
     an info argument. One may associate info hints with a duplicate of any communicator
                                                                                       18
     at creation time through a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.)
                                                                                       19
                                                                                       20
                                                                                      21
MPI_COMM_IDUP(comm, newcomm, request)
                                                                                      22
                                                                                       23
  IN
                                      communicator (handle)
           comm
                                                                                       24
  OUT
           newcomm
                                      copy of comm (handle)
                                                                                       25
                                                                                       26
  OUT
                                      communication request (handle)
           request
                                                                                       27
                                                                                       28
int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
                                                                                       29
MPI_Comm_idup(comm, newcomm, request, ierror)
                                                                                       30
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                       31
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
                                                                                       32
    TYPE(MPI_Request), INTENT(OUT) ::
                                          request
                                                                                       33
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                           ierror
                                                                                      34
                                                                                      35
MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
                                                                                      36
    INTEGER COMM, NEWCOMM, REQUEST, IERROR
                                                                                      37
    MPI_COMM_IDUP is a nonblocking variant of MPI_COMM_DUP. The semantics of
                                                                                      38
MPI_COMM_IDUP are as if MPI_COMM_DUP was executed at the time that
                                                                                      39
MPI_COMM_IDUP is called. For example, attributes changed after MPI_COMM_IDUP will
                                                                                       40
not be copied to the new communicator. All restrictions and assumptions for nonblock-
                                                                                      41
ing collective operations (see Section 5.12) apply to MPI_COMM_IDUP and the returned
                                                                                      42
```

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

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

#### Unofficial Draft for Comment Only

43

44

45 46

47

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

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

1

2 IN comm communicator (handle) 3 IN group, which is a subset of the group of comm (handle) group 4 5OUT newcomm new communicator (handle) 6  $\overline{7}$ int MPI\_Comm\_create(MPI\_Comm comm, MPI\_Group group, MPI\_Comm \*newcomm) 8 MPI\_Comm\_create(comm, group, newcomm, ierror) 9 TYPE(MPI\_Comm), INTENT(IN) :: comm 10 TYPE(MPI\_Group), INTENT(IN) :: group 11 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1314MPI\_COMM\_CREATE(COMM, GROUP, NEWCOMM, IERROR) 15INTEGER COMM, GROUP, NEWCOMM, IERROR 1617If comm is an intracommunicator, this function returns a new communicator newcomm with communication group defined by the group argument. No cached information 18 propagates from comm to newcomm. Each process must call MPI\_COMM\_CREATE with 19a group argument that is a subgroup of the group associated with comm; this could be 2021MPI\_GROUP\_EMPTY. The processes may specify different values for the group argument. If a process calls with a non-empty group then all processes in that group must call the 22function with the same group as argument, that is the same processes in the same order. 23Otherwise, the call is erroneous. This implies that the set of groups specified across the  $^{24}$ processes must be disjoint. If the calling process is a member of the group given as group 2526argument, then **newcomm** is a communicator with group as its associated group. In the case that a process calls with a group to which it does not belong, e.g., MPI\_GROUP\_EMPTY, 27then MPI\_COMM\_NULL is returned as newcomm. The function is collective and must be 28called by all processes in the group of comm. 2930 Rationale. The interface supports the original mechanism from MPI-1.1, which re-31quired the same group in all processes of comm. It was extended in MPI-2.2 to allow 32 the use of disjoint subgroups in order to allow implementations to eliminate unnec-33 essary communication that MPI\_COMM\_SPLIT would incur when the user already 34 knows the membership of the disjoint subgroups. (End of rationale.) 3536 The requirement that the entire group of comm participate in the call Rationale. 37 stems from the following considerations: 38 39 • It allows the implementation to layer MPI\_COMM\_CREATE on top of regular 40 collective communications. 41 • It provides additional safety, in particular in the case where partially overlapping 42groups are used to create new communicators. 43 • It permits implementations to sometimes avoid communication related to context 44 creation. 4546 (End of rationale.) 47 48

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

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

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

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

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

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

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

1 2

3

4

 $\mathbf{5}$ 

6 7

8

9

10

11

12

13

14

15 16

17

18

19

20 21

22

23

 $\frac{24}{25}$ 

26

27 28

29

30

 $^{31}$ 

32 33 34

35

36

```
1
                                INTER-COMMUNICATOR CREATE
2
                       Before
3
4
                               0
                              5
6
                         0
7
                          4
8
                                                                     2
                                    IŌ
9
10
11
                                  1
                                                                 ١
                                                               ١
                        After
12
                                   I
13
                                 1
14
                               00
15
                                                                   1
16
17
                                                                        6
                                                                  2
18
19
20
21
22
     Figure 6.1: Intercommunicator creation using MPI_COMM_CREATE extended to intercom-
23
     municators. The input groups are those in the grey circle.
^{24}
25
                 MPI_Comm_group ( inter_comm, &local_group );
26
                 MPI_Group_incl ( local_group, 1, &rank, &group );
27
                 MPI_Group_free ( &local_group );
28
              }
29
              else
30
                 MPI_Comm_group ( inter_comm, &group );
^{31}
32
              MPI_Comm_create ( inter_comm, group, &new_inter_comm );
33
              MPI_Group_free( &group );
34
35
36
37
     MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)
38
       IN
                 comm
                                             intracommunicator (handle)
39
       IN
                                             group, which is a subset of the group of comm (handle)
                 group
40
41
       IN
                                             tag (integer)
                 tag
42
       OUT
                                             new communicator (handle)
                 newcomm
43
44
     int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,
45
                     MPI_Comm *newcomm)
46
47
     MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
48
          TYPE(MPI_Comm), INTENT(IN) :: comm
```

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

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

1

			wcomm)	
2 3	IN	comm	communicator (handle)	
4	IN	color	control of subset assignment (integer)	
5	IN	key	control of rank assignment (integer)	
6	OUT	newcomm	new communicator (handle)	
7				
8 9	int MPI_C	Comm_split(MPI_Comm comm,	int color, int key, MPI_Comm *newcomm)	
10		split(comm, color, key, r		
11 12		(MPI_Comm), INTENT(IN) ::		
12		GER, INTENT(IN) :: color, (MPI_Comm), INTENT(OUT) ::	0	
14		GER, OPTIONAL, INTENT(OUT)		
15				
16		SPLIT(COMM, COLOR, KEY, N		
17	INTEC	ER COMM, COLOR, KEY, NEWC	JUMM, IERROR	
18			ciated with comm into disjoint subgroups, one for	
19 20		0 1	tains all processes of the same color. Within each	
20		-	the order defined by the value of the argument rank in the old group. A new communicator is	
22	• /	0	n newcomm. A process may supply the color value	
23			nm returns MPI_COMM_NULL. This is a collective	
24	call, but each process is permitted to provide different values for color and key.			
25	With an intracommunicator comm, a call to $MPI_COMM_CREATE(comm, group, new)$			
26 27	comm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where			
27	processes that are members of their group argument provide $color =$ number of the group (based on a unique numbering of all disjoint groups) and key = rank in group, and all			
29	•		roup argument provide color $=$ MPI_UNDEFINED.	
30	-	alue of color must be non-nega		
31				
32			remely powerful mechanism for dividing a single	
33 34		00	into $k$ subgroups, with $k$ chosen implicitly by the erted over all the processes). Each resulting com-	
35			g. Such a division could be useful for defining a	
36			for multigrid, or linear algebra. For intracommu-	
37		· · · · · ·	ides similar capability as MPI_COMM_CREATE to	
38			disjoint subgroups. MPI_COMM_SPLIT is useful	
39		_	omplete information of the other members in their	
40			color of) the group to which they belong. In this overs the other group members via communication.	
41 42		=	the all processes have complete information of the	
42			e, MPI can avoid the extra communication required	
44			I_COMM_CREATE_GROUP is useful when all pro-	
45			ete information of the members of their group and	
46	sync	hronization with processes out	side the group can be avoided.	
47	Mult	iple calls to MPI_COMM_SPL	IT can be used to overcome the requirement that	

<sup>47</sup> Multiple calls to MPI\_COMM\_SPLIT can be used to overcome the requirement that <sup>48</sup> any call have no overlap of the resulting communicators (each process is of only one

## Unofficial Draft for Comment Only

color per call). In this way, multiple overlapping communication structures can be created. Creative use of the color and key in such splitting operations is encouraged.

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

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

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

The result of MPI\_COMM\_SPLIT on an intercommunicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see Figure 6.2). For those colors that are specified only on one side of the intercommunicator, MPI\_COMM\_NULL is returned. MPI\_COMM\_NULL is also returned to those processes that specify MPI\_UNDEFINED as the color.

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

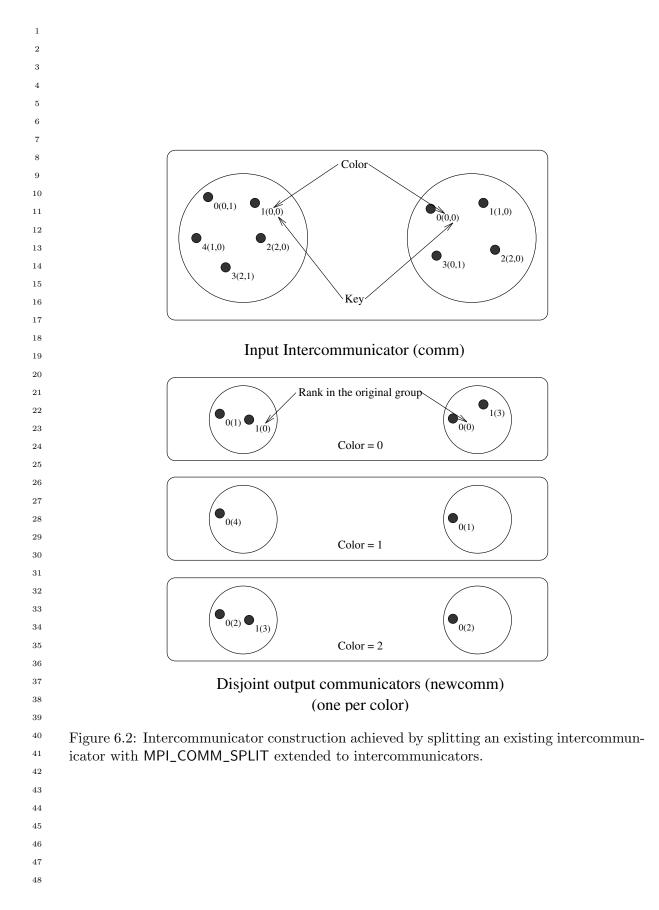
**Example 6.2** (Parallel client-server model). The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

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

Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 



The following is the corresponding server code:

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

This function partitions the group associated with comm into disjoint subgroups, based on the type specified by split\_type. Each subgroup contains all processes of the same type. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. This is a collective call; all processes must provide the same split\_type, but each process is permitted to provide different values for key. An exception to this rule is that a process may supply the type value MPI\_UNDEFINED, in which case newcomm returns MPI\_COMM\_NULL.

The following type is predefined by MPI:

#### Unofficial Draft for Comment Only

1

39

40

41

42

43

44

45

46

47

1 MPI\_COMM\_TYPE\_SHARED — this type splits the communicator into subcommunicators,  $\mathbf{2}$ each of which can create a shared memory region. 3 Advice to implementors. Implementations can define their own types, or use the 4 info argument, to assist in creating communicators that help expose platform-specific 5information to the application. (End of advice to implementors.) 6 7 8 Communicator Destructors 6.4.3 9 10 11MPI\_COMM\_FREE(comm) 12INOUT communicator to be destroyed (handle) comm 13 14int MPI\_Comm\_free(MPI\_Comm \*comm) 1516MPI\_Comm\_free(comm, ierror) 17TYPE(MPI\_Comm), INTENT(INOUT) :: comm 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19MPI\_COMM\_FREE(COMM, IERROR) 20INTEGER COMM, IERROR 2122This collective operation marks the communication object for deallocation. The handle 23is set to MPI\_COMM\_NULL. Any pending operations that use this communicator will com-24plete normally; the object is actually deallocated only if there are no other active references 25to it. This call applies to intra- and inter-communicators. The delete callback functions for 26all cached attributes (see Section 6.7) are called in arbitrary order. 2728Advice to implementors. A reference-count mechanism may be used: the reference 29 count is incremented by each call to MPI\_COMM\_DUP or MPI\_COMM\_IDUP, and 30 decremented by each call to MPI\_COMM\_FREE. The object is ultimately deallocated 31when the count reaches zero. 32 Though collective, it is anticipated that this operation will normally be implemented 33 to be local, though a debugging version of an MPI library might choose to synchronize. 34 (End of advice to implementors.) 35 36 6.4.4 Communicator Info 37 38

<sup>30</sup> Hints specified via info (see Chapter 9) allow a user to provide information to direct opti <sup>31</sup> mization. Providing hints may enable an implementation to deliver increased performance
 <sup>41</sup> or minimize use of system resources. However, hints do not change the semantics of any MPI
 <sup>42</sup> interfaces. In other words, an implementation is free to ignore all hints. Hints are specified
 <sup>43</sup> on a per communicator basis, in MPI\_COMM\_DUP\_WITH\_INFO, MPI\_COMM\_SET\_INFO,
 <sup>44</sup> MPI\_COMM\_SPLIT\_TYPE, MPI\_DIST\_GRAPH\_CREATE\_ADJACENT, and
 <sup>44</sup> MPI\_DIST\_CRAPH\_CREATE\_via the opeque info object\_When an info object that speci

<sup>44</sup> MPI\_DIST\_GRAPH\_CREATE, via the opaque info object. When an info object that speci-<sup>45</sup> fies a subset of valid hints is passed to MPI\_COMM\_SET\_INFO, there will be no effect on <sup>46</sup> previously set or defaulted hints that the info does not specify.

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

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

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

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

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

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

INTEGER COMM, INFO, IERROR

MPI\_COMM\_SET\_INFO sets new values for the hints of the communicator associated with comm. MPI\_COMM\_SET\_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

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

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

Unofficial Draft for Comment Only

 $^{24}$ 

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

INTEGER COMM, INFO\_USED, IERROR

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

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

156.5 16

3

4

5

6

7

8 9

10

11

12

13

14

## Motivating Examples

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

Example #1a is a do-nothing program that initializes itself, and refers to the "all" commu-35 nicator, and prints a message. It terminates itself too. This example does not imply that 36 MPI supports printf-like communication itself. 37

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

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

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

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

## 6.5.2 Current Practice #2

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

This example illustrates the use of a collective communication.

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

Unofficial Draft for Comment Only

 $\frac{24}{25}$ 

 $^{31}$ 

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

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

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

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

```
1
           if(comm_b != MPI_COMM_NULL)
\mathbf{2}
           {
3
             lib_call(comm_b);
4
             lib_call(comm_b);
5
           }
6
7
           if(comm_a != MPI_COMM_NULL)
8
             MPI_Comm_free(&comm_a);
9
           if(comm_b != MPI_COMM_NULL)
10
             MPI_Comm_free(&comm_b);
11
           MPI_Group_free(&group_a);
12
           MPI_Group_free(&group_b);
13
           MPI_Group_free(&group_world);
14
           MPI_Finalize();
15
           return 0;
16
         }
17
     The library:
18
19
         void lib_call(MPI_Comm comm)
20
         ſ
21
           int me, done = 0;
22
           MPI_Status status;
23
           MPI_Comm_rank(comm, &me);
24
           if(me == 0)
25
              while(!done)
26
              {
27
                  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
28
                  . . .
29
              }
30
           else
31
           ſ
32
             /* work */
33
             MPI_Send(..., 0, ARBITRARY_TAG, comm);
34
              . . . .
35
           }
36
     #ifdef EXAMPLE_2C
37
           /* include (resp, exclude) for safety (resp, no safety): */
38
           MPI_Barrier(comm);
39
     #endif
40
         }
41
42
     The above example is really three examples, depending on whether or not one includes rank
```

The above example is really three examples, depending on whether or not one includes rank 3 in list\_b, and whether or not a synchronize is included in lib\_call. This example illustrates that, despite contexts, subsequent calls to lib\_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI\_Barrier is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back-masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no back-masking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [57]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity — deleting either feature removes the guarantee that back-masking cannot be required.

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

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

# 6.6 Inter-Communication

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All communication described thus far has involved communication between processes that are members of the same group. This type of communication is called "intra-communication" and the communicator used is called an "intra-communicator," as we have noted earlier in the chapter.

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

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the "local group," that is, the sender in a send and the receiver in a receive. The group containing the target process is called the "remote group," that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking except for MPI\_COMM\_IDUP and require that the local and remote groups be disjoint.

Advice to users. The groups must be disjoint for several reasons. Primarily, this is the intent of the intercommunicators — to provide a communicator for communication between disjoint groups. This is reflected in the definition of

1 2

3

4

 $\mathbf{5}$ 

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

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

1 2 3 4 5	MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the pro- cesses in the created intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to inter- communicators makes the most sense when the groups are disjoint. ( <i>End of advice to</i> <i>users</i> .)
6 7	Here is a summary of the properties of inter-communication and inter-communicators:
8 9 10 11	• The syntax of point-to-point and collective communication is the same for both inter- and intra-communication. The same communicator can be used both for send and for receive operations.
12 13 14	• A target process is addressed by its rank in the remote group, both for sends and for receives.
14 15 16	• Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
17 18	• A communicator will provide either intra- or inter-communication, never both.
19 20 21 22 23	The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_CART_CREATE).
24 25	Advice to implementors. For the purpose of point-to-point communication, commu- nicators can be represented in each process by a tuple consisting of:
26 27	group
28	send_context
29 30	receive_context
31	source
32 33 34 35 36	For inter-communicators, group describes the remote group, and source is the rank of the process in the local group. For intra-communicators, group is the communicator group (remote=local), source is the rank of the process in this group, and send context and receive context are identical. A group can be represented by a rank-to-absolute-address translation table.
37 38 39 40 41	The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process $\mathbf{P}$ in group $\mathcal{P}$ , which has an inter-communicator $\mathbf{C}_{\mathcal{P}}$ , and a process $\mathbf{Q}$ in group $\mathcal{Q}$ , which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$ . Then
42	• $\mathbf{C}_{\mathcal{P}}$ .group describes the group $\mathcal{Q}$ and $\mathbf{C}_{\mathcal{Q}}$ .group describes the group $\mathcal{P}$ .
43 44 45	• $C_{\mathcal{P}}$ .send_context = $C_{\mathcal{Q}}$ .receive_context and the context is unique in $\mathcal{Q}$ ; $C_{\mathcal{P}}$ .receive_context = $C_{\mathcal{Q}}$ .send_context and this context is unique in $\mathcal{P}$ .
45 46 47	• $\mathbf{C}_{\mathcal{P}}$ .source is rank of $\mathbf{P}$ in $\mathcal{P}$ and $\mathbf{C}_{\mathcal{Q}}$ .source is rank of $\mathbf{Q}$ in $\mathcal{Q}$ .
48	

Assume that  $\mathbf{P}$  sends a message to  $\mathbf{Q}$  using the inter-communicator. Then  $\mathbf{P}$  uses the **group** table to find the absolute address of  $\mathbf{Q}$ ; **source** and **send\_context** are appended to the message.

Assume that  $\mathbf{Q}$  posts a receive with an explicit source argument using the intercommunicator. Then  $\mathbf{Q}$  matches **receive\_context** to the message context and source argument to the message source.

The same algorithm is appropriate for intra-communicators as well.

In order to support inter-communicator accessors and constructors, it is necessary to supplement this model with additional structures, that store information about the local communication group, and additional safe contexts. (*End of advice to implementors.*)

#### 6.6.1 Inter-communicator Accessors

MPI\_COMM\_TEST\_INTER(comm, flag)

IN	comm	communicator (handle)
OUT	flag	(logical)

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

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

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

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

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

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

Furthermore, the operation MPI\_COMM\_COMPARE is valid for inter-communicators. Both communicators must be either intra- or inter-communicators, or else MPI\_UNEQUAL results. Both corresponding local and remote groups must compare correctly to get the results 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. 44 45 46 47 48

#### Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12 13 14

15 16 17

18 19 20

21 22

23

 $^{24}$ 

25

26

27 28

29

30

31

32

33

34

35

The following accessors provide consistent access to the remote group of an intercommunicator. The following are all local operations.

1

 $\mathbf{2}$ 

3

48

```
4
     MPI_COMM_REMOTE_SIZE(comm, size)
5
6
       IN
                 comm
                                            inter-communicator (handle)
7
       OUT
                                            number of processes in the remote group of comm
                size
8
                                            (integer)
9
10
     int MPI_Comm_remote_size(MPI_Comm comm, int *size)
11
12
     MPI_Comm_remote_size(comm, size, ierror)
13
          TYPE(MPI_Comm), INTENT(IN) ::
                                            comm
14
          INTEGER, INTENT(OUT) :: size
15
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
17
          INTEGER COMM, SIZE, IERROR
18
19
20
21
     MPI_COMM_REMOTE_GROUP(comm, group)
22
       IN
                 comm
                                            inter-communicator (handle)
23
       OUT
                                            remote group corresponding to comm (handle)
                 group
^{24}
25
     int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
26
27
     MPI_Comm_remote_group(comm, group, ierror)
28
          TYPE(MPI_Comm), INTENT(IN) :: comm
29
          TYPE(MPI_Group), INTENT(OUT) :: group
30
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
^{31}
     MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
32
          INTEGER COMM, GROUP, IERROR
33
34
35
           Rationale.
                        Symmetric access to both the local and remote groups of an inter-
36
          communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE
37
          have been provided. (End of rationale.)
38
39
            Inter-communicator Operations
     6.6.2
40
41
     This section introduces four blocking inter-communicator operations.
42
     MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-com-
     municator; the function MPI_INTERCOMM_MERGE creates an intra-communicator by merg-
43
     ing the local and remote groups of an inter-communicator. The functions MPI_COMM_DUP
44
     and MPI_COMM_FREE, introduced previously, duplicate and free an inter-communicator,
45
46
     respectively.
47
          Overlap of local and remote groups that are bound into an inter-communicator is
     prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If
```

a process is multithreaded, and MPI calls block only a thread, rather than a process, then "dual membership" can be supported. It is then the user's responsibility to make sure that calls on behalf of the two "roles" of a process are executed by two independent threads.) 4

The function MPI\_INTERCOMM\_CREATE can be used to create an inter-communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator. Furthermore, members of each group know the rank of their leader.

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

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

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

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

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

INlocal_commlocal intra-communicator (handle)27INlocal_leaderrank of local group leader in local_comm (integer)28INpeer_comm"peer" communicator; significant only at the local_leader (handle)30INremote_leaderrank of remote group leader in peer_comm; significant only at the local_leader (integer)32INtagtag (integer)33INtagtag (integer)36OUTnewintercommnew inter-communicator (handle)36INtagtag (integer)36OUTnewintercomm, int remote_leader, int tag, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)36MPI_Intercomm_create(local_com, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror)41MPI_Comm), INTENT(IN) ::local_leader, remote_leader, tag TYPE(MPI_Comm), INTENT(OUT) ::43INTEGER, OPTIONAL, INTENT(OUT) ::ierror46INTEGER, OPTIONAL, INTENT(OUT) ::ierror47		,		
IN       local_leader       rank of local group leader in local_comm (integer)       29         IN       peer_comm       "peer" communicator; significant only at the local_leader (handle)       30         IN       remote_leader       rank of remote group leader in peer_comm; significant only at the local_leader (integer)       33         IN       tag       tag (integer)       34         IN       tag       tag (integer)       35         OUT       newintercomm       new inter-communicator (handle)       36         int       MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, int tag, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)       37         MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror)       41         MPI_Comm), INTENT(IN) :: local_leader, remote_leader, tag       41         INTEGER, INTENT(IN) :: local_leader, remote_leader, tag       43         TYPE(MPI_Comm), INTENT(OUT) :: newintercomm       44         INTEGER, OPTIONAL, INTENT(OUT) :: newintercomm       45	IN	local_comm	local intra-communicator (handle)	
<pre>IN peer_comm "peer" communicator; significant only at the local_leader (handle) 31 IN remote_leader rank of remote group leader in peer_comm; significant only at the local_leader (integer) 33 IN tag tag (integer) 34 IN tag tag (integer) 35 OUT newintercomm new inter-communicator (handle) 36 IN MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, 38 MPI_Comm peer_comm, int remote_leader, int tag, 39 MPI_Comm *newintercomm) 40 MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, 42 tag, newintercomm, ierror) 40 MPI_Comm ), INTENT(IN) :: local_leader, peer_comm 44 INTEGER, INTENT(IN) :: local_leader, remote_leader, tag 7YPE(MPI_Comm), INTENT(OUT) :: newintercomm 45 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46 </pre>	IN	local_leader	rank of local group leader in local_comm (integer)	
IN       Trank of Tenote group reader in peer_comm, significant only at the local_leader (integer)       33         IN       tag       tag (integer)       34         OUT       newintercomm       new inter-communicator (handle)       36         int       MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)       37         MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror)       40         MPI_Comm), INTENT(IN) ::       local_leader, peer_comm       41         INTEGER, INTENT(IN) ::       local_leader, remote_leader, tag       42         TYPE(MPI_Comm), INTENT(OUT) ::       newintercomm       44         INTEGER, OPTIONAL, INTENT(OUT) ::       newintercomm       46	IN	peer_comm		30
<pre>IN tag tag (integer) 35 OUT newintercomm new inter-communicator (handle) 36 int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, 38</pre>	IN	remote_leader		
OUT       newintercomm       new inter-communicator (handle)       36         int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,       37         MPI_Comm peer_comm, int remote_leader, int tag,       39         MPI_Comm *newintercomm)       40         MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,       41         tag, newintercomm, ierror)       43         TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm       44         INTEGER, INTENT(IN) :: local_leader, remote_leader, tag       45         TYPE(MPI_Comm), INTENT(OUT) :: newintercomm       46	IN	tag	tag (integer)	
<pre>int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,</pre>	OUT	newintercomm	new inter-communicator (handle)	
<pre>MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm) 40 MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, 42 tag, newintercomm, ierror) 43 TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm 44 INTEGER, INTENT(IN) :: local_leader, remote_leader, tag 45 TYPE(MPI_Comm), INTENT(OUT) :: newintercomm 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				37
MPI_Comm *newintercomm)       40         MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror)       41         TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm       43         INTEGER, INTENT(IN) :: local_leader, remote_leader, tag       44         TYPE(MPI_Comm), INTENT(OUT) :: newintercomm       45         INTEGER, OPTIONAL, INTENT(OUT) :: ierror       46	int MPI_	Intercomm_create(MPI_Co	omm local_comm, int local_leader,	38
MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror)       41         TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm       43         INTEGER, INTENT(IN) :: local_leader, remote_leader, tag       44         TYPE(MPI_Comm), INTENT(OUT) :: newintercomm       45         INTEGER, OPTIONAL, INTENT(OUT) :: ierror       46		MPI_Comm peer_com	m, int remote_leader, int tag,	39
<pre>MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror) TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm INTEGER, INTENT(IN) :: local_leader, remote_leader, tag TYPE(MPI_Comm), INTENT(OUT) :: newintercomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>		MPI_Comm *newinte:	rcomm)	40
tag, newintercomm, ierror)42TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm43INTEGER, INTENT(IN) :: local_leader, remote_leader, tag44TYPE(MPI_Comm), INTENT(OUT) :: newintercomm45INTEGER, OPTIONAL, INTENT(OUT) :: ierror46	MPT Inte	ercomm create(local comm	n. local leader, peer comm, remote leader.	
TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm43INTEGER, INTENT(IN) :: local_leader, remote_leader, tag44TYPE(MPI_Comm), INTENT(OUT) :: newintercomm45INTEGER, OPTIONAL, INTENT(OUT) :: ierror46			-	
INTEGER, INTENT(IN) :: local_leader, remote_leader, tag TYPE(MPI_Comm), INTENT(OUT) :: newintercomm INTEGER, OPTIONAL, INTENT(OUT) :: ierror	TYPE	0.	-	
TYPE(MPI_Comm), INTENT(OUT) :: newintercomm INTEGER. OPTIONAL. INTENT(OUT) :: ierror 45 46			-	
INTEGER. OPTIONAL. INTENT(OUT) :: ierror			0	
47	INTE	GER, OPTIONAL, INTENT(C	DUT) :: ierror	
				47

**Unofficial Draft for Comment Only** 

1  $\mathbf{2}$ 

3

 $\mathbf{5}$ 

6  $\overline{7}$ 

8 9

10 11

12

13

1415

16

17

18

19 20

21

2223 $^{24}$ 

25

26

1 MPI\_INTERCOMM\_CREATE(LOCAL\_COMM, LOCAL\_LEADER, PEER\_COMM, REMOTE\_LEADER,  $\mathbf{2}$ TAG, NEWINTERCOMM, IERROR) 3 INTEGER LOCAL\_COMM, LOCAL\_LEADER, PEER\_COMM, REMOTE\_LEADER, TAG, 4 NEWINTERCOMM, IERROR 5This call creates an inter-communicator. It is collective over the union of the local and 6 remote groups. Processes should provide identical local\_comm and local\_leader arguments 7 within each group. Wildcards are not permitted for remote\_leader, local\_leader, and tag. 8 9 10 MPI\_INTERCOMM\_MERGE(intercomm, high, newintracomm) 11IN intercomm Inter-Communicator (handle) 12IN high (logical) 13 14OUT newintracomm new intra-communicator (handle) 1516int MPI\_Intercomm\_merge(MPI\_Comm intercomm, int high, 17MPI\_Comm \*newintracomm) 18 19MPI\_Intercomm\_merge(intercomm, high, newintracomm, ierror) 20TYPE(MPI\_Comm), INTENT(IN) :: intercomm 21LOGICAL, INTENT(IN) :: high 22TYPE(MPI\_Comm), INTENT(OUT) :: newintracomm 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24MPI\_INTERCOMM\_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) 25INTEGER INTERCOMM, NEWINTRACOMM, IERROR 26LOGICAL HIGH 2728This function creates an intra-communicator from the union of the two groups that are 29associated with intercomm. All processes should provide the same high value within each 30 of the two groups. If processes in one group provided the value high = false and processes  $^{31}$ in the other group provided the value high = true then the union orders the "low" group 32 before the "high" group. If all processes provided the same high argument then the order 33 of the union is arbitrary. This call is blocking and collective within the union of the two 34 groups. 35 The error handler on the new intercommunicator in each process is inherited from 36 the communicator that contributes the local group. Note that this can result in different 37 processes in the same communicator having different error handlers. 38 Advice to implementors. The implementation of MPI\_INTERCOMM\_MERGE, 39 MPI\_COMM\_FREE, and MPI\_COMM\_DUP are similar to the implementation of 40 41 MPI\_INTERCOMM\_CREATE, except that contexts private to the input inter-com-42municator are used for communication between group leaders rather than contexts inside a bridge communicator. (End of advice to implementors.) 43 44454647 48

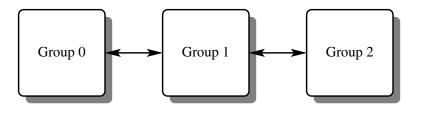


Figure 6.3: Three-group pipeline

# 6.6.3 Inter-Communication Examples

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

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

```
16
int main(int argc, char *argv[])
                                                                                 17
{
                                                                                 18
 MPI_Comm
             myComm;
                            /* intra-communicator of local sub-group */
                                                                                 19
 MPI_Comm
             myFirstComm; /* inter-communicator */
                                                                                 20
 MPI_Comm
             mySecondComm; /* second inter-communicator (group 1 only) */
                                                                                 21
  int membershipKey;
                                                                                 22
  int rank;
                                                                                 23
                                                                                 24
  MPI_Init(&argc, &argv);
                                                                                 25
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                                                                                 26
                                                                                 27
  /* User code must generate membershipKey in the range [0, 1, 2] */
                                                                                 28
  membershipKey = rank % 3;
                                                                                 29
                                                                                 30
  /* Build intra-communicator for local sub-group */
                                                                                 31
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
                                                                                 32
                                                                                 33
  /* Build inter-communicators. Tags are hard-coded. */
                                                                                 34
  if (membershipKey == 0)
                                                                                 35
  {
                         /* Group 0 communicates with group 1. */
                                                                                 36
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                                                                                 37
                          1, &myFirstComm);
                                                                                 38
  }
                                                                                 39
  else if (membershipKey == 1)
                                                                                 40
  {
                  /* Group 1 communicates with groups 0 and 2. */
                                                                                 41
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                                                                                 42
                          1, &myFirstComm);
                                                                                 43
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                                                                                 44
                          12, &mySecondComm);
                                                                                 45
  }
                                                                                 46
  else if (membershipKey == 2)
                                                                                 47
                         /* Group 2 communicates with group 1. */
  ſ
                                                                                 48
```

1 2

9

10 11

12

13

14

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

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

#### 6.7 Caching

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects, communicators, windows, and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window, or datatype,
- quickly retrieve that information, and

# **Unofficial Draft for Comment Only**

1 2

3

4

5 6

7

9

10

11

12

13

15

16

17

18

19

24

26

31

32

33

38

39 40

41

42

43

44

4546

47

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

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

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

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

One difficulty is the potential for size differences between Fortran integers and C pointers. For this reason, the Fortran versions of these routines use integers of kind MPI\_ADDRESS\_KIND.

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI\_XXX\_CREATE\_KEYVAL is used with an object of the wrong type with a call to MPI\_YYY\_GET\_ATTR, MPI\_YYY\_SET\_ATTR, MPI\_YYY\_DELETE\_ATTR, or MPI\_YYY\_FREE\_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

 $^{24}$ 

# 6.7.1 Functionality

Attributes can be attached to communicators, windows, and datatypes. Attributes are local to the process and specific to the communicator to which they are attached. Attributes are not propagated by MPI from one communicator to another except when the communicator is duplicated using MPI\_COMM\_DUP or MPI\_COMM\_IDUP (and even then the application must give specific permission through callback functions for the attribute to be copied).

Advice to users. Attributes in C are of type void \*. Typically, such an attribute will be a pointer to a structure that contains further information, or a handle to an MPI object. In Fortran, attributes are of type INTEGER. Such attribute can be a handle to an MPI object, or just an integer-valued attribute. (*End of advice to users.*)

Advice to implementors. Attributes are scalar values, equal in size to, or larger than a C-language pointer. Attributes can always hold an MPI handle. (*End of advice to implementors.*)

The caching interface defined here requires that attributes be stored by MPI opaquely within a communicator, window, and datatype. Accessor functions include the following:

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

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

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

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

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

# 6.7.2 Communicators

Functions for caching on communicators are:

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

IN	comm_copy_attr_fn	copy callback function for $comm\_keyval$ (function)	32
IN	comm_delete_attr_fn	delete callback function for $comm\_keyval$ (function)	33
OUT	comm_keyval	key value for future access (integer)	34 35
IN	extra_state	extra state for callback functions	36

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

Unofficial Draft for Comment Only

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

1 The comm\_copy\_attr\_fn function is invoked when a communicator is duplicated by  $\mathbf{2}$ MPI\_COMM\_DUP or MPI\_COMM\_IDUP. comm\_copy\_attr\_fn should be of type 3 MPI\_Comm\_copy\_attr\_function. The copy callback function is invoked for each key value in 4 oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its  $\mathbf{5}$ corresponding attribute. If it returns flag = 0 or .FALSE, then the attribute is deleted in the duplicated communicator. Otherwise (flag = 1 or .TRUE.), the new attribute value is set to 6  $\overline{7}$ the value returned in attribute\_val\_out. The function returns MPI\_SUCCESS on success and an error code on failure (in which case MPI\_COMM\_DUP or MPI\_COMM\_IDUP will fail). 8

The argument comm\_copy\_attr\_fn may be specified as MPI\_COMM\_NULL\_COPY\_FN or MPI\_COMM\_DUP\_FN from either C or Fortran. MPI\_COMM\_NULL\_COPY\_FN is a function that does nothing other than returning flag = 0 or .FALSE. (depending on whether 12the keyval was created with a C or Fortran binding to MPI\_COMM\_CREATE\_KEYVAL) and MPI\_SUCCESS. MPI\_COMM\_DUP\_FN is a simple-minded copy function that sets flag = 1 or .TRUE., returns the value of attribute\_val\_in in attribute\_val\_out, and returns MPI\_SUCCESS. These replace the MPI-1 predefined callbacks MPI\_NULL\_COPY\_FN and MPI\_DUP\_FN, whose use is deprecated.

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

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

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

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

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

41 The argument comm\_delete\_attr\_fn may be specified as 42MPI\_COMM\_NULL\_DELETE\_FN from either C or Fortran. MPI\_COMM\_NULL\_DELETE\_FN is a function that does nothing, other than returning 43 MPI\_SUCCESS. MPI\_COMM\_NULL\_DELETE\_FN replaces MPI\_NULL\_DELETE\_FN, whose 44use is deprecated. 4546

If an attribute copy function or attribute delete function returns other than MPI\_SUCCESS, then the call that caused it to be invoked (for example, MPI\_COMM\_FREE), is erroneous.

9

10 11

13

1415

16

17

18

19

20

21

2223

 $^{24}$ 

25

26

27

2829

30

 $^{31}$ 

32 33

34

35

36

37

38

39

40

1 2 3			INVALID is never returned by ore, it can be used for static initialization of key
4 5 7 8 9 10 11 12 13	Advia MPI_ MPI_ the m can o impli routi name	pi_f08 module with the same coexist twice with the same r acit interface outside of the mpi ne within mpi_f08 declared v	•
14 15 16 17 18 19 20	MPI_ MPI_ that	COMM_NULL_COPY_FN, M COMM_NULL_DELETE_FN s uses the mpi_f08 module to an oif.h, and vice versa; see also	Iding the predefined Fortran functions PI_COMM_DUP_FN, and should not be passed from one application routine other application routine that uses the mpi module the advice to users on page 680. ( <i>End of advice to</i>
21 22	MPI_COM	M_FREE_KEYVAL(comm_keyv	/al)
23 24	INOUT	comm_keyval	key value (integer)
25	int MPI C	comm_free_keyval(int *comm	n kevval)
26 27 28 29	MPI_Comm_ INTEG	free_keyval(comm_keyval, ER, INTENT(INOUT) :: com ER, OPTIONAL, INTENT(OUT)	ierror) m_keyval
30 31 32		FREE_KEYVAL(COMM_KEYVAL, ER COMM_KEYVAL, IERROR	IERROR)
33 34 35 36 37 38 39 40 41	MPI_KEYV/ because th on the proc program, e	AL_INVALID. Note that it is not e actual free does not transpir- cess) to the key have been freed wither via calls to MPI_COMM to MPI_COMM_FREE that fr	s function sets the value of keyval to t erroneous to free an attribute key that is in use, e until after all references (in other communicators I. These references need to be explicitly freed by the _DELETE_ATTR that free one attribute instance, ee all attribute instances associated with the freed
42	MPI_COM	M_SET_ATTR(comm, comm_l	<eyval, attribute_val)<="" td=""></eyval,>
43 44	INOUT	comm	communicator from which attribute will be attached (handle)
45 46	IN	comm_keyval	key value (integer)
47 48	IN	attribute_val	attribute value

# 6.7. CACHING

int MPI_	Comm_set_attr(MPI_Comm_co	omm, int comm_keyval, void *attribute_val)	1
MPI Comm	set attr(comm, comm kevy	val, attribute_val, ierror)	2
	(MPI_Comm), INTENT(IN) :		$\frac{3}{4}$
INTE	GER, INTENT(IN) :: comm_	_keyval	5
INTE	GER(KIND=MPI_ADDRESS_KINI	), INTENT(IN) :: attribute_val	6
INTE	GER, OPTIONAL, INTENT(OUT	[) :: ierror	7
MPI_COMM	_SET_ATTR(COMM, COMM_KEY)	/AL, ATTRIBUTE_VAL, IERROR)	8
	GER COMM, COMM_KEYVAL, II		9
INTE	GER(KIND=MPI_ADDRESS_KINI	)) ATTRIBUTE_VAL	10
This	function stores the stipulated	attribute value attribute_val for subsequent retrieval	11
	-	alue is already present, then the outcome is as if	12
		called to delete the previous value (and the callback	13 14
		cuted), and a new value was next stored. The call	14
is erroneo	us if there is no key with val	ue keyval; in particular MPI_KEYVAL_INVALID is an	16
erroneous	key value. The call will fail if	the $comm\_delete\_attr\_fn$ function returned an error	17
code other	$t $ than MPI_SUCCESS.		18
			19
MPI COM	IM_GET_ATTR(comm, comm	kevval. attribute val. flag)	20
IN		,	21
IIN	comm	communicator to which the attribute is attached (han- dle)	22
INI		,	23 24
IN	comm_keyval	key value (integer)	24 25
OUT	attribute_val	attribute value, unless $flag = false$	26
OUT	flag	false if no attribute is associated with the key (logical)	27
			28
int MPI_	-	omm, int comm_keyval, void *attribute_val,	29
	int *flag)		30
MPI_Comm	_get_attr(comm, comm_keyv	val, attribute_val, flag, ierror)	31
TYPE	(MPI_Comm), INTENT(IN) ::	comm	32
	GER, INTENT(IN) :: comm_		$33 \\ 34$
		)), INTENT(OUT) :: attribute_val	35
	CAL, INTENT(OUT) :: flag		36
INIE	GER, OPTIONAL, INTENT(OUT	l) :: lerror	37
MPI_COMM	_GET_ATTR(COMM, COMM_KEY)	/AL, ATTRIBUTE_VAL, FLAG, IERROR)	38
	GER COMM, COMM_KEYVAL, IN		39
	GER(KIND=MPI_ADDRESS_KINI	)) ATTRIBUTE_VAL	40
LOGI	CAL FLAG		41
Retri	eves attribute value by key.	The call is erroneous if there is no key with value	42
keyval. O	n the other hand, the call is	correct if the key value exists, but no attribute is	43

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

Advice to users. The call to MPI\_Comm\_set\_attr passes in attribute\_val the value of the attribute; the call to MPI\_Comm\_get\_attr passes in attribute\_val the address of the 48

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

44

45

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

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

```
MPI_COMM_DELETE_ATTR(comm, comm_keyval)
```

1

 $\mathbf{2}$ 

3

4

5

6

7

12

48

communicator from which the attribute is deleted (han-INOUT comm 13 dle) 1415IN comm\_keyval key value (integer) 1617 int MPI\_Comm\_delete\_attr(MPI\_Comm comm, int comm\_keyval) 18 19MPI\_Comm\_delete\_attr(comm, comm\_keyval, ierror) TYPE(MPI\_Comm), INTENT(IN) :: comm 20INTEGER, INTENT(IN) :: comm\_keyval 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23MPI\_COMM\_DELETE\_ATTR(COMM, COMM\_KEYVAL, IERROR) 24INTEGER COMM, COMM\_KEYVAL, IERROR 2526Delete attribute from cache by key. This function invokes the attribute delete function 27comm\_delete\_attr\_fn specified when the keyval was created. The call will fail if the comm\_delete\_attr\_fn function returns an error code other than MPI\_SUCCESS. 28Whenever a communicator is replicated using the function MPI\_COMM\_DUP or 2930 MPI\_COMM\_IDUP, all call-back copy functions for attributes that are currently set are  $^{31}$ invoked (in arbitrary order). Whenever a communicator is deleted using the function 32 MPI\_COMM\_FREE all callback delete functions for attributes that are currently set are 33 invoked. 3435 6.7.3 Windows 36

The functions for caching on windows are:

MPI_WIN_CREATE	_KEYVAL(win_	_copy_	_attr_fr	, win	_delete_	_attr_f	n, win	_keyval,	extra_	_state)

IN	win_copy_attr_fn	copy callback function for win_keyval (function)
IN	win_delete_attr_fn	delete callback function for $win\_keyval$ (function)
OUT	win_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

int MPI\_Win\_create\_keyval(MPI\_Win\_copy\_attr\_function \*win\_copy\_attr\_fn,

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

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

INTEG	GER(KIND=MPI_ADDRES	SS_KIND) ATTRIBUTE_VAL	1 $2$
			3 4
MPI_WIN	_GET_ATTR(win, win_	_keyval, attribute_val, flag)	5
IN	win	window to which the attribute is attached (handle)	6
IN	win_keyval	key value (integer)	7
OUT	attribute_val	attribute value, unless $flag = false$	8
OUT	flag	false if no attribute is associated with the key (logical)	9 10 11
int MPI_V	√in_get_attr(MPI_Wi int *flag)	in win, int win_keyval, void *attribute_val,	11 12 13
TYPE INTE INTE LOGIO	(MPI_Win), INTENT(] GER, INTENT(IN) :: GER(KIND=MPI_ADDRES CAL, INTENT(OUT) ::	win_keyval SS_KIND), INTENT(OUT) :: attribute_val	14 15 16 17 18 19
INTEC INTEC	GER WIN, WIN_KEYVAI	KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) L, IERROR SS_KIND) ATTRIBUTE_VAL	20 21 22 23 24 25 26
MPI WIN	_DELETE_ATTR(win,	win keyval)	20 27
INOUT		window from which the attribute is deleted (handle)	28
			29
IN	win_keyval	key value (integer)	30
int MPI_V	Vin_delete_attr(MPI	[_Win win, int win_keyval)	31 32
MPI_Win_c	delete_attr(win, wi	n_kevval, ierror)	33
	(MPI_Win), INTENT(]	•	34 35
	GER, INTENT(IN) ::		36
INTE	GER, OPTIONAL, INTE	ENT(OUT) :: ierror	37
MPI_WIN_I	DELETE_ATTR(WIN, WI	IN_KEYVAL, IERROR)	38
INTE	GER WIN, WIN_KEYVAI	., IERROR	39
			40
6.7.4 Da	itatypes		41 42
	functions for caching o	n datatunes are:	43
THE HEW I	unctions for caching 0	n datatypes are.	44
			45
			46 47
			-11

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

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

```
1
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                      2
      attribute_val_out
                                                                                      3
      LOGICAL :: flag
                                                                                      4
and
                                                                                      5
ABSTRACT INTERFACE
                                                                                      6
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
                                                                                      7
  attribute_val, extra_state, ierror)
                                                                                      8
      TYPE(MPI_Datatype) :: datatype
                                                                                      9
      INTEGER :: type_keyval, ierror
                                                                                      10
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                      11
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                      12
                                                                                     13
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
                                                                                     14
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                      15
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
                                                                                      16
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
                                                                                      17
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
                                                                                      18
    LOGICAL FLAG
                                                                                      19
and
                                                                                      20
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
                                                                                     21
              EXTRA_STATE, IERROR)
                                                                                     22
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
                                                                                     23
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                     24
                                                                                      25
    If an attribute copy function or attribute delete function returns other than
                                                                                      26
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),
                                                                                     27
is erroneous.
                                                                                     28
                                                                                      29
MPI_TYPE_FREE_KEYVAL(type_keyval)
                                                                                      30
                                                                                      31
  INOUT
           type_keyval
                                     key value (integer)
                                                                                      32
                                                                                      33
int MPI_Type_free_keyval(int *type_keyval)
                                                                                     34
MPI_Type_free_keyval(type_keyval, ierror)
                                                                                     35
    INTEGER, INTENT(INOUT) :: type_keyval
                                                                                     36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     37
                                                                                      38
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
                                                                                      39
    INTEGER TYPE_KEYVAL, IERROR
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
                                                                                      45
                                                                                      46
                                                                                      47
```

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

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

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

```
1
                                       &gop_key, (void *)0) ) {
2
            /* get the key while assigning its copy and delete callback
3
               behavior. */
4
            } else
5
                MPI_Abort (comm, 99);
6
          }
7
8
          MPI_Comm_get_attr (comm, gop_key, &gop_stuff, &foundflag);
9
          if (foundflag)
10
          { /* This module has executed in this group before.
11
                We will use the cached information */
12
          }
13
          else
14
          { /* This is a group that we have not yet cached anything in.
15
               We will now do so.
            */
16
17
18
            /* First, allocate storage for the stuff we want,
19
                and initialize the reference count */
20
21
            gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));
22
            if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
23
24
            gop_stuff -> ref_count = 1;
25
26
            /* Second, fill in *gop_stuff with whatever we want.
27
                This part isn't shown here */
28
29
            /* Third, store gop_stuff as the attribute value */
30
            MPI_Comm_set_attr (comm, gop_key, gop_stuff);
31
          3
32
          /* Then, in any case, use contents of *gop_stuff
33
             to do the global op ... */
34
        }
35
36
        /* The following routine is called by MPI when a group is freed */
37
38
        int gop_stuff_destructor (MPI_Comm comm, int keyval, void *gop_stuffP,
39
                               void *extra)
40
        {
41
          gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
42
          if (keyval != gop_key) { /* abort -- programming error */ }
43
44
          /* The group's being freed removes one reference to gop_stuff */
45
          gop_stuff -> ref_count -= 1;
46
47
          /* If no references remain, then free the storage */
48
          if (gop_stuff -> ref_count == 0) {
```

```
free((void *)gop_stuff);
  }
  return MPI_SUCCESS;
}
/* The following routine is called by MPI when a group is copied */
int gop_stuff_copier (MPI_Comm comm, int keyval, void *extra,
 void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
{
  gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
  gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
  if (keyval != gop_key) { /* abort -- programming error */ }
  /* The new group adds one reference to this gop_stuff */
  gop_stuff_in -> ref_count += 1;
  *gop_stuff_out = gop_stuff_in;
  return MPI_SUCCESS;
}
```

# 6.8 Naming Objects

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

```
MPI_COMM_SET_NAME (comm, comm_name)
```

		,	
INOUT	comm	communicator whose identifier is to be set (handle)	30
			31
IN	comm_name	the character string which is remembered as the name	32
		(string)	33
			34
int MPI_C	omm_set_name(MPI_Comm com	m, const char *comm_name)	35
			36
MP1_Comm_	<pre>set_name(comm, comm_name,</pre>	ierror)	37
TYPE(	MPI_Comm), INTENT(IN) ::	comm	38
CHARA	CTER(LEN=*), INTENT(IN) :	. comm name	38
		_	39
INTEG	ER, OPTIONAL, INTENT(OUT)	:: lerror	40
MPI_COMM_	SET_NAME(COMM, COMM_NAME,	IERROR)	41
INTEG	ER COMM, IERROR		42
CHARA	CTER*(*) COMM_NAME		43
			44

MPI\_COMM\_SET\_NAME allows a user to associate a name string with a communicator. The character string which is passed to MPI\_COMM\_SET\_NAME will be saved inside the MPI library (so it can be freed by the caller immediately after the call, or allocated on the

283

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

45

46

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

1stack). Leading spaces in name are significant but trailing ones are not.  $\mathbf{2}$ MPI\_COMM\_SET\_NAME is a local (non-collective) operation, which only affects the 3 name of the communicator as seen in the process which made the MPI\_COMM\_SET\_NAME 4 call. There is no requirement that the same (or any) name be assigned to a communicator  $\mathbf{5}$ in every process where it exists. 6 Advice to users. Since MPI\_COMM\_SET\_NAME is provided to help debug code, it 7 8 is sensible to give the same name to a communicator in all of the processes where it exists, to avoid confusion. (End of advice to users.) 9 10 The length of the name which can be stored is limited to the value of 11 MPI\_MAX\_OBJECT\_NAME in Fortran and MPI\_MAX\_OBJECT\_NAME-1 in C to allow for the 12null terminator. Attempts to put names longer than this will result in truncation of the 13 name. MPI\_MAX\_OBJECT\_NAME must have a value of at least 64. 1415Advice to users. Under circumstances of store exhaustion an attempt to put a name 16of any length could fail, therefore the value of MPI\_MAX\_OBJECT\_NAME should be 17 viewed only as a strict upper bound on the name length, not a guarantee that setting 18 names of less than this length will always succeed. (End of advice to users.) 19 20Advice to implementors. Implementations which pre-allocate a fixed size space for a 21name should use the length of that allocation as the value of MPI\_MAX\_OBJECT\_NAME. 22 Implementations which allocate space for the name from the heap should still define 23MPI\_MAX\_OBJECT\_NAME to be a relatively small value, since the user has to allocate 24space for a string of up to this size when calling MPI\_COMM\_GET\_NAME. (End of 25advice to implementors.) 26272829MPI\_COMM\_GET\_NAME (comm, comm\_name, resultlen) 30 IN communicator whose name is to be returned (handle) comm  $^{31}$ OUT the name previously stored on the communicator, or comm\_name 32 an empty string if no such name exists (string) 33 34 OUT resultlen length of returned name (integer) 35 36 int MPI\_Comm\_get\_name(MPI\_Comm comm, char \*comm\_name, int \*resultlen) 37 MPI\_Comm\_get\_name(comm, comm\_name, resultlen, ierror) 38 TYPE(MPI\_Comm), INTENT(IN) :: comm 39 CHARACTER(LEN=MPI\_MAX\_OBJECT\_NAME), INTENT(OUT) :: 40 comm name INTEGER, INTENT(OUT) :: resultlen 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4243 MPI\_COMM\_GET\_NAME(COMM, COMM\_NAME, RESULTLEN, IERROR) 44 INTEGER COMM, RESULTLEN, IERROR 45CHARACTER\*(\*) COMM\_NAME 46 47MPI\_COMM\_GET\_NAME returns the last name which has previously been associated 48 with the given communicator. The name may be set and retrieved from any language. The

same name will be returned independent of the language used. name should be allocated so that it can hold a resulting string of length MPI\_MAX\_OBJECT\_NAME characters. MPI\_COMM\_GET\_NAME returns a copy of the set name in name.

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

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

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

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

(End of rationale.)

Advice to users. The above definition means that it is safe simply to print the string returned by MPI\_COMM\_GET\_NAME, as it is always a valid string even if there was no name.

Note that associating a name with a communicator has no effect on the semantics of an MPI program, and will (necessarily) increase the store requirement of the program, since the names must be saved. Therefore there is no requirement that users use these functions to associate names with communicators. However debugging and profiling MPI applications may be made easier if names are associated with communicators, since the debugger or profiler should then be able to present information in a less cryptic manner. (*End of advice to users.*)

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

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

 $\frac{44}{45}$ 

46

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

```
1
     MPI_TYPE_SET_NAME (datatype, type_name)
2
       INOUT
                 datatype
                                             datatype whose identifier is to be set (handle)
3
       IN
                 type_name
                                             the character string which is remembered as the name
4
                                             (string)
5
6
\overline{7}
     int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)
8
     MPI_Type_set_name(datatype, type_name, ierror)
9
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
          CHARACTER(LEN=*), INTENT(IN) :: type_name
11
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
13
14
          INTEGER DATATYPE, IERROR
15
          CHARACTER*(*) TYPE_NAME
16
17
18
     MPI_TYPE_GET_NAME (datatype, type_name, resultlen)
19
       IN
                 datatype
                                             datatype whose name is to be returned (handle)
20
       OUT
                 type_name
                                             the name previously stored on the datatype, or a empty
21
                                             string if no such name exists (string)
22
23
       OUT
                 resultlen
                                             length of returned name (integer)
^{24}
25
     int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,
26
                     int *resultlen)
27
     MPI_Type_get_name(datatype, type_name, resultlen, ierror)
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
30
          CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
^{31}
          INTEGER, INTENT(OUT) :: resultlen
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
34
          INTEGER DATATYPE, RESULTLEN, IERROR
35
          CHARACTER*(*) TYPE_NAME
36
37
          Named predefined datatypes have the default names of the datatype name. For exam-
38
     ple, MPI_WCHAR has the default name of MPI_WCHAR.
39
          The following functions are used for setting and getting names of windows. The con-
40
     stant MPI_MAX_OBJECT_NAME also applies to these names.
41
42
     MPI_WIN_SET_NAME (win, win_name)
43
44
       INOUT
                                             window whose identifier is to be set (handle)
                 win
45
       IN
                 win_name
                                             the character string which is remembered as the name
46
                                             (string)
47
48
```

1 int MPI\_Win\_set\_name(MPI\_Win win, const char \*win\_name) 2 MPI\_Win\_set\_name(win, win\_name, ierror) TYPE(MPI\_Win), INTENT(IN) :: win 4 CHARACTER(LEN=\*), INTENT(IN) :: win\_name 5 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 6 MPI\_WIN\_SET\_NAME(WIN, WIN\_NAME, IERROR) 7 INTEGER WIN, IERROR 9 CHARACTER\*(\*) WIN\_NAME 10 11 12MPI\_WIN\_GET\_NAME (win, win\_name, resultlen) 13 IN win window whose name is to be returned (handle) 14OUT 15win\_name the name previously stored on the window, or a empty 16string if no such name exists (string) 17OUT resultlen length of returned name (integer) 18 19 int MPI\_Win\_get\_name(MPI\_Win win, char \*win\_name, int \*resultlen) 2021MPI\_Win\_get\_name(win, win\_name, resultlen, ierror) 22 TYPE(MPI\_Win), INTENT(IN) :: win 23CHARACTER(LEN=MPI\_MAX\_OBJECT\_NAME), INTENT(OUT) :: win\_name 24INTEGER, INTENT(OUT) :: resultlen 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 26MPI\_WIN\_GET\_NAME(WIN, WIN\_NAME, RESULTLEN, IERROR) 27INTEGER WIN, RESULTLEN, IERROR 28 CHARACTER\*(\*) WIN\_NAME 29 30 316.9 Formalizing the Loosely Synchronous Model 32 33 In this section, we make further statements about the loosely synchronous model, with 34

## 6.9.1 Basic Statements

particular attention to intra-communication.

When a caller passes a communicator (that contains a context and group) to a callee, that communicator must be free of side effects throughout execution of the subprogram: there should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work "safely." For libraries so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communications will interfere. Since we permit good implementations to create new communicators without synchronization (such as by preallocated contexts on communicators), this does not impose a significant overhead.

46 47 48

35 36

37 38

39

40

41

42

43

44

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

# 6.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a parallel procedure is effected by
 having each executing process invoke the procedure. The invocation is a collective operation:
 it is executed by all processes in the execution group, and invocations are similarly ordered
 at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

16

1

 $\mathbf{2}$ 

3

4 5

6

## 17 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.

27 28

29

## 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.

<sup>33</sup> In such a case, a new communicator needs to be dynamically allocated for each new <sup>34</sup> invocation of a parallel procedure. The allocation is done by the caller. A new communicator <sup>35</sup> can be generated by a call to MPI\_COMM\_DUP, if the callee execution group is identical to <sup>36</sup> the caller execution group, or by a call to MPI\_COMM\_SPLIT if the caller execution group <sup>37</sup> is split into several subgroups executing distinct parallel routines. The new communicator <sup>38</sup> 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:

 $46 \\ 47$ 

48

• messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;

• messages are always selected by source (no use is made of MPI\_ANY\_SOURCE).

#### The General Case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, communicator creation is properly coordinated.  $\overline{7}$ 

1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
23			
24 25			
25 26			
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			
46			
47			
48			

# Chapter 7

# **Process Topologies**

# 7.1 Introduction

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

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

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal with machine-independent mapping and communication on virtual process topologies.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [44]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [11, 12].

 $^{24}$ 

 $^{31}$ 

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and notational power in message-passing programming. (End of rationale.)

#### 7.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping.

16Specifying the virtual topology in terms of a graph is sufficient for all applications. 17However, in many applications the graph structure is regular, and the detailed set-up of the 18 graph would be inconvenient for the user and might be less efficient at run time. A large frac-19tion of all parallel applications use process topologies like rings, two- or higher-dimensional 20grids, or tori. These structures are completely defined by the number of dimensions and 21the numbers of processes in each coordinate direction. Also, the mapping of grids and tori 22is generally an easier problem than that of general graphs. Thus, it is desirable to address 23these cases explicitly.  $^{24}$ 

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for 26example, the relation between group rank and coordinates for four processes in a  $(2 \times 2)$ grid is as follows.

coord $(0,0)$ :	$\operatorname{rank} 0$
coord $(0,1)$ :	rank 1
coord $(1,0)$ :	rank $2$
coord $(1,1)$ :	$\operatorname{rank} 3$

#### 7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

#### Overview of the Functions 7.4

43 MPI supports three topology types: Cartesian, graph, and distributed graph. The 44function MPI\_CART\_CREATE is used to create Cartesian topologies, the function 45

MPI\_GRAPH\_CREATE is used to create graph topologies, and the functions 46

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT and MPI\_DIST\_GRAPH\_CREATE are used to cre-47

ate distributed graph topologies. These topology creation functions are collective. As with 48

# **Unofficial Draft for Comment Only**

1

2

3

4  $\mathbf{5}$ 

> 6 7

> > 8

9

10

11

12

13

14

15

25

27

35 36

37

38

39

40 41

other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm\_old, which defines the set of processes on which the topology is to be mapped. For MPI\_GRAPH\_CREATE and MPI\_CART\_CREATE, all input arguments must have identical values on all processes of the group of comm\_old. When calling MPI\_GRAPH\_CREATE, each process specifies all nodes and edges in the graph. In contrast, the functions MPI\_DIST\_GRAPH\_CREATE\_ADJACENT or MPI\_DIST\_GRAPH\_CREATE are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator comm\_topol is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI\_COMM\_CREATE, no cached information propagates from comm\_old to comm\_topol.

MPI\_CART\_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an *n*-dimensional hypercube is an *n*-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI\_DIMS\_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

MPI defines functions to query a communicator for topology information. The function 22MPI\_TOPO\_TEST is used to query for the type of topology associated with a communicator. 23Depending on the topology type, different information can be extracted. For a graph topology, the functions MPI\_GRAPHDIMS\_GET and MPI\_GRAPH\_GET return the values that were specified in the call to MPI\_GRAPH\_CREATE. Additionally, the functions MPI\_GRAPH\_NEIGHBORS\_COUNT and MPI\_GRAPH\_NEIGHBORS can be used to obtain 2728the neighbors of an arbitrary node in the graph. For a distributed graph topology, the functions MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT and MPI\_DIST\_GRAPH\_NEIGHBORS 2930 can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI\_CARTDIM\_GET and MPI\_CART\_GET return the values that were specified in the call to MPI\_CART\_CREATE. Additionally, the functions MPI\_CART\_RANK and 33 MPI\_CART\_COORDS translate Cartesian coordinates into a group rank, and vice-versa. 34The function MPI\_CART\_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function MPI\_CART\_SUB can be used to extract a Cartesian subspace (analogous to MPI\_COMM\_SPLIT). This function is collective over the input communicator's group.

The two additional functions, MPI\_GRAPH\_MAP and MPI\_CART\_MAP, are, in general, not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation.

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

MPI\_NEIGHBOR\_ALLTOALLV, and MPI\_NEIGHBOR\_ALLTOALLW communicate with the 4546nearest neighbors on the topology associated with the communicator. The nonblocking 47variants are MPI\_INEIGHBOR\_ALLGATHER, MPI\_INEIGHBOR\_ALLGATHERV, 48 MPI\_INEIGHBOR\_ALLTOALL, MPI\_INEIGHBOR\_ALLTOALLV, and

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8 9

10 11

12

13

14

15

16

17

18

1920

21

 $^{24}$ 

2526

31

32

35

36 37

38

39

40 41

42

43

```
1
     MPI_INEIGHBOR_ALLTOALLW.
\mathbf{2}
3
     7.5
            Topology Constructors
4
5
     7.5.1 Cartesian Constructor
6
7
8
     MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
9
10
       IN
                  comm_old
                                              input communicator (handle)
11
       IN
                  ndims
                                              number of dimensions of Cartesian grid (integer)
12
       IN
                  dims
                                              integer array of size ndims specifying the number of
13
                                              processes in each dimension
14
15
       IN
                  periods
                                              logical array of size ndims specifying whether the grid
16
                                              is periodic (true) or not (false) in each dimension
17
       IN
                  reorder
                                              ranking may be reordered (true) or not (false) (logical)
18
       OUT
                                              communicator with new Cartesian topology (handle)
                  comm_cart
19
20
     int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
21
                     const int periods[], int reorder, MPI_Comm *comm_cart)
22
23
     MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
^{24}
          TYPE(MPI_Comm), INTENT(IN) :: comm_old
25
          INTEGER, INTENT(IN) :: ndims, dims(ndims)
26
          LOGICAL, INTENT(IN) :: periods(ndims), reorder
27
          TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
29
     MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
30
          INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
^{31}
          LOGICAL PERIODS(*), REORDER
32
33
          MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian
34
     topology information is attached. If reorder = false then the rank of each process in the
35
     new group is identical to its rank in the old group. Otherwise, the function may reorder
36
     the processes (possibly so as to choose a good embedding of the virtual topology onto
37
     the physical machine). If the total size of the Cartesian grid is smaller than the size of
38
     the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to
39
     MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created.
40
     The call is erroneous if it specifies a grid that is larger than the group size or if ndims is
41
     negative.
42
43
            Cartesian Convenience Function: MPI_DIMS_CREATE
     7.5.2
44
45
     For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced
```

<sup>40</sup> For Cartesian topologies, the function MPI\_DIMS\_CREATE helps the user select a balanced
 <sup>46</sup> distribution of processes per coordinate direction, depending on the number of processes
 <sup>47</sup> 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)

IN	nnodes	number of nodes in a grid (integer)	6
IN	ndims	much an of Chartenian dimensions (internet)	7
IIN	naims	number of Cartesian dimensions (integer)	8
INOUT	dims	integer array of size ndims specifying the number of	9
		nodes in each dimension	10
			11
int MPI_Di	ms_create(int nnodes, in	t ndims, int dims[])	12
		· · · · · · · · · · · · · · · · · · ·	13
	reate(nnodes, ndims, dim	•	14
	R, INTENT(IN) :: nnodes	-	15
INTEGE	R, INTENT(INOUT) :: dim	us(ndims)	16
INTEGE	R, OPTIONAL, INTENT(OUT)	:: ierror	17
MDT DTMG C	REATE(NNODES, NDIMS, DIM	Ις ΤΕΡΡΟΡ)	18
			19
INIEGE	R NNODES, NDIMS, DIMS(*)	, IEKKUK	19

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions and a total of nnodes nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array dims. If dims[i] is set to a positive number, the routine will not modify the number of nodes in dimension i; only those entries where dims[i] = 0 are modified by the call.

Negative input values of  $\mathsf{dims}[i]$  are erroneous. An error will occur if  $\mathsf{nnodes}$  is not a multiple of

$$\prod_{i,dims[i]\neq 0} dims[i].$$

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

#### Example 7.1

dims	function call	dims
before call		on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

1 2

3 4

5

20

21

22

23

 $^{24}$ 

25

26

27

28 29

30

 $^{31}$ 

32 33

1

2 3 4

 $\mathbf{5}$ 

## 7.5.3 Graph Constructor

MPI\_GRAPH\_CREATE(comm\_old, nnodes, index, edges, reorder, comm\_graph)

5			
6	IN	comm_old	input communicator (handle)
7	IN	nnodes	number of nodes in graph (integer)
8	IN	index	array of integers describing node degrees (see below)
9 10	IN	edges	array of integers describing graph edges (see below)
11	IN	reorder	ranking may be reordered (true) or not (false) (logical)
12	OUT	comm_graph	communicator with graph topology added (handle)
13 14			
15	int MPI_Gr	•	_old, int nnodes, const int index[],
16		const int edges[], in	nt reorder, MPI_Comm *comm_graph)
17	MPI_Graph_		index, edges, reorder, comm_graph,
18		ierror)	
19	TYPE(M	<pre>IPI_Comm), INTENT(IN) ::</pre>	comm_old
20	INTEGE	CR, INTENT(IN) :: nnodes	, index(nnodes), edges(*)
21	LOGICA	L, INTENT(IN) :: reorde	r
22	TYPE(M	<pre>IPI_Comm), INTENT(OUT) ::</pre>	comm_graph
23		CR, OPTIONAL, INTENT(OUT)	
24			
25	MP1_GRAPH_		INDEX, EDGES, REORDER, COMM_GRAPH,
26		IERROR)	
27			X(*), EDGES(*), COMM_GRAPH, IERROR
28	LOGICA	L REORDER	

29MPI\_GRAPH\_CREATE returns a handle to a new communicator to which the graph 30topology information is attached. If reorder = false then the rank of each process in the  $^{31}$ new group is identical to its rank in the old group. Otherwise, the function may reorder the 32 processes. If the size, nnodes, of the graph is smaller than the size of the group of comm\_old, 33 then some processes are returned MPI\_COMM\_NULL, in analogy to MPI\_CART\_CREATE 34and MPI\_COMM\_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI\_COMM\_NULL 35 is returned in all processes. The call is erroneous if it specifies a graph that is larger than 36 the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the
 following simple example.

46

```
47 Example 7.2
```

48

## Unofficial Draft for Comment Only

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

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

Then, the input arguments are:

$\overline{7}$
8

 $^{24}$ 

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

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

In Fortran, index(1) is the degree of node zero, and index(i+1) - index(i) is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges(j), for  $1 \le j \le$  index(1) and the list of neighbors of node i, i > 0, is stored in edges(j), index(i)+1  $\le j \le$  index(i+1).

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.

Advice to users. Performance implications of using multiple edges or a non-symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (*End of advice to users.*)

Advice to implementors. The following topology information is likely to be stored with a communicator:

- Type of topology (Cartesian/graph),
- For a Cartesian topology:
  - 1. ndims (number of dimensions),
  - 2. dims (numbers of processes per coordinate direction),
  - 3. periods (periodicity information),
  - 4. own\_position (own position in grid, could also be computed from rank and dims)
- For a graph topology:

index,
 edges,

which are the vectors defining the graph structure.

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array index simplifies access to the topology information. (*End of advice to implementors.*) 45

#### 1 7.5.4 Distributed Graph Constructor 2

MPI\_GRAPH\_CREATE requires that each process passes the full (global) communication 3 graph to the call. This limits the scalability of this constructor. With the distributed graph interface, the communication graph is specified in a fully distributed fashion. Each process 5specifies only the part of the communication graph of which it is aware. Typically, this 6 could be the set of processes from which the process will eventually receive or get data. or the set of processes to which the process will send or put data, or some combination of such edges. Two different interfaces can be used to create a distributed graph topology. 9 MPI\_DIST\_GRAPH\_CREATE\_ADJACENT creates a distributed graph communicator with 10 each process specifying each of its incoming and outgoing (adjacent) edges in the logical 11 communication graph and thus requires minimal communication during creation. 12

MPI\_DIST\_GRAPH\_CREATE provides full flexibility such that any process can indicate that 13 communication will occur between any pair of processes in the graph. 14

To provide better possibilities for optimization by the MPI library, the distributed 15graph constructors permit weighted communication edges and take an info argument that 16 can further influence process reordering or other optimizations performed by the MPI library. 17For example, hints can be provided on how edge weights are to be interpreted, the quality 18 of the reordering, and/or the time permitted for the MPI library to process the graph. 19

2021

22 23

4

7

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights,	
outdegree, destinations, destweights, info, reorder, comm_dist_graph)	

23 24	IN	comm_old	input communicator (handle)
25 26	IN	indegree	size of sources and source weights arrays (non-negative integer)
27 28	IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)
29 30 31	IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)
32 33	IN	outdegree	size of destinations and destweights $\operatorname{arrays}$ (non-negative integer)
34 35	IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)
36 37 38	IN	destweights	weights of the edges out of the calling process (array of non-negative integers)
39 40	IN	info	hints on optimization and interpretation of weights (handle)
41 42	IN	reorder	the ranks may be reordered (true) or not (false) (logical)
43 44 45	OUT	comm_dist_graph	communicator with distributed graph topology (handle)
46 47 48	int MPI_D		(MPI_Comm comm_old, int indegree, const int sourceweights[], int outdegree,
			· · · · · · · · · · · · · · · · · · ·

```
1
              const int destinations[], const int destweights[],
                                                                                   \mathbf{2}
              MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
                                                                                   3
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
                                                                                   4
              outdegree, destinations, destweights, info, reorder,
                                                                                   5
              comm_dist_graph, ierror)
                                                                                   6
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                   7
    INTEGER, INTENT(IN) :: indegree, sources(indegree), outdegree,
                                                                                   8
        destinations(outdegree)
                                                                                   9
    INTEGER, INTENT(IN) :: sourceweights(*), destweights(*)
                                                                                   10
    TYPE(MPI_Info), INTENT(IN) ::
                                    info
                                                                                   11
    LOGICAL, INTENT(IN) ::
                             reorder
                                                                                   12
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   14
                                                                                   15
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
                                                                                   16
              OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
                                                                                   17
              COMM_DIST_GRAPH, IERROR)
                                                                                   18
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
        DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
                                                                                   19
                                                                                  20
    LOGICAL REORDER
```

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT returns a handle to a new communicator to which the distributed graph topology information is attached. Each process passes all information about its incoming and outgoing edges in the virtual distributed graph topology. The calling processes must ensure that each edge of the graph is described in the source and in the destination process with the same weights. If there are multiple edges for a given (source,dest) pair, then the sequence of the weights of these edges does not matter. The complete communication topology is the combination of all edges shown in the sources arrays of all processes in comm\_old, which must be identical to the combination of all edges shown in the destinations arrays. Source and destination ranks must be process ranks of comm\_old. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that have specified indegree and outdegree as zero and thus do not occur as source or destination rank in the graph specification) are allowed.

The call creates a new communicator comm\_dist\_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm\_dist\_graph is identical to the number of processes in comm\_old. The call to MPI\_DIST\_GRAPH\_CREATE\_ADJACENT is collective.

Weights are specified as non-negative integers and can be used to influence the process 39 remapping strategy and other internal MPI optimizations. For instance, approximate count 40 arguments of later communication calls along specific edges could be used as their edge 41 weights. Multiplicity of edges can likewise indicate more intense communication between 42pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 43 standard and is left to the implementation. In C or Fortran, an application can supply 44the special value MPI\_UNWEIGHTED for the weight array to indicate that all edges have 45the same (effectively no) weight. It is erroneous to supply MPI\_UNWEIGHTED for some 46but not all processes of comm\_old. If the graph is weighted but indegree or outdegree is 47zero, then MPI\_WEIGHTS\_EMPTY or any arbitrary array may be passed to sourceweights 48

#### Unofficial Draft for Comment Only

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

or destweights respectively. Note that MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are  $\mathbf{2}$ not special weight values; rather they are special values for the total array argument. In 3 Fortran, MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are objects like MPI\_BOTTOM (not 4 usable for initialization or assignment). See Section 2.5.4. 5In the case of an empty weights array argument passed while Advice to users. 6 constructing a weighted graph, one should not pass NULL because the value of 7 MPI\_UNWEIGHTED may be equal to NULL. The value of this argument would then 8 be indistinguishable from MPI\_UNWEIGHTED to the implementation. In this case 9 MPI\_WEIGHTS\_EMPTY should be used instead. (End of advice to users.) 10 11 Advice to implementors. It is recommended that MPI\_UNWEIGHTED not be imple-12mented as NULL. (End of advice to implementors.) 13 14*Rationale.* To ensure backward compatibility, MPI\_UNWEIGHTED may still be imple-15mented as NULL. See Annex B.3. (End of rationale.) 1617The meaning of the info and reorder arguments is defined in the description of the 18 following routine. 1920MPI\_DIST\_GRAPH\_CREATE(comm\_old, n, sources, degrees, destinations, weights, info, 21reorder, comm\_dist\_graph) 22 23IN comm\_old input communicator (handle)  $^{24}$ number of source nodes for which this process specifies IN n 25edges (non-negative integer) 26IN 27sources array containing the n source nodes for which this process specifies edges (array of non-negative integers) 2829IN degrees array specifying the number of destinations for each 30 source node in the source node array (array of non- $^{31}$ negative integers) 32 IN destinations destination nodes for the source nodes in the source 33 node array (array of non-negative integers) 34 IN weights weights for source to destination edges (array of non-35 negative integers) 36 37 IN info hints on optimization and interpretation of weights 38 (handle) 39 IN reorder the process may be reordered (true) or not (false) (log-40 ical) 41 OUT comm\_dist\_graph communicator with distributed graph topology added 42(handle) 43 4445int MPI\_Dist\_graph\_create(MPI\_Comm comm\_old, int n, const int sources[], 46const int degrees[], const int destinations[], 47const int weights[], MPI\_Info info, int reorder, 48 MPI\_Comm \*comm\_dist\_graph)

**Unofficial Draft for Comment Only** 

```
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
             info, reorder, comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*)
    INTEGER, INTENT(IN) :: weights(*)
    TYPE(MPI_Info), INTENT(IN) ::
                                   info
    LOGICAL, INTENT(IN) ::
                            reorder
    TYPE(MPI_Comm), INTENT(OUT) ::
                                    comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
             INFO, REORDER, COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
        WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
```

MPI\_DIST\_GRAPH\_CREATE returns a handle to a new communicator to which the distributed graph topology information is attached. Concretely, each process calls the constructor with a set of directed (source, destination) communication edges as described below. Every process passes an array of n source nodes in the sources array. For each source node, a 20non-negative number of destination nodes is specified in the degrees array. The destination 21nodes are stored in the corresponding consecutive segment of the destinations array. More 22precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the 23j-th such edge stored in destinations[degrees[0]+ $\dots$ +degrees[i-1]+j]. The weight of this edge is stored in weights[degrees[0]+ $\ldots$ +degrees[i-1]+i]. Both the sources and the destinations arrays may contain the same node more than once, and the order in which nodes are listed as destinations or sources is not significant. Similarly, different processes may specify edges with the same source and destination nodes. Source and destination nodes must be process ranks of comm\_old. Different processes may specify different numbers of source and destination nodes, as well as different source to destination edges. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that do not occur as source or destination node in the graph specification) are allowed.

The call creates a new communicator **comm\_dist\_graph** of distributed graph topology type to which topology information has been attached. The number of processes in comm\_dist\_graph is identical to the number of processes in comm\_old. The call to MPI\_DIST\_GRAPH\_CREATE is collective.

If reorder = false, all processes will have the same rank in comm\_dist\_graph as in  $comm_old$ . If reorder = true then the MPI library is free to remap to other processes (of comm\_old) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering.

42Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count 4344arguments of later communication calls along specific edges could be used as their edge 45weights. Multiplicity of edges can likewise indicate more intense communication between 46pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 47standard and is left to the implementation. In C or Fortran, an application can supply

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9 10

11

12

13

14

1516

17

18

19

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

the special value MPI\_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. It is erroneous to supply MPI\_UNWEIGHTED for some but not all processes of comm\_old. If the graph is weighted but n = 0, then MPI\_WEIGHTS\_EMPTY or any arbitrary array may be passed to weights. Note that MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are not special weight values; rather they are special values for the total array argument. In Fortran, MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are objects like MPI\_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

- Advice to users. In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI\_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI\_UNWEIGHTED to the implementation.
- 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.3. (*End of rationale.*)

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is valid for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

27 28

29

30

 $^{31}$ 

32

33

34

35

36

37

9

10

11

12

13 14

15

16 17

18

19 20

- Advice to implementors. MPI implementations must document any additionally supported key-value info pairs. MPI\_INFO\_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.
- An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI\_GRAPH\_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors*.)
- 38 39 40

41

**Example 7.3** As for Example 7.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

process	neighbors	
0	1, 3	
1	0	
2	3	
3	0, 2	
	$ \begin{array}{c} 0\\ 1\\ 2\\ 3 \end{array} $	$\begin{array}{ccc} 0 & 1, 3 \\ 1 & 0 \end{array}$

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

process	n	sources	degrees	destinations	weights
0	1	0	2	1,3	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

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

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

In both cases above, the application could supply MPI\_UNWEIGHTED instead of explicitly providing identical weights.

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT could be used to specify this graph using the following arguments:

process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	1,3	1,1	2	$1,\!3$	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	$_{0,2}$	$1,\!1$	2	0,2	1,1

**Example 7.4** A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be modeled with Cartesian topologies, but can easily be captured with MPI\_DIST\_GRAPH\_CREATE as shown in the following code. In this example, the communication along the dimensions is twice as heavy as the communication along the diagonals:

 $^{31}$ 

```
1
\mathbf{2}
     /* get my communication partners along x dimension */
3
     destinations[0] = P*y+(x+1)%P; weights[0] = 2;
4
     destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
5
6
     /* get my communication partners along y dimension */
7
     destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
     destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;
8
9
10
     /* get my communication partners along diagonals */
11
     destinations[4] = P*((y+1))(Q)+(x+1)(P); weights[4] = 1;
     destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
12
13
     destinations[6] = P*((y+1))(Q)+(P+x-1)(P); weights[6] = 1;
14
     destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
15
16
     sources[0] = rank;
17
     degrees[0] = 8;
18
     MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
19
                             weights, MPI_INFO_NULL, 1, &comm_dist_graph);
20
21
     7.5.5
            Topology Inquiry Functions
22
     If a topology has been defined with one of the above functions, then the topology information
23
     can be looked up using inquiry functions. They all are local calls.
24
25
26
     MPI_TOPO_TEST(comm, status)
27
       IN
                 comm
                                           communicator (handle)
28
29
       OUT
                                           topology type of communicator comm (state)
                 status
30
^{31}
     int MPI_Topo_test(MPI_Comm comm, int *status)
32
     MPI_Topo_test(comm, status, ierror)
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         INTEGER, INTENT(OUT) :: status
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_TOPO_TEST(COMM, STATUS, IERROR)
38
          INTEGER COMM, STATUS, IERROR
39
         The function MPI_TOPO_TEST returns the type of topology that is assigned to a
40
     communicator.
41
         The output value status is one of the following:
42
43
       MPI_GRAPH
                                              graph topology
44
                                              Cartesian topology
       MPI_CART
45
                                             distributed graph topology
       MPI_DIST_GRAPH
46
                                              no topology
       MPI_UNDEFINED
47
48
```

MPI\_GRAPHDIMS\_GET(comm, nnodes, nedges)

MPI_GRAH	PHDIMS_GET(comm, nnodes,	nedges)	1
IN	comm	communicator for group with graph structure (handle)	2 3
OUT	nnodes	number of nodes in graph (integer) (same as number	4
		of processes in the group)	5
OUT	nedges	number of edges in graph (integer)	6
			7
int MPI_G	raphdims_get(MPI_Comm co	mm, int *nnodes, int *nedges)	8 9
MPI_Graph	dims_get(comm, nnodes, n	edges, ierror)	10
	MPI_Comm), INTENT(IN) ::	comm	11
	ER, INTENT(OUT) :: nnod ER, OPTIONAL, INTENT(OUT	0	12
			13 14
	DIMS_GET(COMM, NNODES, N		15
INTEG	ER COMM, NNODES, NEDGES,	LERKUR	16
		and MPI_GRAPH_GET retrieve the graph-topology	17
		communicator by MPI_GRAPH_CREATE. _GRAPHDIMS_GET can be used to dimension the	18 19
	- 0	e following call to MPI_GRAPH_GET.	20
			21
	PH_GET(comm, maxindex, ma	vodros indov odros)	22
	· ·	,	23
IN	comm	communicator with graph structure (handle)	24 25
IN	maxindex	length of vector index in the calling program	26
	1	(integer)	27
IN	maxedges	length of vector <b>edges</b> in the calling program (integer)	28
OUT	index	array of integers containing the graph structure (for	29 30
001	Index	details see the definition of MPI_GRAPH_CREATE)	31
OUT	edges	array of integers containing the graph structure	32
		and, of meeders concerning one Brokin portaceare	33
int MPI_G	raph_get(MPI_Comm comm,	<pre>int maxindex, int maxedges, int index[],</pre>	34 35
	<pre>int edges[])</pre>		36
MPI_Graph	_get(comm, maxindex, max	edges, index, edges, ierror)	37
TYPE(	MPI_Comm), INTENT(IN) ::	comm	38
	ER, INTENT(IN) :: maxin		39
	ER, INTENT(OUT) :: inde ER, OPTIONAL, INTENT(OUT	x(maxindex), edges(maxedges)	40 41
			41
		EDGES, INDEX, EDGES, IERROR)	43
INTEG	ER CUMM, MAXINDEX, MAXED	GES, INDEX(*), EDGES(*), IERROR	44
			45
			46 47
			48

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

MPI\_CART\_RANK(comm, coords, rank) 1  $\mathbf{2}$ IN communicator with Cartesian structure (handle) comm 3 IN coords integer array (of size ndims) specifying the Cartesian 4 coordinates of a process 5 6 OUT rank rank of specified process (integer) 7 int MPI\_Cart\_rank(MPI\_Comm comm, const int coords[], int \*rank) 9 MPI\_Cart\_rank(comm, coords, rank, ierror) 10 TYPE(MPI\_Comm), INTENT(IN) :: comm 11 INTEGER, INTENT(IN) :: coords(\*) 12INTEGER, INTENT(OUT) :: rank 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 14 15MPI\_CART\_RANK(COMM, COORDS, RANK, IERROR) 16INTEGER COMM, COORDS(\*), RANK, IERROR 17For a process group with Cartesian structure, the function MPI\_CART\_RANK trans-18 lates the logical process coordinates to process ranks as they are used by the point-to-point 19 routines. 20For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that 21is, coords(i) < 0 or  $coords(i) \ge dims(i)$ , it is shifted back to the interval 22  $0 \leq coords(i) < dims(i)$  automatically. Out-of-range coordinates are erroneous for non-23periodic dimensions.  $^{24}$ If comm is associated with a zero-dimensional Cartesian topology, coords is not signif-25icant and 0 is returned in rank. 262728MPI\_CART\_COORDS(comm, rank, maxdims, coords) 29 IN communicator with Cartesian structure (handle) comm 30 31IN rank of a process within group of comm (integer) rank 32 IN maxdims length of vector **coords** in the calling program (integer) 33 OUT coords integer array (of size ndims) containing the Cartesian 34 coordinates of specified process (array of integers) 35 36 37 int MPI\_Cart\_coords(MPI\_Comm comm, int rank, int maxdims, int coords[]) 38 MPI\_Cart\_coords(comm, rank, maxdims, coords, ierror) 39 TYPE(MPI\_Comm), INTENT(IN) :: comm 40 INTEGER, INTENT(IN) :: rank, maxdims 41 INTEGER, INTENT(OUT) :: coords(maxdims) 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44 MPI\_CART\_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR) 45INTEGER COMM, RANK, MAXDIMS, COORDS(\*), IERROR 46The inverse mapping, rank-to-coordinates translation is provided by 47MPI\_CART\_COORDS.

1 If comm is associated with a zero-dimensional Cartesian topology,  $\mathbf{2}$ coords will be unchanged. 3 4 MPI\_GRAPH\_NEIGHBORS\_COUNT(comm, rank, nneighbors) 56 IN communicator with graph topology (handle) comm 7 IN rank of process in group of comm (integer) rank 8 OUT nneighbors number of neighbors of specified process (integer) 9 10 11int MPI\_Graph\_neighbors\_count(MPI\_Comm comm, int rank, int \*nneighbors) 12MPI\_Graph\_neighbors\_count(comm, rank, nneighbors, ierror) 13 TYPE(MPI\_Comm), INTENT(IN) :: comm 14INTEGER, INTENT(IN) :: rank 15INTEGER, INTENT(OUT) :: nneighbors 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1718 MPI\_GRAPH\_NEIGHBORS\_COUNT(COMM, RANK, NNEIGHBORS, IERROR) 19INTEGER COMM, RANK, NNEIGHBORS, IERROR 202122MPI\_GRAPH\_NEIGHBORS(comm, rank, maxneighbors, neighbors) 23IN communicator with graph topology (handle) comm  $^{24}$ IN rank rank of process in group of comm (integer) 2526IN maxneighbors size of array neighbors (integer) 27OUT neighbors ranks of processes that are neighbors to specified pro-28cess (array of integer) 29 30 int MPI\_Graph\_neighbors(MPI\_Comm comm, int rank, int maxneighbors,  $^{31}$ int neighbors[]) 32 33 MPI\_Graph\_neighbors(comm, rank, maxneighbors, neighbors, ierror) 34 TYPE(MPI\_Comm), INTENT(IN) :: comm 35 INTEGER, INTENT(IN) :: rank, maxneighbors 36 INTEGER, INTENT(OUT) :: neighbors(maxneighbors) 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 MPI\_GRAPH\_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR) 39 INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(\*), IERROR 4041 MPI\_GRAPH\_NEIGHBORS\_COUNT and MPI\_GRAPH\_NEIGHBORS provide adjacency 42information for a graph topology. The returned count and array of neighbors for the queried 43 rank will both include all neighbors and reflect the same edge ordering as was specified by 44 the original call to MPI\_GRAPH\_CREATE. Specifically, MPI\_GRAPH\_NEIGHBORS\_COUNT 45 and MPI\_GRAPH\_NEIGHBORS will return values based on the original index and edges array 46 passed to MPI\_GRAPH\_CREATE (for the purpose of this example, we assume that index[-1] 47is zero): 48

#### 7.5. TOPOLOGY CONSTRUCTORS

- The number of neighbors (nneighbors) returned from MPI\_GRAPH\_NEIGHBORS\_COUNT will be (index[rank] - index[rank-1]).
- The neighbors array returned from MPI\_GRAPH\_NEIGHBORS will be edges[index[rank-1]] through edges[index[rank]-1].

#### Example 7.5

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix (note that some neighbors are listed multiple times):

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

Thus, the input arguments to MPI\_GRAPH\_CREATE are:

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

Therefore, calling MPI\_GRAPH\_NEIGHBORS\_COUNT and MPI\_GRAPH\_NEIGHBORS for each of the 4 processes will return:

Input rank	Count	Neighbors
0	3	1, 1, 3
1	2	0, 0
2	1	3
3	3	0, 2, 2

#### Example 7.6

Suppose that comm is a communicator with a shuffle-exchange topology. The group has  $2^n$  members. Each process is labeled by  $a_1, \ldots, a_n$  with  $a_i \in \{0, 1\}$ , and has three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$  ( $\bar{a} = 1 - a$ ), shuffle $(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$ , and unshuffle $(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$ . The graph adjacency list is illustrated below for n = 3.

1	node	exchange	shuffle	unshuffle
		neighbors(1)	neighbors(2)	neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

#### Unofficial Draft for Comment Only

 $^{31}$ 

Suppose that the communicator **comm** has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
4
     ! assume: each process has stored a real number A.
5
        extract neighborhood information
6
     !
            CALL MPI_COMM_RANK(comm, myrank, ierr)
7
            CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
8
9
     ! perform exchange permutation
            CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0, &
10
11
                 neighbors(1), 0, comm, status, ierr)
     ! perform shuffle permutation
12
            CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, &
13
                 neighbors(3), 0, comm, status, ierr)
14
     ! perform unshuffle permutation
15
16
            CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0, &
17
                 neighbors(2), 0, comm, status, ierr)
18
         MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS pro-
19
     vide adjacency information for a distributed graph topology.
20
21
22
     MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted)
23
       IN
                                           communicator with distributed graph topology (han-
                comm
^{24}
                                           dle)
25
26
       OUT
                indegree
                                           number of edges into this process (non-negative inte-
27
                                           ger)
28
       OUT
                outdegree
                                           number of edges out of this process (non-negative in-
29
                                           teger)
30
       OUT
                weighted
                                           false if MPI_UNWEIGHTED was supplied during cre-
^{31}
                                           ation, true otherwise (logical)
32
33
34
     int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
35
                    int *outdegree, int *weighted)
36
     MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         INTEGER, INTENT(OUT) :: indegree, outdegree
39
         LOGICAL, INTENT(OUT) :: weighted
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
43
         INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
44
         LOGICAL WEIGHTED
45
46
47
48
```

1

 $\mathbf{2}$ 

MPI_DIST	GRAPH_NEIGHBORS(comm	, maxindegree, sources, sourceweights, maxoutdegree, )	1 2
IN	comm	communicator with distributed graph topology (han- dle)	3 4
IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)	5 6 7
OUT	sources	processes for which the calling process is a destination (array of non-negative integers)	8 9
OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)	10 11 12
IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)	12 13 14
OUT	destinations	processes for which the calling process is a source (array of non-negative integers)	15 16
OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)	17 18 19
int MPI_I		<pre>Comm comm, int maxindegree, int sources[],     int maxoutdegree, int destinations[],</pre>	20 21 22 23
TYPE INTE INTE INTE		degree, maxoutdegree ces(maxindegree), ) destweights(*)	24 25 26 27 28 29 30 31 32
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR			
These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (poten- tially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for			

43sourceweights or destweights or both, or if MPI\_UNWEIGHTED was supplied during the con-44struction of the graph then no weight information is returned in that array or those arrays. If the communicator was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT then for 4546each rank in comm, the order of the values in sources and destinations is identical to the in-47put that was used by the process with the same rank in comm\_old in the creation call. If the 48communicator 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 in put argument comm will return the same sequence of edges. If maxindegree or maxoutdegree
 is smaller than the numbers returned by MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT, then
 only the first part of the full list is returned.

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI\_DIST\_GRAPH\_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (End of advice to implementors.)

## 7.5.6 Cartesian Shift Coordinates

<sup>13</sup> 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.

```
21
22
```

23

5

6

7

8

9

10 11 12

MPI\_CART\_SHIFT(comm, direction, disp, rank\_source, rank\_dest)

23				
24	IN	comm	communicator with Cartesian structure (handle)	
25	IN	direction	coordinate dimension of shift (integer)	
26 27 28	IN	disp	displacement (> 0: upwards shift, < 0: downwards shift) (integer)	
29	OUT	rank_source	rank of source process (integer)	
30	OUT	rank_dest	rank of destination process (integer)	
31 32 33 34	int MPI_Ca	art_shift(MPI_Comm comm, int *rank_source, in	<pre>int direction, int disp, it *rank_dest)</pre>	
34 35 36 37 38	MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: direction, disp INTECEPINTENT(OUT) :: rank sourcerank_dest			
39 40		ER, OPTIONAL, INTENT(OUT SHIFT(COMM, DIRECTION, D	) :: ierror ISP, RANK_SOURCE, RANK_DEST, IERROR)	
41 42	INTEG	ER COMM, DIRECTION, DISP	, RANK_SOURCE, RANK_DEST, IERROR	
43 44		0	e coordinate dimension to be traversed by the shift. ndims-1, where ndims is the number of dimensions.	
45 46	-	· ·	e Cartesian group in the specified coordinate direc- entifiers for a circular or an end-off shift. In the case	
47 48	of an end-o	ff shift, the value $MPI_PROC_$	NULL may be returned in rank_source or rank_dest, ation for the shift is out of range.	
	0		<u> </u>	

It is erroneous to call MPI\_CART\_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI\_CART\_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.

#### Example 7.7

The communicator, **comm**, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of **REALs** is stored one element per process, in variable **A**. One wishes to skew this array, by shifting column i (vertically, i.e., along the column) by i steps.

! find process rank
CALL MPI_COMM_RANK(comm, rank, ierr)
! find Cartesian coordinates
CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
! compute shift source and destination
CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
! skew array
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
status, ierr)

Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1) nodes, where DIMS is the array that was used to create the grid. In C, the dimension indicated by direction = i is the dimension specified by dims[i]. (*End of advice to users.*)

## 7.5.7 Partitioning of Cartesian Structures

#### MPI\_CART\_SUB(comm, remain\_dims, newcomm)

IN	comm	communicator with Cartesian structure (handle)	31	
IN	remain_dims	the i-th entry of remain_dims specifies whether the	32	
		i-th dimension is kept in the subgrid (true) or is drop-	33 34	
		ped (false) (logical vector)	34	
OUT	newcomm	communicator containing the subgrid that includes	36	
		the calling process (handle)	37	
			38	
int MPI_Ca	int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm) 39			
MDT Cort	MPI_Cart_sub(comm, remain_dims, newcomm, ierror) 40			
	API_Comm), INTENT(IN) ::		41	
	AL, INTENT(IN) :: remain		42	
	<pre>/PI_Comm), INTENT(OUT) :</pre>		43	
	ER, OPTIONAL, INTENT(OUT)		44	
INIEGI	in, of Honke, INTENT(001)		45	
MPI_CART_S	SUB(COMM, REMAIN_DIMS, NI	EWCOMM, IERROR)	46	
INTEG	ER COMM, NEWCOMM, IERROR		47	
LOGIC	AL REMAIN_DIMS(*)		48	

 $\overline{7}$ 

 $^{24}$ 

1 If a Cartesian topology has been created with MPI\_CART\_CREATE, the function  $\mathbf{2}$ MPI\_CART\_SUB can be used to partition the communicator group into subgroups that 3 form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator 4 with the associated subgrid Cartesian topology. If all entries in remain\_dims are false or  $\mathbf{5}$ comm is already associated with a zero-dimensional Cartesian topology then newcomm is 6 associated with a zero-dimensional Cartesian topology. (This function is closely related to 7MPI\_COMM\_SPLIT.) 8 Example 7.8 9 Assume that MPI\_CART\_CREATE(..., comm) has defined a  $(2 \times 3 \times 4)$  grid. Let 10 remain\_dims = (true, false, true). Then a call to 11 12MPI\_CART\_SUB(comm, remain\_dims, comm\_new); 13 will create three communicators each with eight processes in a  $2 \times 4$  Cartesian topology. 14If remain\_dims = (false, false, true) then the call to  $MPI_CART_SUB(comm, remain_dims,$ 15comm\_new) will create six non-overlapping communicators, each with four processes, in a 1617one-dimensional Cartesian topology. 18 197.5.8 Low-Level Topology Functions 20The two additional functions introduced in this section can be used to implement all other 21topology functions. In general they will not be called by the user directly, unless he or she 22 is creating additional virtual topology capability other than that provided by MPI. The two 23calls are both local. 242526MPI\_CART\_MAP(comm, ndims, dims, periods, newrank) 27IN comm input communicator (handle) 28IN number of dimensions of Cartesian structure (integer) ndims 2930 IN dims integer array of size ndims specifying the number of  $^{31}$ processes in each coordinate direction 32 IN periods logical array of size ndims specifying the periodicity 33 specification in each coordinate direction 34 OUT newrank reordered rank of the calling process; 35 MPI\_UNDEFINED if calling process does not belong 36 to grid (integer) 37 38 39 int MPI\_Cart\_map(MPI\_Comm comm, int ndims, const int dims[], 40const int periods[], int \*newrank) 41 MPI\_Cart\_map(comm, ndims, dims, periods, newrank, ierror) 42TYPE(MPI\_Comm), INTENT(IN) :: comm 43 INTEGER, INTENT(IN) :: ndims, dims(ndims) 44 LOGICAL, INTENT(IN) :: periods(ndims) 45INTEGER, INTENT(OUT) :: newrank 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748MPI\_CART\_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)

INTEGER COMM, NDIMS, DIMS(\*), NEWRANK, IERROR 1  $\mathbf{2}$ LOGICAL PERIODS(\*) 3 MPI\_CART\_MAP computes an "optimal" placement for the calling process on the phys-4 ical machine. A possible implementation of this function is to always return the rank of the 5calling process, that is, not to perform any reordering. 6 7 The function MPI\_CART\_CREATE(comm, ndims, dims, Advice to implementors. 8 periods, reorder, comm\_cart), with reorder = true can be implemented by calling 9 MPI\_CART\_MAP(comm, ndims, dims, periods, newrank), then calling 10 MPI\_COMM\_SPLIT(comm, color, key, comm\_cart), with color = 0 if newrank  $\neq$ 11 MPI\_UNDEFINED, color = MPI\_UNDEFINED otherwise, and key = newrank. If ndims 12is zero then a zero-dimensional Cartesian topology is created. 13 The function MPI\_CART\_SUB(comm, remain\_dims, comm\_new) can be implemented 14 by a call to MPI\_COMM\_SPLIT(comm, color, key, comm\_new), using a single number 15encoding of the lost dimensions as color and a single number encoding of the preserved 16 dimensions as key. 17 18 All other Cartesian topology functions can be implemented locally, using the topology 19information that is cached with the communicator. (End of advice to implementors.) 20The corresponding function for graph structures is as follows. 2122 23MPI\_GRAPH\_MAP(comm, nnodes, index, edges, newrank)  $^{24}$ 25IN comm input communicator (handle) 26IN nnodes number of graph nodes (integer) 27IN index integer array specifying the graph structure, see 28 MPI\_GRAPH\_CREATE 29 30 IN edges integer array specifying the graph structure 31OUT newrank reordered rank of the calling process; 32 MPI\_UNDEFINED if the calling process does not be-33 long to graph (integer) 34 35 int MPI\_Graph\_map(MPI\_Comm comm, int nnodes, const int index[], 36 const int edges[], int \*newrank) 37 38 MPI\_Graph\_map(comm, nnodes, index, edges, newrank, ierror) 39 TYPE(MPI\_Comm), INTENT(IN) :: comm 40 INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(\*) 41 INTEGER, INTENT(OUT) :: newrank 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 MPI\_GRAPH\_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 44INTEGER COMM, NNODES, INDEX(\*), EDGES(\*), NEWRANK, IERROR 4546Advice to implementors. The function MPI\_GRAPH\_CREATE(comm, nnodes, index, 47

edges, reorder, comm\_graph), with reorder = true can be implemented by calling

MPI\_GRAPH\_MAP(comm, nnodes, index, edges, newrank), then calling MPI\_COMM\_SPLIT(comm, color, key, comm\_graph), with color = 0 if newrank  $\neq$ MPI\_UNDEFINED, color = MPI\_UNDEFINED otherwise, and key = newrank.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

# 7.6 Neighborhood Collective Communication on Process Topologies

<sup>10</sup> MPI process topologies specify a communication graph, but they implement no commu-<sup>11</sup> nication function themselves. Many applications require sparse nearest neighbor commu-<sup>12</sup> nications that can be expressed as graph topologies. We now describe several collective <sup>13</sup> operations that perform communication along the edges of a process topology. All of these <sup>14</sup> functions are collective; i.e., they must be called by all processes in the specified commu-<sup>15</sup> nicator. See Section 5 for an overview of other dense (global) collective communication <sup>16</sup> 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 [35]. This functionality can significantly simplify the implementation of neighbor exchanges [31]. (End of rationale.)

 $^{31}$ For a distributed graph topology, created with MPI\_DIST\_GRAPH\_CREATE, the se-32 quence of neighbors in the send and receive buffers at each process is defined as the sequence 33 returned by MPI\_DIST\_GRAPH\_NEIGHBORS for destinations and sources, respectively. For 34a general graph topology, created with MPI\_GRAPH\_CREATE, the use of neighborhood col-35 lective communication is restricted to adjacency matrices, where the number of edges be-36 tween any two processes is defined to be the same for both processes (i.e., with a symmetric 37 adjacency matrix). In this case, the order of neighbors in the send and receive buffers is 38 defined as the sequence of neighbors as returned by MPI\_GRAPH\_NEIGHBORS. Note that 39 general graph topologies should generally be replaced by the distributed graph topologies.

<sup>40</sup> For a Cartesian topology, created with MPI\_CART\_CREATE, the sequence of neigh-<sup>41</sup> bors in the send and receive buffers at each process is defined by order of the dimensions, <sup>42</sup> first the neighbor in the negative direction and then in the positive direction with dis-<sup>43</sup> placement 1. The numbers of sources and destinations in the communication routines are <sup>44</sup> **2\*ndims** with ndims defined in MPI\_CART\_CREATE. If a neighbor does not exist, i.e., at <sup>45</sup> the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., <sup>46</sup> periods[...]==false), then this neighbor is defined to be MPI\_PROC\_NULL.

<sup>47</sup> If a neighbor in any of the functions is MPI\_PROC\_NULL, then the neighborhood collec-<sup>48</sup> tive communication behaves like a point-to-point communication with MPI\_PROC\_NULL in

## Unofficial Draft for Comment Only

1

2

3

4

5

6 7

8 9

 $^{24}$ 

25

26

27

28

29

this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

#### Neighborhood Gather 7.6.1

In this function, each process i gathers data items from each process j if an edge (j, i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i, j) exists. The send buffer is sent to each neighboring process and the *l*-th block in the receive buffer is received from the *l*-th neighbor.

MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,			11
	comm)		12
IN	sendbuf	starting address of and huffer (shaise)	13
IIN		starting address of send buffer (choice)	14
IN	sendcount	number of elements sent to each neighbor (non-negative	15
		integer)	16
IN	sendtype	data type of send buffer elements (handle)	17 18
OUT	recvbuf	starting address of receive buffer (choice)	19
IN	recvcount	number of elements received from each neighbor (non-	20
		negative integer)	21
IN	recvtype	data type of receive buffer elements (handle)	22
		·-	23
IN	comm	communicator with topology structure (handle)	24
			25
int MPI_N	0 0	roid* sendbuf, int sendcount,	26
	01 01	e, void* recvbuf, int recvcount,	27
	MPI_Datatype recvtype	e, MPI_Comm comm)	28
MPT Neigh	bor allgather(sendbuf, se	endcount, sendtype, recvbuf, recvcount,	29
	recvtype, comm, ierro		30
TYPE (	*), DIMENSION(), INTENT		31
	*), DIMENSION() :: re		32
	ER, INTENT(IN) :: sendco		33
	MPI_Datatype), INTENT(IN)		34
	MPI_Comm), INTENT(IN) ::		35
	ER, OPTIONAL, INTENT(OUT)		36
	INTEGER, OFFICIARE, INTENT(COT) TETTOT		
MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 38			

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

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

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

1

 $\mathbf{2}$ 

3 4

5

6

 $\overline{7}$ 

8

9 10 1

39

40

41 42

43

44

45

4647

```
1
      int *srcs=(int*)malloc(indegree*sizeof(int));
\mathbf{2}
      int *dsts=(int*)malloc(outdegree*sizeof(int));
3
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
4
                                    outdegree,dsts,MPI_UNWEIGHTED);
5
      int k,l;
6
7
      /* assume sendbuf and recvbuf are of type (char*) */
8
      for(k=0; k<outdegree; ++k)</pre>
9
        MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
10
11
      for(l=0; l<indegree; ++1)</pre>
12
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
13
                    srcs[1]...);
14
15
     MPI_Waitall(...);
16
          Figure 7.1 shows the neighborhood gather communication of one process with outgoing
17
      neighbors d_0 \ldots d_3 and incoming neighbors s_0 \ldots s_5. The process will send its sendbuf to
18
      all four destinations (outgoing neighbors) and it will receive the contribution from all six
19
      sources (incoming neighbors) into separate locations of its receive buffer.
20
21
                                              d_0
22
                                                              d_2, s_4
23
                                           s_0
24
25
                               d_1
                                                          s_1
26
27
28
                                                                  s_3
                                           s_2
29
30
                                                      d_{3}, s_{5}
^{31}
                       sendbuf
32
33
34
                                                                   s_4
                                                                          s_5
                                    s_0
                                           s_1
                                                   s_2
                                                           s_3
35
                       recvbuf
36
37
                      Figure 7.1: Neighborhood gather communication example.
38
39
          All arguments are significant on all processes and the argument comm must have iden-
40
      tical values on all processes.
41
          The type signature associated with sendcount, sendtype, at a process must be equal to
42
      the type signature associated with recvcount, recvtype at all other processes. This implies
43
      that the amount of data sent must be equal to the amount of data received, pairwise between
44
      every pair of communicating processes. Distinct type maps between sender and receiver are
45
      still allowed.
46
47
                        For optimization reasons, the same type signature is required indepen-
            Rationale.
```

### Unofficial Draft for Comment Only

dently of whether the topology graph is connected or not. (*End of rationale.*)

## 7.6. NEIGHBORHOOD COLLECTIVE COMMUNICATION

1 The "in place" option is not meaningful for this operation. The vector variant of MPI\_NEIGHBOR\_ALLGATHER allows one to gather different 2 3 numbers of elements from each neighbor. 4 5MPI\_NEIGHBOR\_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 6 recvtype, comm) 7 8 IN sendbuf starting address of send buffer (choice) 9 IN sendcount number of elements sent to each neighbor (non-negative 10 integer) 11 IN sendtype data type of send buffer elements (handle) 1213 OUT recvbuf starting address of receive buffer (choice) 14IN recvcounts non-negative integer array (of length indegree) con-15taining the number of elements that are received from 16 each neighbor 17IN displs integer array (of length indegree). Entry i specifies the 18 displacement (relative to recvbuf) at which to place the 19 incoming data from neighbor i 2021IN data type of receive buffer elements (handle) recvtype 22 IN communicator with topology structure (handle) comm 23 $^{24}$ int MPI\_Neighbor\_allgatherv(const void\* sendbuf, int sendcount, 25MPI\_Datatype sendtype, void\* recvbuf, const int recvcounts[], 26const int displs[], MPI\_Datatype recvtype, MPI\_Comm comm) 2728 MPI\_Neighbor\_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 29 displs, recvtype, comm, ierror) 30 TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 31TYPE(\*), DIMENSION(..) :: recvbuf 32 INTEGER, INTENT(IN) :: sendcount, recvcounts(\*), displs(\*) 33 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 34 TYPE(MPI\_Comm), INTENT(IN) :: comm 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 MPI\_NEIGHBOR\_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 37 DISPLS, RECVTYPE, COMM, IERROR) 38 <type> SENDBUF(\*), RECVBUF(\*) 39 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 40 IERROR 41

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

MPI\_Dist\_graph\_neighbors\_count(comm,&indegree,&outdegree,&weighted); int \*srcs=(int\*)malloc(indegree\*sizeof(int));

### Unofficial Draft for Comment Only

42

43

44

45

 $46 \\ 47$ 

```
1
     int *dsts=(int*)malloc(outdegree*sizeof(int));
\mathbf{2}
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
3
                                 outdegree,dsts,MPI_UNWEIGHTED);
4
     int k,l;
5
6
     /* assume sendbuf and recvbuf are of type (char*) */
7
     for(k=0; k<outdegree; ++k)</pre>
8
       MPI_Isend(sendbuf,sendcount,sendtype,dsts[k],...);
9
10
     for(l=0; l<indegree; ++l)</pre>
11
       MPI_Irecv(recvbuf+displs[l]*extent(recvtype), recvcounts[l], recvtype,
12
                  srcs[1],...);
13
14
     MPI_Waitall(...);
```

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[I], recvtype at any other process with srcs[I]==j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data received from the I-th neighbor is placed into recvbuf beginning at offset displs[I] elements (in terms of the recvtype).

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

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

25 26

27

22

15

## 7.6.2 Neighbor Alltoall

In this function, each process *i* receives data items from each process *j* if an edge (j, i)exists in the topology graph or Cartesian topology. Similarly, each process *i* sends data items to all processes *j* where an edge (i, j) exists. This call is more general than MPI\_NEIGHBOR\_ALLGATHER in that different data items can be sent to each neighbor. The *h* the block in each buffer is cent to the *h* the price process and the *l* the block in

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.

- 46
- 47 48

MPI_NE	IGHBOR_ALLTOALL(sen comm)	ndbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,	1 $2$		
IN	sendbuf	starting address of send buffer (choice)	3		
IN	sendcount	number of elements sent to each neighbor (non-negative	4 5		
	Sendoount	integer)	6		
IN	sendtype	data type of send buffer elements (handle)	7		
OUT	recvbuf	starting address of receive buffer (choice)	8		
IN	recvcount	number of elements received from each neighbor (non-negative integer)	9 10 11		
IN	recvtype	data type of receive buffer elements (handle)	12		
IN	comm	communicator with topology structure (handle)	13 14		
int MPT	Neighbor alltoall(c	onst void* sendbuf, int sendcount,	15		
1110 111 1	•	sendtype, void* recvbuf, int recvcount,	16 17		
	• -	recvtype, MPI_Comm comm)	18		
MPI Nei	.ghbor alltoall(sendb	uf, sendcount, sendtype, recvbuf, recvcount,	19		
_	recvtype, comm	• •	20		
		INTENT(IN) :: sendbuf	21		
	<pre>PE(*), DIMENSION()</pre>		22		
		sendcount, recvcount	23 24		
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype2TYPE(MPI_Comm), INTENT(IN) :: comm2				
	EGER, OPTIONAL, INTE		26		
			27		
MPI_NEI		UF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	28		
< +	RECVTYPE, COMM ype> SENDBUF(*), REC		29		
•		TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	30		
			31 32		
		esian communicators, graph communicators, and distributed	33		
•		ed in Section 7.6. If comm is a distributed graph commu-	34		
	eives from each of its inco	h process executed sends to each of its outgoing neighbors	35		
and rece	aves nom each of its mee	Jinnig neighbors.	36		
MPI_Dis	t_graph_neighbors_co	unt(comm,&indegree,&outdegree,&weighted);	37		
	cs=(int*)malloc(inde	•	38		
	ts=(int*)malloc(outd	0	39 40		
MPI_Dis	<pre>MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,</pre>				
		<pre>tdegree,dsts,MPI_UNWEIGHTED);</pre>	41 42		
int k,l	;		43		
/* assu	me sendbuf and recyb	uf are of type (char*) */	44		
	; k <outdegree; ++k)<="" td=""><td></td><td>45</td></outdegree;>		45		
	-	ount*extent(sendtype),sendcount,sendtype,	46		
	dsts[k],);	· - · · ·	47		
			48		

```
1
      for(l=0; l<indegree; ++1)</pre>
\mathbf{2}
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
3
                    srcs[1],...);
4
5
     MPI_Waitall(...);
6
          The type signature associated with sendcount, sendtype, at a process must be equal to
7
      the type signature associated with recvcount, recvtype at any other process. This implies
8
      that the amount of data sent must be equal to the amount of data received, pairwise between
9
      every pair of communicating processes. Distinct type maps between sender and receiver are
10
     still allowed.
11
          The "in place" option is not meaningful for this operation.
12
          All arguments are significant on all processes and the argument comm must have iden-
13
     tical values on all processes.
14
          The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
15
      numbers of elements to and from each neighbor.
16
17
18
      MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
19
                      rdispls, recvtype, comm)
20
        IN
                  sendbuf
                                                starting address of send buffer (choice)
21
22
        IN
                  sendcounts
                                                non-negative integer array (of length outdegree) speci-
23
                                                fying the number of elements to send to each neighbor
^{24}
        IN
                  sdispls
                                                integer array (of length outdegree). Entry j specifies
25
                                                the displacement (relative to sendbuf) from which to
26
                                                send the outgoing data to neighbor j
27
        IN
                  sendtype
                                                data type of send buffer elements (handle)
28
29
        OUT
                  recvbuf
                                                starting address of receive buffer (choice)
30
        IN
                  recvcounts
                                                non-negative integer array (of length indegree) speci-
^{31}
                                                fying the number of elements that are received from
32
                                                each neighbor
33
        IN
                  rdispls
                                                integer array (of length indegree). Entry i specifies the
34
                                                displacement (relative to recvbuf) at which to place the
35
                                                incoming data from neighbor i
36
37
        IN
                                                data type of receive buffer elements (handle)
                  recvtype
38
        IN
                  comm
                                                communicator with topology structure (handle)
39
40
      int MPI_Neighbor_alltoallv(const void* sendbuf, const int sendcounts[],
41
                      const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
42
                      const int recvcounts[], const int rdispls[],
43
                      MPI_Datatype recvtype, MPI_Comm comm)
44
45
     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
46
                      recvcounts, rdispls, recvtype, comm, ierror)
47
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
48
          TYPE(*), DIMENSION(..) :: recvbuf
```

```
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                 2
        rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::
                                                                                 4
                                   comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
             RECVCOUNTS, RDISPLS,
    RECVTYPE, COMM, IERROR)
                                                                                 9
    <type> SENDBUF(*), RECVBUF(*)
                                                                                 10
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                 11
    RECVTYPE, COMM, IERROR
                                                                                 12
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

<pre>MPI_Dist_graph_neighbors_count(comm,&amp;indegree,&amp;outdegree,&amp;weighted);</pre>		
<pre>int *srcs=(int*)malloc(indegree*sizeof(int));</pre>		
<pre>int *dsts=(int*)malloc(outdegree*sizeof(int));</pre>		
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,		
<pre>outdegree,dsts,MPI_UNWEIGHTED);</pre>		
int k,l;		

```
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)</pre>
```

```
MPI_Isend(sendbuf+sdispls[k]*extent(sendtype),sendcounts[k],sendtype,
          dsts[k],...);
```

```
for(l=0; l<indegree; ++1)</pre>
 MPI_Irecv(recvbuf+rdispls[1]*extent(recvtype),recvcounts[1],recvtype,
            srcs[1],...);
```

```
MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process 36 i must be equal to the type signature associated with recvcounts[I], recvtype with srcs[I] = = i37 at process j. This implies that the amount of data sent must be equal to the amount of 38 data received, pairwise between every pair of communicating processes. Distinct type maps 39 between sender and receiver are still allowed. The data in the sendbuf beginning at offset sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data 41 received from the I-th incoming neighbor is placed into recvbuf beginning at offset rdispls[I] 42elements (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.

MPI\_NEIGHBOR\_ALLTOALLW allows one to send and receive with different datatypes to and from each neighbor.

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

1

3

5 6

7

8

13

14

15

16

24

2526

27

2829

30

31

32 33 34

35

40

43

44

45

46

47

	rdispls, recvtypes, comm	, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, )
IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)
IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)
IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)
IN	comm	communicator with topology structure (handle)
int MPI_1	const MPI_Aint sdis void* recvbuf, cons	<pre>void* sendbuf, const int sendcounts[], pls[], const MPI_Datatype sendtypes[], t int recvcounts[], pls[], const MPI_Datatype recvtypes[],</pre>
<pre>MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,</pre>		
	IN IN IN IN OUT IN IN IN IN IN IN IN IN IN IN IN IN IN	rdispls, recvtypes, comm IN sendbuf IN sendcounts IN sdispls IN sdispls IN sendtypes OUT recvbuf IN recvcounts IN rdispls IN rdispls IN recvtypes IN comm int MPI_Neighbor_alltoallw(const const MPI_Aint sdisp void* recvbuf, const const MPI_Aint rdisp WPI_Comm comm) MPI_Neighbor_alltoallw(sendbuf, s recvcounts, rdispls TYPE(*), DIMENSION(), INTEN TYPE(*), DIMENSION(), INTEN TYPE(*), DIMENSION() :: r INTEGER, INTENT(IN) :: sendod INTEGER, INTENT(IN) :: sendod INTEGER, OPTIONAL, INTENT(OUT MPI_NEIGHBOR_ALLTOALLW(SENDBUF, S RECVCOUNTS, RDISPLS <type> SENDBUF(*), RECVBUF(*)</type>

#### IERROR

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm,&indegree,&outdegree,&weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm,indegree,srcs,MPI_UNWEIGHTED,
outdegree,dsts,MPI_UNWEIGHTED);
```

int k,l;

```
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
    MPI_Isend(sendbuf+sdispls[k],sendcounts[k], sendtypes[k],dsts[k],...);</pre>
```

```
for(1=0; 1<indegree; ++1)
    MPI_Irecv(recvbuf+rdispls[1],recvcounts[1], recvtypes[1],srcs[1],...);</pre>
```

```
MPI_Waitall(...);
```

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

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

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

## 7.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 5.12. 1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13 14

15

16

17 18

19

20 21

22 23

 $^{24}$ 

25

26

27

28

29

30

31 32

33 34

35

36

	326		CHAPTER 7. PROCESS TOPOLOGIES	
1 2 3	7.7.1 No	nblocking Neighborhood Gatl	her	
4 5	MPI_INEIG	GHBOR_ALLGATHER(sendbuf, comm, request)	sendcount, sendtype, recvbuf, recvcount, recvtype,	
6 7	IN	sendbuf	starting address of send buffer (choice)	
8 9	IN	sendcount	number of elements sent to each neighbor (non-negative integer)	
10	IN	sendtype	data type of send buffer elements (handle)	
11 12	OUT	recvbuf	starting address of receive buffer (choice)	
13 14	IN	recvcount	number of elements received from each neighbor (non-negative integer)	
15	IN	recvtype	data type of receive buffer elements (handle)	
16 17	IN	comm	communicator with topology structure (handle)	
18	OUT	request	communication request (handle)	
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34	<pre>int MPI_Ineighbor_allgather(const void* sendbuf, int sendcount,</pre>			
35 36 37 38 39 40 41 42 43 44 45 46 47 48	INTEG		AECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	

MPI\_INEIGHBOR\_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 1  $\mathbf{2}$ recvtype, comm, request) 3 IN sendbuf starting address of send buffer (choice) 4 IN sendcount number of elements sent to each neighbor (non-negative 5integer) 6 7 IN sendtype data type of send buffer elements (handle) 8 OUT recvbuf starting address of receive buffer (choice) 9 non-negative integer array (of length indegree) con-IN recvcounts 10 taining the number of elements that are received from 11 each neighbor 12IN 13 displs integer array (of length indegree). Entry i specifies the 14displacement (relative to recvbuf) at which to place the 15incoming data from neighbor i 16IN recvtype data type of receive buffer elements (handle) 17IN communicator with topology structure (handle) comm 18 19 OUT communication request (handle) request 2021int MPI\_Ineighbor\_allgatherv(const void\* sendbuf, int sendcount, 22 MPI\_Datatype sendtype, void\* recvbuf, const int recvcounts[], 23const int displs[], MPI\_Datatype recvtype, MPI\_Comm comm, 24MPI\_Request \*request) 25MPI\_Ineighbor\_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 26displs, recvtype, comm, request, ierror) 27TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 28TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 29 INTEGER, INTENT(IN) :: sendcount 30 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(\*), displs(\*) 31TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 32 TYPE(MPI\_Comm), INTENT(IN) :: comm 33 TYPE(MPI\_Request), INTENT(OUT) :: request 34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3536 MPI\_INEIGHBOR\_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 37 DISPLS, RECVTYPE, COMM, REQUEST, IERROR) 38 <type> SENDBUF(\*), RECVBUF(\*) 39 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 40 REQUEST, IERROR 41 This call starts a nonblocking variant of MPI\_NEIGHBOR\_ALLGATHERV. 4243 44 4546

	328		CHAPTER 7. PROCESS TOPOLOGIES
1 2 3	7.7.2 Nonblocking Neighborhood Alltoall		
4 5	MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)		
6 7	IN	sendbuf	starting address of send buffer (choice)
8 9	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
10	IN	sendtype	data type of send buffer elements (handle)
11 12	OUT	recvbuf	starting address of receive buffer (choice)
13 14	IN	recvcount	number of elements received from each neighbor (non-negative integer)
15	IN	recvtype	data type of receive buffer elements (handle)
16 17	IN	comm	communicator with topology structure (handle)
18	OUT	request	communication request (handle)
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	<pre>int MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount,</pre>		
44 45 46 47 48			

	rdispls, recvtype,	sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, comm, request)	
IN	sendbuf	starting address of send buffer (choice)	
IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor	
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to <b>sendbuf</b> ) from which send the outgoing data to neighbor j	
IN	sendtype	data type of send buffer elements (handle)	1
OUT	recvbuf	starting address of receive buffer (choice)	
IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor	
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	
IN	recvtype	data type of receive buffer elements (handle)	:
IN	comm	communicator with topology structure (handle)	
OUT	request	communication request (handle)	
MPI_Inei TYPE TYPE INTE TYPE TYPE TYPE	const int sdis const int recy MPI_Datatype n recvcounts, re E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN), AS recvcounts(*), rdis E(MPI_Datatype), INT E(MPI_Comm), INTENT(	YNCHRONOUS :: sendcounts(*), sdispls(*), ppls(*) ENT(IN) :: sendtype, recvtype IN) :: comm ENT(OUT) :: request	
<pre>MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, REQUEST, IERROR This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.</type></pre>			

1 2	MPI_INEI	GHBOR_ALLTOALLW(ser rdispls, recvtypes, co	ndbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, cmm, request)		
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)		
4 5 6	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor		
7 8 9 10	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)		
11 12 13 14	IN	sendtypes	array of datatypes (of length outdegree). Entry j spec- ifies the type of data to send to neighbor j (array of handles)		
15	OUT	recvbuf	starting address of receive buffer (choice)		
16 17 18	IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor		
19 20 21 22 23	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)		
24 25 26	IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)		
27	IN	comm	communicator with topology structure (handle)		
28 29	OUT	request	communication request (handle)		
30 31 32 33 34 35	<pre>int MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[], const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void* recvbuf, const int recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm_comm_MPI_Bequest *request)</pre>				
36	MPI_Inei	ghbor_alltoallw(sendb	uf, sendcounts, sdispls, sendtypes, recvbuf,		
37 38			pls, recvtypes, comm, request, ierror)		
39			NTENT(IN), ASYNCHRONOUS :: sendbuf SYNCHRONOUS :: recvbuf		
40			CHRONOUS :: sendcounts(*), recvcounts(*)		
41			KIND), INTENT(IN), ASYNCHRONOUS ::		
42		<pre>sdispls(*), rdispls(*</pre>			
$43 \\ 44$		• -	T(IN), ASYNCHRONOUS :: sendtypes(*),		
45		<pre>recvtypes(*) (MPI_Comm), INTENT(IN</pre>	):: comm		
46		(MPI_Request), INTENT			
47		GER, OPTIONAL, INTENT	-		
48					

This call starts a nonblocking variant of MPI\_NEIGHBOR\_ALLTOALLW.

### 7.8 An Application Example

**Example 7.9** The example in Figures 7.2-7.4 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine **relax**.

In each relaxation step each process computes new values for the solution grid function at the points u(1:100,1:100) owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the newly calculated values in u(1,1:100) must be sent into the halo cells u(101,1:100) of the left-hand neighbor with coordinates (own\_coord(1)-1,own\_coord(2)).

```
1
2
3
4
5
6
7
8
     INTEGER ndims, num_neigh
9
     LOGICAL reorder
10
     PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
11
     INTEGER comm, comm_cart, dims(ndims), ierr
12
     INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
13
     LOGICAL periods(ndims)
14
     REAL u(0:101,0:101), f(0:101,0:101)
15
     DATA dims / ndims * 0 /
16
     comm = MPI_COMM_WORLD
17
     !
         Set process grid size and periodicity
18
     CALL MPI_DIMS_CREATE(comm, ndims, dims, ierr)
19
     periods(1) = .TRUE.
20
     periods(2) = .TRUE.
21
         Create a grid structure in WORLD group and inquire about own position
22
     CALL MPI_CART_CREATE (comm, ndims, dims, periods, reorder, &
23
                            comm_cart, ierr)
^{24}
     CALL MPI_CART_GET (comm_cart, ndims, dims, periods, own_coords, ierr)
25
     i = own_coords(1)
26
     j = own_coords(2)
27
     ! Look up the ranks for the neighbors. Own process coordinates are (i,j).
28
     ! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
     CALL MPI_CART_SHIFT (comm_cart, 0,1, neigh_rank(1),neigh_rank(2), ierr)
29
30
     CALL MPI_CART_SHIFT (comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
^{31}
     ! Initialize the grid functions and start the iteration
32
     CALL init (u, f)
33
     DO it=1,100
34
        CALL relax (u, f)
35
            Exchange data with neighbor processes
     !
36
        CALL exchange (u, comm_cart, neigh_rank, num_neigh)
37
     END DO
38
     CALL output (u)
39
40
        Figure 7.2: Set-up of process structure for two-dimensional parallel Poisson solver.
41
42
43
44
45
46
47
48
```

```
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
REAL u(0:101,0:101)
                                                                                   12
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                   13
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
                                                                                   14
INTEGER ierr
sndbuf(1:100,1) = u( 1,1:100)
sndbuf(1:100,2) = u(100,1:100)
sndbuf(1:100,3) = u(1:100, 1)
                                                                                   18
sndbuf(1:100,4) = u(1:100,100)
                                                                                   19
CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
                                                                                   20
                             comm_cart, ierr)
                                                                                   21
! instead of
                                                                                   22
! DO i=1,num_neigh
                                                                                   23
    CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1),&
!
                                                                                   24
!
                    ierr)
                                                                                   25
!
    CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i ),&
                                                                                   26
!
                    ierr)
                                                                                   27
! END DO
                                                                                   28
! CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)
                                                                                   29
                                                                                   30
u( 0,1:100) = rcvbuf(1:100,1)
u(101,1:100) = rcvbuf(1:100,2)
                                                                                   32
u(1:100, 0) = rcvbuf(1:100,3)
                                                                                   33
u(1:100,101) = rcvbuf(1:100,4)
                                                                                   34
END
                                                                                   35
                                                                                   36
```

Figure 7.3: Communication routine with local data copying and sparse neighborhood allto-all.

**Unofficial Draft for Comment Only** 

11

15

16

17

 $^{31}$ 

37

38

2 3 4 SUBROUTINE exchange (u, comm\_cart, neigh\_rank, num\_neigh) 5IMPLICIT NONE 6 USE MPI 7 REAL u(0:101,0:101) 8 INTEGER comm\_cart, num\_neigh, neigh\_rank(num\_neigh) 9 INTEGER sndcounts(num\_neigh), sndtypes(num\_neigh) 10 INTEGER rcvcounts(num\_neigh), rcvtypes(num\_neigh) 11 INTEGER (KIND=MPI\_ADDRESS\_KIND) lb, sizeofreal 12INTEGER (KIND=MPI\_ADDRESS\_KIND) sdispls(num\_neigh), rdispls(num\_neigh) 13 INTEGER type\_vec, ierr 14 ! The following initialization need to be done only once 15 ! before the first call of exchange. 16 CALL MPI\_TYPE\_GET\_EXTENT (MPI\_REAL, 1b, sizeofreal, ierr) 17CALL MPI\_TYPE\_VECTOR (100, 1, 102, MPI\_REAL, type\_vec, ierr) 18 CALL MPI\_TYPE\_COMMIT (type\_vec, ierr) 19 sndtypes(1:2) = type\_vec 20sndcounts(1:2) = 121sndtypes(3:4) = MPI\_REAL 22 sndcounts(3:4) = 10023 rcvtypes = sndtypes 24 rcvcounts = sndcounts 25sdispls(1) = ( 1 + 1\*102) \* sizeofreal ! first element of u( 1 , 1:100) 26, 1:100) sdispls(2) = (100 + 1\*102) \* size of real ! first element of u(100)27sdispls(3) = ( 1 + 1\*102) \* sizeofreal ! first element of u( 1:100, 1 ) 28sdispls(4) = (1 + 100\*102) \* sizeofreal ! first element of u( 1:100,100 ) 29 rdispls(1) = ( 0 + 1\*102) \* sizeofreal ! first element of u( 0 , 1:100) 30 rdispls(2) = (101 + 1\*102) \* size of real ! first element of u(101). 1:100) 31 rdispls(3) = (1 + 0\*102) \* size of real ! first element of u(1:100, 0)) 32 rdispls(4) = (1 + 101\*102) \* sizeofreal ! first element of u( 1:100,101 ) 33 ! the following communication has to be done in each call of exchange 34 CALL MPI\_NEIGHBOR\_ALLTOALLW (u, sndcounts, sdispls, sndtypes, & 35 u, rcvcounts, rdispls, rcvtypes, & 36 comm\_cart, ierr) 37 ! The following finalizing need to be done only once 38 ! after the last call of exchange. 39 CALL MPI\_TYPE\_FREE (type\_vec, ierr) 40END 41 4243 Figure 7.4: Communication routine with sparse neighborhood all-to-all-w and without local  $^{44}$ data copying. 4546 47 48

## Chapter 8

# **MPI** Environmental Management

> 6 7

9

15

16

17

1819

2021

22

23

 $^{24}$ 25

26

2728

29

30  $^{31}$ 

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.

#### Implementation Information 8.1

#### 8.1.1 Version Inquiries

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

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

#define MPI\_VERSION 3 #define MPI\_SUBVERSION 1

. Б .

in Fortran,		
	32	
INTEGER :: MPI_VERSION, MPI_SUBVERSION		
PARAMETER $(MPI_VERSION = 3)$	34	
PARAMETER (MPI_SUBVERSION = 1)	35	
	36	
For runtime determination,	37	
	38	
MPI_GET_VERSION( version, subversion )	39	
	40	
OUTversionversion number (integer)	41	
OUT subversion subversion number (integer)	42	
	43	
<pre>int MPI_Get_version(int *version, int *subversion)</pre>	44	
,,,,,,, _	45	
MPI_Get_version(version, subversion, ierror)	46	
INTEGER, INTENT(OUT) :: version, subversion	47	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48	

<b>Unofficial Draft for Comment</b>	Only	335
-------------------------------------	------	-----

	000		CHAI IER 8. WETENVIRONMENTAL MANAGEMENT
1 2		VERSION(VERSION, GER VERSION, SUBV	SUBVERSION, IERROR) VERSION, IERROR
3 4 5 6 7 8	function MPI_SUB	must always be thr	a be called before MPI_INIT and after MPI_FINALIZE. This read-safe, as defined in Section 12.4. Valid (MPI_VERSION, is and previous versions of the MPI standard are (3,1), (3,0),
9	MPI_GET	_LIBRARY_VERSIO	N( version, resultlen )
10 11	OUT	version	version string (string)
12 13 14	OUT	resultlen	Length (in printable characters) of the result returned in <b>version</b> (integer)
15	int MPI_	Get_library_versi	ion(char *version, int *resultlen)
16 17 18 19 20	CHAR INTE	U	
21 22 23	CHAR	LIBRARY_VERSION(\ ACTER*(*) VERSION GER RESULTLEN,IEF	
24 25 26			ring representing the version of the MPI library. The version g for maximum flexibility.
27 28 29 30	for e	-	. An implementation of MPI should return a different string source code or build that could be visible to the user. ( <i>End of</i> )
<ol> <li>31</li> <li>32</li> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> </ol>	MPI_MAX write up to The to In C, a nut be larger the right MPI_MAX MPI_	_LIBRARY_VERSION_ to this many character number of characters ill character is additi- than MPI_MAX_LIBF with blank character _LIBRARY_VERSION_ _GET_LIBRARY_VE	s actually written is returned in the output argument, resultlen. onally stored at version[resultlen]. The value of resultlen cannot RARY_VERSION_STRING - 1. In Fortran, version is padded on ers. The value of resultlen cannot be larger than
42 43	8.1.2 Er	nvironmental Inquiri	es
44 45 46 47 48	cator MP	LCOMM_WORLD wh by using the function	ibe the execution environment are attached to the communi- nen MPI is initialized. The values of these attributes can be on MPI_COMM_GET_ATTR described in Section 6.7 and in us to delete these attributes, free their keys, or change their

CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

The list of predefined attribute keys include	1
<b>MPI_TAG_UB</b> Upper bound for tag value.	2
	3
<b>MPI_HOST</b> Host process rank, if such exists, MPI_PROC_NULL, otherwise.	4 5
<b>MPI_IO</b> rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.	6 7
<b>MPI_WTIME_IS_GLOBAL</b> Boolean variable that indicates whether clocks are synchronized.	8 9
<b>MPI_FT</b> Boolean variable that indicates whether fault tolerance is supported.	$\frac{10}{11}$
Vendors may add implementation-specific parameters (such as node number, real mem- ory size, virtual memory size, etc.) These predefined attributes do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.	11 12 13 14 15
Advice to users. Note that in the C binding, the value returned by these attributes is a <i>pointer</i> to an <b>int</b> containing the requested value. ( <i>End of advice to users.</i> )	16 17 18
The required parameter values are discussed in more detail below:	19 20
Tag Values	21
Tag values range from 0 to the value returned for MPI_TAG_UB, inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be <i>at least</i> 32767. An MPI implementation is free to make the value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a valid value for MPI_TAG_UB. The attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.	22 23 24 25 26 27 28 20
Host Rank	29 30
The value returned for MPI_HOST gets the rank of the <i>HOST</i> process in the group associated with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if there is no host. MPI does not specify what it means for a process to be a <i>HOST</i> , nor does it requires that a <i>HOST</i> exists. The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.	31 32 33 34 35 36
IO Rank	37 38
The value returned for MPI_IO is the rank of a processor that can provide language-standard $I/O$ facilities. For Fortran, this means that all of the Fortran $I/O$ operations are supported	39 40

I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., OPEN, REWIND, WRITE). For C, this means that all of the ISO C I/O operations are supported (e.g., fopen, fprintf, lseek).

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

#### Unofficial Draft for Comment Only

 $^{41}$ 

```
1
           Advice to users. Note that input is not collective, and this attribute does not indicate
\mathbf{2}
           which process can or does provide input. (End of advice to users.)
3
4
      Clock Synchronization
5
      The value returned for MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in
6
      MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered
7
      synchronized if explicit effort has been taken to synchronize them. The expectation is that
8
      the variation in time, as measured by calls to MPI_WTIME, will be less then one half the
9
      round-trip time for an MPI message of length zero. If time is measured at a process just
10
      before a send and at another process just after a matching receive, the second time should
11
     be always higher than the first one.
12
          The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not
13
      synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This
14
      attribute may be associated with communicators other then MPI_COMM_WORLD.
15
          The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of
16
      MPI_COMM_WORLD.
17
18
      Fault Tolerance
19
20
     The value returned for MPI_FT is 1 if fault tolerance (as defined in Chapter 15) is supported,
21
     0 otherwise.
\overline{22}
          The attribute MPI_FT need not be present when fault tolerance is not supported (how-
23
      ever, the attribute key MPI_FT is always valid).
24
          The attribute MPI_FT has the same value on all processes of MPI_COMM_WORLD.
25
26
     Inquire Processor Name
27
28
29
      MPI_GET_PROCESSOR_NAME( name, resultlen )
30
^{31}
        OUT
                                               A unique specifier for the actual (as opposed to vir-
                  name
32
                                               tual) node.
33
        OUT
                  resultlen
                                               Length (in printable characters) of the result returned
34
                                               in name
35
36
      int MPI_Get_processor_name(char *name, int *resultlen)
37
38
     MPI_Get_processor_name(name, resultlen, ierror)
          CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
39
40
          INTEGER, INTENT(OUT) :: resultlen
41
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
      MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
43
          CHARACTER*(*) NAME
44
          INTEGER RESULTLEN, IERROR
45
46
          This routine returns the name of the processor on which it was called at the moment
47
      of the call. The name is a character string for maximum flexibility. From this value it
```

CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

338

<sup>48</sup> 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.*)

### 8.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of some RMA functionality as defined in Section 11.5.3.

#### MPI\_ALLOC\_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative inte-	32
		ger)	33
	in fa	(1, 1)	34
IN	info	info argument (handle)	35
OUT	baseptr	pointer to beginning of memory segment allocated	36
			37
int MPI_A	lloc_mem(MPI_Aint size, N	<pre>IPI_Info info, void *baseptr)</pre>	38
NDT 411			39
	_mem(size, info, baseptr,		40
USE,	INTRINSIC :: ISO_C_BINDI	NG, ONLY : C_PTR	41
INTEC	ER(KIND=MPI_ADDRESS_KIND)	, INTENT(IN) :: size	42
TYPE(	MPI_Info), INTENT(IN) ::	info	43
TYPE(	C_PTR), INTENT(OUT) :: h	paseptr	44
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	45
MPI_ALLOC	_MEM(SIZE, INFO, BASEPTR,	IERROR)	46
INTEG	ER INFO, IERROR		47
INTEG	ER(KIND=MPI_ADDRESS_KIND)	SIZE, BASEPTR	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

<sup>1</sup> If the Fortran compiler provides TYPE(C\_PTR), then the following generic interface must <sup>2</sup> be provided in the mpi module and should be provided in mpif.h through overloading, <sup>3</sup> i.e., with the same routine name as the routine with INTEGER(KIND=MPI\_ADDRESS\_KIND) <sup>4</sup> BASEPTR, but with a different specific procedure name:

```
INTERFACE MPI_ALLOC_MEM
6
         SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
\overline{7}
8
              IMPORT :: MPI_ADDRESS_KIND
              INTEGER INFO, IERROR
9
              INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
10
11
         END SUBROUTINE
         SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
12
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
13
              IMPORT :: MPI_ADDRESS_KIND
14
              INTEGER :: INFO, IERROR
15
              INTEGER(KIND=MPI_ADDRESS_KIND) ::
                                                      SIZE
16
              TYPE(C_PTR) :: BASEPTR
17
         END SUBROUTINE
18
19
     END INTERFACE
20
         The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The
21
     implied specific procedure names are described in Section 18.1.5.
22
         The info argument can be used to provide directives that control the desired location
23
     of the allocated memory. Such a directive does not affect the semantics of the call. Valid
^{24}
     info values are implementation-dependent; a null directive value of info = MPI_INFO_NULL
25
     is always valid.
26
         The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
27
     to indicate it failed because memory is exhausted.
28
29
30
     MPI_FREE_MEM(base)
^{31}
       IN
                 base
                                            initial address of memory segment allocated by
32
                                             MPI_ALLOC_MEM (choice)
33
34
35
     int MPI_Free_mem(void *base)
36
     MPI_Free_mem(base, ierror)
37
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: base
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_FREE_MEM(BASE, IERROR)
41
          <type> BASE(*)
42
         INTEGER IERROR
43
         The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to
44
     indicate an invalid base argument.
45
46
          Rationale. The C bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar
47
          to the bindings for the malloc and free C library calls: a call to
48
```

MPI\_Alloc\_mem(..., &base) should be paired with a call to MPI\_Free\_mem(base) (one less level of indirection). Both arguments are declared to be of same type void\* so as to facilitate type casting. The Fortran binding is consistent with the C bindings: the Fortran MPI\_ALLOC\_MEM call returns in baseptr the TYPE(C\_PTR) pointer or the (integer valued) address of the allocated memory. The base argument of MPI\_FREE\_MEM is a choice argument, which passes (a reference to) the variable 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 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 8.1** 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/)size = 4 \* shape(1) \* shape(2)! assuming 4 bytes per REAL CALL MPI\_Alloc\_mem(size, MPI\_INFO\_NULL, p, ierr) ! memory is allocated and CALL C\_F\_POINTER(p, a, shape) ! intrinsic ! now accessible via a(i,j) ! in ISO\_C\_BINDING . . . a(3,5) = 2.71;CALL MPI\_Free\_mem(a, ierr) ! memory is freed

**Example 8.2** Example of use of MPI\_ALLOC\_MEM, in Fortran with non-standard *Craypointers*. We assume 4-byte REALs, and assume that these pointers are address-sized.

```
REAL A
                                                                                     39
POINTER (P, A(100,100))
                             ! no memory is allocated
                                                                                     40
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
                                                                                     41
SIZE = 4*100*100
                                                                                     42
CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
                                                                                     43
! memory is allocated
                                                                                     44
. . .
                                                                                     45
A(3,5) = 2.71;
                                                                                     46
. . .
                                                                                     47
CALL MPI_FREE_MEM(A, IERR) ! memory is freed
                                                                                     48
```

1

2

3

4

5 6

7

8

9

10

11

12

13 14

15

16 17

18

19 20

21

22

23

24

25

26

27

28

29

30

31

32 33

34 35 36

37

This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this code or need a special option, e.g., the GNU gFortran compiler needs -fcray-pointer.

Advice to implementors. Some compilers map Cray-pointers to address-sized integers, some to TYPE(C\_PTR) pointers (e.g., Cray Fortran, version 7.3.3). From the user's viewpoint, this mapping is irrelevant because Examples 8.2 should work correctly with an MPI-3.0 (or later) library if Cray-pointers are available. (End of advice to *implementors.*)

10 **Example 8.3** Same example, in C. 11

```
float (* f)[100][100];
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
. . .
(*f)[5][3] = 2.71;
. . .
MPI_Free_mem(f);
```

```
Error Handling
8.3
```

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

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. 29

A user can associate error handlers to three types of objects: communicators, windows, 30 and files. The specified error handling routine will be used for any MPI exception that occurs  $^{31}$ during a call to MPI for the respective object. MPI calls that are not related to any objects 32 are considered to be attached to the communicator MPI\_COMM\_WORLD. The attachment 33 of error handlers to objects is purely local: different processes may attach different error 34 handlers to corresponding objects. 35

36 37

38

39

40

4142

43

Several predefined error handlers are available in MPI:

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

Implementations may provide additional predefined error handlers and programmers 44can code their own error handlers. 45

The error handler MPI\_ERRORS\_ARE\_FATAL is associated by default with MPI\_COMM-46\_WORLD after initialization. Thus, if the user chooses not to control error handling, every 47 error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, 48

1

 $\mathbf{2}$ 

3

4

5

6

7 8

9

12

13

14

15

16

17

18

19

2021

22 23

24

25

26

27

a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI\_ERRORS\_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non-trivial MPI error handler.

Unless specified below for the case of process failures, after an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI\_ERRORS\_RETURN, does *not* necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so.

If the MPI exception raised is one of those relating to process failures as specified in Chapter 15, then that chapter defines the state of MPI.

Advice to implementors. A high-quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors. (End of advice to implementors.)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C has distinct typedefs for user defined error handling callback functions that accept communicator, file, and window arguments. In Fortran there are three user routines.

An error handler object is created by a call to MPI\_XXX\_CREATE\_ERRHANDLER, where XXX is, respectively, COMM, WIN, or FILE.

An error handler is attached to a communicator, window, or file by a call to MPI\_XXX\_SET\_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI\_XXX\_CREATE\_ERRHANDLER, with matching XXX. The predefined error handlers MPI\_ERRORS\_RETURN and MPI\_ERRORS\_ARE\_FATAL can be attached to communicators, windows, and files.

The error handler currently associated with a communicator, window, or file can be retrieved by a call to MPI\_XXX\_GET\_ERRHANDLER.

The MPI function MPI\_ERRHANDLER\_FREE can be used to free an error handler that was created by a call to MPI\_XXX\_CREATE\_ERRHANDLER.

MPI\_{COMM,WIN,FILE}\_GET\_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI\_ERRHANDLER\_FREE should be called with the error handler returned from MPI\_{COMM,WIN,FILE}\_GET\_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI\_COMM\_GROUP and MPI\_GROUP\_FREE.

Advice to implementors. High-quality implementations should raise an error when an error handler that was created by a call to MPI\_XXX\_CREATE\_ERRHANDLER is 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.

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{31}$ 

```
344
                                  CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT
1
     8.3.1
            Error Handlers for Communicators
\mathbf{2}
3
4
     MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)
5
       IN
                 comm_errhandler_fn
                                            user defined error handling procedure (function)
6
       OUT
7
                 errhandler
                                            MPI error handler (handle)
8
9
     int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function
10
                    *comm_errhandler_fn, MPI_Errhandler *errhandler)
11
     MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
12
          PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
13
          TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
17
          EXTERNAL COMM_ERRHANDLER_FN
18
          INTEGER ERRHANDLER, IERROR
19
          Creates an error handler that can be attached to communicators.
20
         The user routine should be, in C, a function of type MPI_Comm_errhandler_function, which
21
     is defined as
22
     typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
23
24
         The first argument is the communicator in use. The second is the error code to be
25
     returned by the MPI routine that raised the error. If the routine would have returned
26
     MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused
27
     the error handler to be invoked. The remaining arguments are "varargs" arguments whose
28
     number and meaning is implementation-dependent. An implementation should clearly doc-
29
     ument these arguments. Addresses are used so that the handler may be written in Fortran.
30
     With the Fortran mpi_f08 module, the user routine comm_errhandler_fn should be of the
31
     form:
32
     ABSTRACT INTERFACE
33
       SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
34
            TYPE(MPI_Comm) :: comm
35
            INTEGER :: error_code
36
     With the Fortran mpi module and mpif.h, the user routine COMM_ERRHANDLER_FN
37
     should be of the form:
38
     SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
39
          INTEGER COMM, ERROR_CODE
40
41
42
                        The variable argument list is provided because it provides an ISO-
           Rationale.
43
           standard hook for providing additional information to the error handler; without this
44
          hook, ISO C prohibits additional arguments. (End of rationale.)
45
46
                             A newly created communicator inherits the error handler that
           Advice to users.
47
          is associated with the "parent" communicator. In particular, the user can specify
48
           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.*)

			4		
			4 5		
MPI_COMM_SET_ERRHANDLER(comm, errhandler)					
INOUT	comm	communicator (handle)	7		
IN	errhandler		8		
IIN	ermandler	new error handler for communicator (handle)	9		
int MDT (		MDI Euchendlen euchendlen)	10		
int MPI_C	.omm_set_errnandler(MP1_CC	omm comm, MPI_Errhandler errhandler)	11		
MPI_Comm_	<pre>set_errhandler(comm, errh</pre>	nandler, ierror)	12		
	MPI_Comm), INTENT(IN) ::		13		
	MPI_Errhandler), INTENT(]		14		
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	15		
MPI_COMM_	SET_ERRHANDLER(COMM, ERRH	HANDLER, IERROR)	16 17		
	ER COMM, ERRHANDLER, IERF		18		
Attac	has a new array handler to a	communicator. The error handler must be either	19		
		handler created by a call to	20		
-	M_CREATE_ERRHANDLER.	nandier created by a can to	21		
			22		
			23		
MPI_COM	M_GET_ERRHANDLER(comm	n, errhandler)	24		
IN	comm	communicator (handle)	25		
OUT	errhandler	error handler currently associated with communicator	26		
001		(handle)	27		
		()	28		
int MPT (	comm get errhandler(MPI Co	omm comm, MPI_Errhandler *errhandler)	29		
	-		30 31		
	get_errhandler(comm, errh		32		
	MPI_Comm), INTENT(IN) ::		33		
	<pre>MPI_Errhandler), INTENT(C ER, OPTIONAL, INTENT(OUT)</pre>		34		
	ER, OFIIONAL, INIENI(001)		35		
	GET_ERRHANDLER(COMM, ERRH	-	36		
INTEG	ER COMM, ERRHANDLER, IERF	ROR	37		
Retrie	eves the error handler currently	y associated with a communicator.	38		
For ex	ample, a library function may	register at its entry point the current error handler	39		
for a com	nunicator, set its own private	error handler for this communicator, and restore	40		
before exit	ing the previous error handler		41		
			42		
			43 $44$		
			44		
			47		

12

3

```
346
                                 CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT
1
     8.3.2
            Error Handlers for Windows
\mathbf{2}
3
4
     MPI_WIN_CREATE_ERRHANDLER(win_errhandler_fn, errhandler)
5
                win_errhandler_fn
       IN
                                           user defined error handling procedure (function)
6
       OUT
7
                errhandler
                                           MPI error handler (handle)
8
9
     int MPI_Win_create_errhandler(MPI_Win_errhandler_function
10
                    *win_errhandler_fn, MPI_Errhandler *errhandler)
11
     MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
12
         PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
13
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
17
         EXTERNAL WIN_ERRHANDLER_FN
18
         INTEGER ERRHANDLER, IERROR
19
         Creates an error handler that can be attached to a window object. The user routine
20
     should be, in C, a function of type MPI_Win_errhandler_function which is defined as
21
     typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
22
23
         The first argument is the window in use, the second is the error code to be returned.
^{24}
     With the Fortran mpi_f08 module, the user routine win_errhandler_fn should be of the form:
25
     ABSTRACT INTERFACE
26
       SUBROUTINE MPI_Win_errhandler_function(win, error_code)
27
            TYPE(MPI_Win) :: win
28
            INTEGER :: error_code
29
     With the Fortran mpi module and mpif.h, the user routine WIN_ERRHANDLER_FN should
30
     be of the form:
^{31}
     SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
32
         INTEGER WIN, ERROR_CODE
33
34
35
36
     MPI_WIN_SET_ERRHANDLER(win, errhandler)
37
       INOUT
                win
                                            window (handle)
38
       IN
                errhandler
                                           new error handler for window (handle)
39
40
     int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
41
42
     MPI_Win_set_errhandler(win, errhandler, ierror)
43
         TYPE(MPI_Win), INTENT(IN) :: win
44
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
48
         INTEGER WIN, ERRHANDLER, IERROR
```

1 Attaches a new error handler to a window. The error handler must be either a pre- $\mathbf{2}$ defined error handler, or an error handler created by a call to 3 MPI\_WIN\_CREATE\_ERRHANDLER. 4 5 MPI\_WIN\_GET\_ERRHANDLER(win, errhandler) 6 IN 7 win window (handle) 8 OUT errhandler error handler currently associated with window (han-9 dle) 10 11 int MPI\_Win\_get\_errhandler(MPI\_Win win, MPI\_Errhandler \*errhandler) 1213 MPI\_Win\_get\_errhandler(win, errhandler, ierror) 14TYPE(MPI\_Win), INTENT(IN) :: win 15TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17 MPI\_WIN\_GET\_ERRHANDLER(WIN, ERRHANDLER, IERROR) 18 INTEGER WIN, ERRHANDLER, IERROR 19 20Retrieves the error handler currently associated with a window. 2122 8.3.3 Error Handlers for Files 23 $^{24}$ 25MPI\_FILE\_CREATE\_ERRHANDLER(file\_errhandler\_fn, errhandler) 26IN file\_errhandler\_fn user defined error handling procedure (function) 2728 OUT errhandler MPI error handler (handle) 2930 int MPI\_File\_create\_errhandler(MPI\_File\_errhandler\_function 31\*file\_errhandler\_fn, MPI\_Errhandler \*errhandler) 32 33 MPI\_File\_create\_errhandler(file\_errhandler\_fn, errhandler, ierror) PROCEDURE(MPI\_File\_errhandler\_function) :: file\_errhandler\_fn 34 TYPE(MPI\_Errhandler), INTENT(OUT) :: errhandler 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI\_FILE\_CREATE\_ERRHANDLER(FILE\_ERRHANDLER\_FN, ERRHANDLER, IERROR) 38 EXTERNAL FILE\_ERRHANDLER\_FN 39 INTEGER ERRHANDLER, IERROR 40 41 Creates an error handler that can be attached to a file object. The user routine should 42be, in C, a function of type MPI\_File\_errhandler\_function, which is defined as typedef void MPI\_File\_errhandler\_function(MPI\_File \*, int \*, ...); 43 44The first argument is the file in use, the second is the error code to be returned. 45With the Fortran mpi\_f08 module, the user routine file\_errhandler\_fn should be of the form: 46 ABSTRACT INTERFACE 47

SUBROUTINE MPI\_File\_errhandler\_function(file, error\_code)

Unofficial Draft for Comment Only

```
1
            TYPE(MPI_File) :: file
\mathbf{2}
            INTEGER :: error_code
3
     With the Fortran mpi module and mpif.h, the user routine FILE_ERRHANDLER_FN should
4
     be of the form:
5
     SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
6
          INTEGER FILE, ERROR_CODE
7
8
9
     MPI_FILE_SET_ERRHANDLER(file, errhandler)
10
11
       INOUT
                 file
                                             file (handle)
12
       IN
                 errhandler
                                            new error handler for file (handle)
13
14
     int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
15
16
     MPI_File_set_errhandler(file, errhandler, ierror)
17
          TYPE(MPI_File), INTENT(IN) :: file
18
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
19
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
21
          INTEGER FILE, ERRHANDLER, IERROR
22
23
          Attaches a new error handler to a file. The error handler must be either a predefined
^{24}
     error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.
25
26
     MPI_FILE_GET_ERRHANDLER(file, errhandler)
27
28
       IN
                 file
                                             file (handle)
29
       OUT
                 errhandler
                                            error handler currently associated with file (handle)
30
^{31}
32
     int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
33
     MPI_File_get_errhandler(file, errhandler, ierror)
34
          TYPE(MPI_File), INTENT(IN) :: file
35
          TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
36
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
38
          INTEGER FILE, ERRHANDLER, IERROR
39
40
         Retrieves the error handler currently associated with a file.
41
42
43
44
45
46
47
48
```

8.3.	ERRO	OR HANDLING		349
8.3.4	1 Fre	eing Errorhandlers and Retrie	ving Error Strings	1 2
				3
MPI.	ERRH	ANDLER_FREE( errhandler )		4
IN	OUT	errhandler	MPI error handler (handle)	5
				6 7
int	MPI_E	rrhandler_free(MPI_Errhan	dler *errhandler)	8
MPI_	Errha	ndler_free(errhandler, ie	rror)	9
		MPI_Errhandler), INTENT(I ER, OPTIONAL, INTENT(OUT)		10 11
мрт	FBBHA	NDLER_FREE(ERRHANDLER, IE	RBUB)	12 13
···· +_	-	ER ERRHANDLER, IERROR		13
				15
			with errhandler for deallocation and sets errhan	16
			handler will be deallocated after all the obj	lects
asso	ciated	with it (communicator, windo	w, or file) have been deallocated.	18
				19
MPI.	_ERRO	R_STRING( errorcode, string,	resultlen )	20
IN		errorcode	Error code returned by an MPI routine	21
	· <b>-</b>		v	22
Οι		string	Text that corresponds to the errorcode	23
Οι	JT	resultlen	Length (in printable characters) of the result retu	
			in string	25 26
				20 27
int	MPI_E	rror_string(int errorcode	, char *string, int *resultlen)	28
MPI	Error	_string(errorcode, string	, resultlen, ierror)	29
_		ER, INTENT(IN) :: errorc		30
	CHARA	CTER(LEN=MPI_MAX_ERROR_ST	RING), INTENT(OUT) :: string	31
	INTEG	ER, INTENT(OUT) :: resul	tlen	32
	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	33
мрт	FBBUB	_STRING(ERRORCODE, STRING	RESULTIEN TERROR)	34
···· +_		ER ERRORCODE, RESULTLEN,		35
		CTER*(*) STRING		36
				37
		0	with an error code or class. The argument st	•
	-	0	PI_MAX_ERROR_STRING characters long.	39
	The nu	imber of characters actually wi	ritten is returned in the output argument, resul	
	Ratio	<i>nale</i> The form of this function	on was chosen to make the Fortran and C bind	ings 41
			pointer to a string has two difficulties. First,	0
		-	cated and different for each error message (allow	
			e calls to MPI_ERROR_STRING to point to	-
	-		n, a function declared as returning CHARACTER	
		- /	ble, a PRINT statement. (End of rationale.)	47
	48			

1	8.4 Eri	or Codes and Classe	S
2 3 4 5 6	of MPI_SU as possible	CCESS). This is done to a e in the error code (for us	e left entirely to the implementation (with the exception llow an implementation to provide as much information e with MPI_ERROR_STRING). Dilication to interpret an error code, the routine
7 8 9 10	MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called <i>error classes</i> . Valid error classes are shown in Table 8.1 and Table 8.2. The error classes are a subset of the error codes: an MPI function may return an error class number; and the function MPI_ERROR_STRING can be used to compute the error		
11 12 13	error code		s. The values defined for MPI error classes are valid MPI
14 15		$0 = MPI_SUCCESS$	$<$ MPI_ERR $\leq$ MPI_ERR_LASTCODE.
16 17			ween MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that surn useful information about MPI_ERR_OTHER.
18 19 20 21 22 23	ratio	on of error classes and err	is necessary to be consistent with C practice; the sepa- cor codes allows us to define the error classes this way. s often a nice sanity check as well. ( <i>End of rationale.</i> )
23 24	MPI_ERR	OR_CLASS( errorcode, err	orclass )
25	IN	errorcode	Error code returned by an MPI routine
26 27	OUT	errorclass	Error class associated with errorcode
28 29	int MPI_1	Error_class(int error	code, int *errorclass)
30 31 32 33 34	INTE INTE	r_class(errorcode, er GER, INTENT(IN) :: en GER, INTENT(OUT) :: e GER, OPTIONAL, INTENT(	crorcode errorclass
35 36	_	R_CLASS(ERRORCODE, ERF GER ERRORCODE, ERRORCI	
37 38 39 40	The fitself.	function MPI_ERROR_CL	ASS maps each standard error code (error class) onto
41 42	8.5 Eri	ror Classes, Error Coo	des, and Error Handlers
43 44 45	this librar	y may have its own set of	library on top of an existing MPI implementation, and f error codes and classes. An example of such a library Chapter 13. For this purpose, functions are needed to:
46 47	1. add	a new error class to the o	ones an MPI implementation already knows.
48	2. asso	ciate error codes with this	s error class, so that $MPI\_ERROR\_CLASS$ works.
		Unofficia	Draft for Commont Only

		2
MPI_SUCCESS	No error	3
MPI_ERR_BUFFER	Invalid buffer pointer	4
MPI_ERR_COUNT	Invalid count argument	5
MPI_ERR_TYPE	Invalid datatype argument	6
MPI_ERR_TAG	Invalid tag argument	7
MPI_ERR_COMM	Invalid communicator	8
MPI_ERR_RANK	Invalid rank	9
MPI_ERR_REQUEST	Invalid request (handle)	10
MPI_ERR_ROOT	Invalid root	11
MPI_ERR_GROUP	Invalid group	12
MPI_ERR_OP	Invalid operation	13
MPI_ERR_TOPOLOGY	Invalid topology	14
MPI_ERR_DIMS	Invalid dimension argument	15
MPI_ERR_ARG	Invalid argument of some other kind	16
MPI_ERR_UNKNOWN	Unknown error	17
MPI_ERR_TRUNCATE	Message truncated on receive	18
MPI_ERR_OTHER	Known error not in this list	19
MPI_ERR_INTERN	Internal MPI (implementation) error	20
MPI_ERR_IN_STATUS	Error code is in status	21
MPI_ERR_PENDING	Pending request	22
MPI_ERR_KEYVAL	Invalid keyval has been passed	23
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory	24
	is exhausted	25
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM	26
MPI_ERR_INFO_KEY	Key longer than $MPI_MAX_INFO_KEY$	27
MPI_ERR_INFO_VALUE	Value longer than $MPI_MAX_INFO_VAL$	28
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE	29
MPI_ERR_SPAWN	Error in spawning processes	30
MPI_ERR_PORT	Invalid port name passed to	31
	MPI_COMM_CONNECT	32
MPI_ERR_SERVICE	Invalid service name passed to	33
	MPI_UNPUBLISH_NAME	34
MPI_ERR_NAME	Invalid service name passed to	35
	MPI_LOOKUP_NAME	36
MPI_ERR_WIN	Invalid win argument	37
MPI_ERR_SIZE	Invalid size argument	38
MPI_ERR_DISP	Invalid disp argument	39
MPI_ERR_INFO	Invalid info argument	40
MPI_ERR_LOCKTYPE	Invalid locktype argument	41
MPI_ERR_ASSERT	Invalid assert argument	42
MPI_ERR_RMA_CONFLICT	Conflicting accesses to window	43
MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls	44
		45
Table 9	1. Ennon alagges (Dant 1)	46

Table 8.1: Error classes (Part 1)

1 2	MPI_ERR_RMA_RANGE	Target memory is not part of the win- dow (in the case of a window created
3		with MPI_WIN_CREATE_DYNAMIC, tar-
4		get memory is not attached)
5	MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because
6		of resource exhaustion)
7	MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some pro-
8		cess in the group of the specified commu-
9		nicator cannot expose shared memory)
10	MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the
11		called function
12	MPI_ERR_FILE	Invalid file handle
13	MI 1_ERR_NOT_SAME	Collective argument not identical on all
14		processes, or collective routines called in
15		a different order by different processes
16	MPI_ERR_AMODE	Error related to the <b>amode</b> passed to
17	MFI_ERR_AMODE	MPI_FILE_OPEN
18	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
19	WFI_ERR_UNSUFFURIED_DATAREF	MPI_FILE_SET_VIEW
20	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
21	MFI_ERR_ONSOFFORTED_OFERATION	a file which supports sequential access only
22		File does not exist
23	MPI_ERR_NO_SUCH_FILE MPI_ERR_FILE_EXISTS	File exists
24	MPI_ERR_BAD_FILE	
25		Invalid file name (e.g., path name too long) Permission denied
26	MPI_ERR_ACCESS	
27	MPI_ERR_NO_SPACE	Not enough space
28	MPI_ERR_QUOTA	Quota exceeded
29	MPI_ERR_READ_ONLY	Read-only file or file system
30	MPI_ERR_FILE_IN_USE	File operation could not be completed, as
31		the file is currently open by some process
32	MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-
33		tered because a data representation identi-
34		fier that was already defined was passed to MPI_REGISTER_DATAREP
35		
36	MPI_ERR_CONVERSION	An error occurred in a user supplied data
37		conversion function. Other $L(Q)$ are a
38	MPI_ERR_IO	Other I/O error
39	MPI_ERR_PROC_FAILED	Operation could not complete due to pro-
40		cess failure (a fail-stop failure).
41	MPI_ERR_REVOKED	Communication object used in the opera-
42		tion has been revoked.
43	MPI_ERR_PROC_FAILED_PENDING	Non-blocking operation could not complete
44		due to process failure, but request is still
45		pending.
45	MPI_ERR_LASTCODE	Last error code
40		
48	Table 8.2: Err	cor classes (Part 2)

3. associate strings with these error codes, so that MPI\_ERROR\_STRING works.

4. invoke the error handler associated with a communicator, window, or object.

Several functions are provided to do this. They are all local. No functions are provided to free error classes or codes: it is not expected that an application will generate them in significant numbers.

MPI_ADD_ERROR_CLASS(errorclass)		9
OUT errorclass	value for the new error class (integer)	10
		11
<pre>int MPI_Add_error_class(int *erro</pre>	rclass)	12
		13
<pre>MPI_Add_error_class(errorclass, i</pre>		14
INTEGER, INTENT(OUT) :: erro		15
INTEGER, OPTIONAL, INTENT(OUT	) :: ierror	16
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)		17
INTEGER ERRORCLASS, IERROR		18
		19
Creates a new error class and retur	rns the value for it.	20
<i>Rationale.</i> To avoid conflicts with existing error codes and classes, the value is set by the implementation and not by the user. ( <i>End of rationale.</i> )		21
		22
by the implementation and not by	the user. (Ena of fationale.)	23
Advice to implementors. A high	h-quality implementation will return the value for	24
a new errorclass in the same deterministic way on all processes. (End of advice to implementors.)		25
		26
		27
	PI_ADD_ERROR_CLASS is local, the same errorclass	28 29
may not be returned on all processes that make this call. Thus, it is not safe to assume that registering a new error on a set of processes at the same time will yield the same		
errorclass in a deterministic way, a	and they are always generated in the same order on	32

the same set of processes (for example, all processes), then the value will be the same. However, even if a deterministic algorithm is used, the value can vary across processes. This can happen, for example, if different but overlapping groups of processes make a series of calls. As a result of these issues, getting the "same" error on multiple processes may not cause the same value of error code to be generated. (*End of advice* to users.)

The value of MPI\_ERR\_LASTCODE is a constant value and is not affected by new userdefined error codes and classes. Instead, a predefined attribute key MPI\_LASTUSEDCODE is associated with MPI\_COMM\_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different processes. The value returned by this key is always greater than or equal to MPI\_ERR\_LASTCODE.

Advice to users. The value returned by the key MPI\_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class/code. In a multithreaded environment, the user must take extra care in assuming this value has not 48

### Unofficial Draft for Comment Only

1 2

3 4

 $\mathbf{5}$ 

6

7

33

34

35

36

37

38 39

40

41

42

43

44

```
1
           changed. Note that error codes and error classes are not necessarily dense. A user
\mathbf{2}
           may not assume that each error class below MPI_LASTUSEDCODE is valid. (End of
3
           advice to users.)
4
5
6
     MPI_ADD_ERROR_CODE(errorclass, errorcode)
7
8
       IN
                 errorclass
                                              error class (integer)
9
       OUT
                 errorcode
                                              new error code to associated with errorclass (integer)
10
11
     int MPI_Add_error_code(int errorclass, int *errorcode)
12
13
     MPI_Add_error_code(errorclass, errorcode, ierror)
14
          INTEGER, INTENT(IN) :: errorclass
15
          INTEGER, INTENT(OUT) ::
                                       errorcode
16
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
18
          INTEGER ERRORCLASS, ERRORCODE, IERROR
19
20
          Creates new error code associated with errorclass and returns its value in errorcode.
21
22
           Rationale.
                       To avoid conflicts with existing error codes and classes, the value of the
23
           new error code is set by the implementation and not by the user. (End of rationale.)
24
           Advice to implementors.
                                      A high-quality implementation will return the value for
25
           a new errorcode in the same deterministic way on all processes. (End of advice to
26
           implementors.)
27
28
29
30
     MPI_ADD_ERROR_STRING(errorcode, string)
^{31}
       IN
                 errorcode
                                              error code or class (integer)
32
33
       IN
                 string
                                              text corresponding to errorcode (string)
34
35
     int MPI_Add_error_string(int errorcode, const char *string)
36
     MPI_Add_error_string(errorcode, string, ierror)
37
          INTEGER, INTENT(IN) :: errorcode
38
          CHARACTER(LEN=*), INTENT(IN) :: string
39
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
40
41
     MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
42
          INTEGER ERRORCODE, IERROR
43
          CHARACTER*(*) STRING
44
          Associates an error string with an error code or class. The string must be no more
45
     than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the
46
```

calling language. The length of the string does not include the null terminator in C. Trailing blanks will be stripped in Fortran. Calling MPI\_ADD\_ERROR\_STRING for an errorcode that

47

already has a string will replace the old string with the new string. It is erroneous to call MPI\_ADD\_ERROR\_STRING for an error code or class with a value  $\leq$  MPI\_ERR\_LASTCODE.

If MPI\_ERROR\_STRING is called when no string has been set, it will return a empty string (all spaces in Fortran, "" in C).

Section 8.3 describes the methods for creating and associating error handlers with communicators, files, and windows.

MPI\_COMM\_CALL\_ERRHANDLER (comm, errorcode)
IN comm communicator with error handler (handle)
IN errorcode error code (integer)
int MPI\_Comm\_call\_errhandler(MPI\_Comm comm, int errorcode)
MPI\_Comm\_call\_errhandler(comm, errorcode, ierror)
TYPE(MPI\_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: errorcode
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI\_COMM\_CALL\_ERRHANDLER(COMM, ERRORCODE, IERROR)
INTEGER COMM, ERRORCODE, IERROR

This function invokes the error handler assigned to the communicator with the error code supplied. This function returns MPI\_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Users should note that the default error handler is MPI\_ERRORS\_ARE\_FATAL. Thus, calling MPI\_COMM\_CALL\_ERRHANDLER will abort the comm processes if the default error handler has not been changed for this communicator or on the parent before the communicator was created. (*End of advice to* users.)

MPI\_WIN\_CALL\_ERRHANDLER (win, errorcode) IN window with error handler (handle) win IN errorcode error code (integer) int MPI\_Win\_call\_errhandler(MPI\_Win win, int errorcode) MPI\_Win\_call\_errhandler(win, errorcode, ierror) TYPE(MPI\_Win), INTENT(IN) :: win INTEGER, INTENT(IN) :: errorcode INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_WIN\_CALL\_ERRHANDLER(WIN, ERRORCODE, IERROR) INTEGER WIN, ERRORCODE, IERROR

 $^{24}$ 

1 This function invokes the error handler assigned to the window with the error code  $\mathbf{2}$ supplied. This function returns MPI\_SUCCESS in C and the same value in IERROR if the 3 error handler was successfully called (assuming the process is not aborted and the error 4 handler returns). 5Advice to users. As with communicators, the default error handler for windows is 6 MPI\_ERRORS\_ARE\_FATAL. (End of advice to users.) 7 8 9 10MPI\_FILE\_CALL\_ERRHANDLER (fh, errorcode) 11IN fh file with error handler (handle) 1213 IN errorcode error code (integer) 1415int MPI\_File\_call\_errhandler(MPI\_File fh, int errorcode) 16MPI\_File\_call\_errhandler(fh, errorcode, ierror) 17TYPE(MPI\_File), INTENT(IN) :: fh 18 INTEGER, INTENT(IN) :: errorcode 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2021MPI\_FILE\_CALL\_ERRHANDLER(FH, ERRORCODE, IERROR) 22INTEGER FH, ERRORCODE, IERROR 23This function invokes the error handler assigned to the file with the error code supplied.  $^{24}$ This function returns MPI\_SUCCESS in C and the same value in IERROR if the error handler 25was successfully called (assuming the process is not aborted and the error handler returns). 2627Advice to users. Unlike errors on communicators and windows, the default behavior 28for files is to have MPI\_ERRORS\_RETURN. (*End of advice to users.*) 29 30 Advice to users. Users are warned that handlers should not be called recursively 31with MPI\_COMM\_CALL\_ERRHANDLER, MPI\_FILE\_CALL\_ERRHANDLER, or 32 MPI\_WIN\_CALL\_ERRHANDLER. Doing this can create a situation where an infinite 33 recursion is created. This can occur if MPI\_COMM\_CALL\_ERRHANDLER, 34 MPI\_FILE\_CALL\_ERRHANDLER, or MPI\_WIN\_CALL\_ERRHANDLER is called inside 35an error handler. 36 Error codes and classes are associated with a process. As a result, they may be used 37 in any error handler. Error handlers should be prepared to deal with any error code 38 they are given. Furthermore, it is good practice to only call an error handler with the 39 appropriate error codes. For example, file errors would normally be sent to the file 40 error handler. (End of advice to users.) 41 42Timers and Synchronization 438.6 4445MPI 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 incon venient or do not provide adequate access to high resolution timers. See also Section 2.6.4.

MPI_WTIME()
double MPI_Wtime(void)
DOUBLE PRECISION MPI_Wtime()
DOUBLE PRECISION MPI_WTIME()
MPL WTIME returns a floating point number of seconds, representing alapsed wall

MPI\_WTIME returns a floating-point number of seconds, representing elapsed wallclock time since some time in the past. The "time in the past" is guaranteed not to change during the life of the process.

The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```
{
    double starttime, endtime;
    starttime = MPI_Wtime();
    .... stuff to be timed ...
    endtime = MPI_Wtime();
    printf("That took %f seconds\n",endtime-starttime);
}
```

}

The times returned are local to the node that called them. There is no requirement that different nodes return "the same time." (But see also the discussion of MPI\_WTIME\_IS\_GLOBAL in Section 8.1.2).

MPI\_WTICK()

double MPI\_Wtick(void)

```
DOUBLE PRECISION MPI_Wtick()
```

```
DOUBLE PRECISION MPI_WTICK()
```

MPI\_WTICK returns the resolution of MPI\_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI\_WTICK should be  $10^{-3}$ .

### 8.7 Startup

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed

1 before other MPI routines may be called. To provide for this, MPI includes an initialization  $\mathbf{2}$ routine MPI\_INIT. 3 4 MPI\_INIT() 56 int MPI\_Init(int \*argc, char \*\*\*argv) 7 8 MPI\_Init(ierror) 9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 MPI\_INIT(IERROR) 11 INTEGER IERROR 1213 All MPI programs must contain exactly one call to an MPI initialization routine: 14MPI\_INIT or MPI\_INIT\_THREAD. Subsequent calls to any initialization routines are erro-15neous. The only MPI functions that may be invoked before the MPI initialization routines 16are called are MPI\_GET\_VERSION, MPI\_GET\_LIBRARY\_VERSION, MPI\_INITIALIZED, 17MPI\_FINALIZED, and any function with the prefix MPI\_T\_ (within the constraints for func-18tions with this prefix listed in Section 14.3.4). The version for ISO C accepts the argc and 19argv that are provided by the arguments to main or NULL: 20int main(int argc, char \*argv[]) 21{ 22MPI\_Init(&argc, &argv); 2324/\* parse arguments \*/ 25/\* main program 26\*/ 27MPI\_Finalize(); /\* see below \*/ 28 return 0; 29 } 30  $^{31}$ The Fortran version takes only IERROR. 32 Conforming implementations of MPI are required to allow applications to pass NULL 33 for both the argc and argv arguments of main in C. 34 After MPI is initialized, the application can access information about the execution 35 environment by querying the predefined info object MPI\_INFO\_ENV. The following keys are 36 predefined for this object, corresponding to the arguments of MPI\_COMM\_SPAWN or of 37 mpiexec: 38 39 command Name of program executed. 40argy Space separated arguments to command. 41 42maxprocs Maximum number of MPI processes to start. 43  $^{44}$ soft Allowed values for number of processors. 45host Hostname. 4647 arch Architecture name. 48

8	3.7. STARTUP 359	
W	vdir Working directory of the MPI process.	1
fi	ile Value is the name of a file in which additional information is specified.	2 3
tl	hread_level Requested level of thread support, if requested before the program started exe- cution.	4 5 6
" p n o t	Note that all values are strings. Thus, the maximum number of processes is represented by a string such as "1024" and the requested level is represented by a string such as "MPI_THREAD_SINGLE". The info object MPI_INFO_ENV need not contain a (key,value) pair for each of these predefined keys; the set of (key,value) pairs provided is implementation-dependent. Imple- mentations may provide additional, implementation specific, (key,value) pairs. In case where the MPI processes were started with MPI_COMM_SPAWN_MULTIPLE or, equivalently, with a startup mechanism that supports multiple process specifications, hen the values stored in the info object MPI_INFO_ENV at a process are those values that affect the local MPI process.	7 8 9 10 11 12 13 14 15 16
E	Example 8.4 If MPI is started with a call to	17 18
	mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos	19
0	Then the first 5 processes will have have in their MPI_INFO_ENV object the pairs (command, ocean), (maxprocs, 5), and (arch, sun). The next 10 processes will have in MPI_INFO_ENV command, atmos), (maxprocs, 10), and (arch, rs6000)	20 21 22 23
	Advice to users. The values passed in MPI_INFO_ENV are the values of the arguments passed to the mechanism that started the MPI execution — not the actual value provided. Thus, the value associated with maxprocs is the number of MPI processes requested; it can be larger than the actual number of processes obtained, if the soft option was used. ( <i>End of advice to users.</i> )	25 26 27
	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.)	30 31 32 33
Ν	/IPI_FINALIZE()	34 35
		36 37
	nt MPI_Finalize(void)	38
М	<pre>IPI_Finalize(ierror)     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	39 40
М	IPI_FINALIZE(IERROR)	41 42

MPI\_FINALIZE(IERROR) INTEGER IERROR

This routine cleans up all MPI state. If an MPI program terminates normally (i.e., not due to a call to MPI\_ABORT or an unrecoverable error) then each process must call MPI\_FINALIZE before it exits.

Before an MPI process invokes MPI\_FINALIZE, the process must perform all MPI calls 4748needed to complete its involvement in MPI communications: It must locally complete all

### **Unofficial Draft for Comment Only**

4344

45

1MPI operations that it initiated and must execute matching calls needed to complete MPI  $\mathbf{2}$ communications initiated by other processes. For example, if the process executed a non-3 blocking send, it must eventually call MPI\_WAIT, MPI\_TEST, MPI\_REQUEST\_FREE, or 4 any derived function; if the process is the target of a send, then it must post the matching  $\mathbf{5}$ receive; if it is part of a group executing a collective operation, then it must have completed 6 its participation in the operation. 7 The call to MPI\_FINALIZE does not free objects created by MPI calls; these objects are 8 freed using MPI\_XXX\_FREE calls. 9 MPI\_FINALIZE is collective over all connected processes. If no processes were spawned, 10 accepted or connected then this means over MPI\_COMM\_WORLD; otherwise it is collective 11over the union of all processes that have been and continue to be connected, as explained 12in Section 10.5.4. 13 The following examples illustrates these rules 14**Example 8.5** The following code is correct 1516Process 0 Process 1 17 \_\_\_\_\_ \_\_\_\_\_ 18 MPI\_Init(); MPI\_Init(); 19 MPI\_Send(dest=1); MPI\_Recv(src=0); 20MPI\_Finalize(); MPI\_Finalize(); 2122 23**Example 8.6** Without a matching receive, the program is erroneous 2425Process 0 Process 1 26\_\_\_\_\_ \_\_\_\_\_ 27MPI\_Init(); MPI\_Init(); 28MPI\_Send (dest=1); 29 MPI\_Finalize(); MPI\_Finalize(); 30  $^{31}$ **Example 8.7** This program is correct: Process 0 calls MPI\_Finalize after it has executed 32 the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call 33 that completes the matching receive operation before it calls MPI\_Finalize. 34 35 Process 0 Proces 1 36 \_\_\_\_\_ \_\_\_\_\_ 37 MPI\_Init(); MPI\_Init(); 38 MPI\_Isend(dest=1); MPI\_Recv(src=0); 39 MPI\_Request\_free(); MPI\_Finalize(); 40 MPI\_Finalize(); exit(); 41 exit(); 4243 **Example 8.8** This program is correct. The attached buffer is a resource allocated by the 44 user, not by MPI; it is available to the user after MPI is finalized. 45464748

Process 0	Process 1
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>
<pre>buffer = malloc(1000000);</pre>	<pre>MPI_Recv(src=0);</pre>
<pre>MPI_Buffer_attach();</pre>	<pre>MPI_Finalize();</pre>
<pre>MPI_Send(dest=1));</pre>	<pre>exit();</pre>
<pre>MPI_Finalize();</pre>	
<pre>free(buffer);</pre>	
<pre>exit();</pre>	

**Example 8.9** This program is correct. The cancel operation must succeed, since the send cannot complete normally. The wait operation, after the call to MPI\_Cancel, is local — no matching MPI call is required on process 1.

Process 0	Process 1
<pre>MPI_Issend(dest=1);</pre>	<pre>MPI_Finalize();</pre>
<pre>MPI_Cancel();</pre>	
<pre>MPI_Wait();</pre>	
<pre>MPI_Finalize();</pre>	

Advice to implementors. Even though a process has executed all MPI calls needed to complete the communications it is involved with, such communication may not yet be completed from the viewpoint of the underlying MPI system. For example, a blocking send may have returned, even though the data is still buffered at the sender in an MPI buffer; an MPI process may receive a cancel request for a message it has completed receiving. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI\_FINALIZE returns. Thus, if a process exits after the call to MPI\_FINALIZE, this will not cause an ongoing communication to fail. The MPI implementation should also complete freeing all objects marked for deletion by MPI calls that freed them. (*End of advice to implementors.*)

Once MPI\_FINALIZE returns, no MPI routine (not even MPI\_INIT) may be called, except for MPI\_GET\_VERSION, MPI\_GET\_LIBRARY\_VERSION, MPI\_INITIALIZED, MPI\_FINALIZED, and any function with the prefix MPI\_T\_ (within the constraints for functions with this prefix listed in Section 14.3.4).

Although it is not required that all processes return from MPI\_FINALIZE, it is required that at least process 0 in MPI\_COMM\_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, users may desire to supply an exit code for each process that returns from MPI\_FINALIZE.

When considering a process fault tolerant application, process 0 may have failed, in which case the semantics regarding process 0 and MPI\_FINALIZE are further defined in Chapter 15.

**Example 8.10** The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

#### Unofficial Draft for Comment Only

 $^{24}$ 

```
1
          . . .
\mathbf{2}
          MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
3
          . . .
4
          MPI_Finalize();
5
          if (myrank == 0) {
6
              resultfile = fopen("outfile","w");
7
              dump_results(resultfile);
8
              fclose(resultfile);
9
          }
10
          exit(0);
11
12
13
     MPI_INITIALIZED(flag)
14
                                              Flag is true if MPI_INIT has been called and false
15
       OUT
                 flag
16
                                              otherwise.
17
18
     int MPI_Initialized(int *flag)
19
     MPI_Initialized(flag, ierror)
20
          LOGICAL, INTENT(OUT) :: flag
21
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_INITIALIZED(FLAG, IERROR)
^{24}
          LOGICAL FLAG
25
          INTEGER IERROR
26
          This routine may be used to determine whether MPI_INIT has been called.
27
     MPI_INITIALIZED returns true if the calling process has called MPI_INIT. Whether
28
     MPI_FINALIZE has been called does not affect the behavior of MPI_INITIALIZED. It is one
29
     of the few routines that may be called before MPI_INIT is called. This function must always
30
     be thread-safe, as defined in Section 12.4.
^{31}
32
33
     MPI_ABORT(comm, errorcode)
34
       IN
                 comm
                                              communicator of tasks to abort
35
36
       IN
                 errorcode
                                              error code to return to invoking environment
37
38
     int MPI_Abort(MPI_Comm comm, int errorcode)
39
     MPI_Abort(comm, errorcode, ierror)
40
          TYPE(MPI_Comm), INTENT(IN) :: comm
41
          INTEGER, INTENT(IN) :: errorcode
42
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
43
44
     MPI_ABORT(COMM, ERRORCODE, IERROR)
45
          INTEGER COMM, ERRORCODE, IERROR
46
          This routine makes a "best attempt" to abort all tasks in the group of comm. This
47
     function does not require that the invoking environment take any action with the error
48
```

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10

11

12 13

14

15

16 17 18

19

20

21 22

23 24

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36 37

38

39

40

41

42

43

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. If no processes were spawned, accepted, or connected then this has the effect of aborting all the processes associated with MPI\_COMM\_WORLD.

*Rationale.* The communicator argument is provided to allow for future extensions of MPI to environments with, for example, dynamic process management. In particular, it allows but does not require an MPI implementation to abort a subset of MPI\_COMM\_WORLD. (*End of rationale.*)

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

### 8.7.1 Allowing User Functions at Process Termination

There are times in which it would be convenient to have actions happen when an MPI process finishes. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that being terminated in the case of dynamically created processes) is finished. This can be accomplished in MPI by attaching an attribute to MPI\_COMM\_SELF with a callback function. When MPI\_FINALIZE is called, it will first execute the equivalent of an MPI\_COMM\_FREE on MPI\_COMM\_SELF. This will cause the delete callback function to be executed on all keys associated with MPI\_COMM\_SELF, in the reverse order that they were set on MPI\_COMM\_SELF. If no key has been attached to MPI\_COMM\_SELF, then no callback is invoked. The "freeing" of MPI\_COMM\_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI\_FINALIZED will return false in any of these callback functions. Once done with MPI\_COMM\_SELF, the order and rest of the actions taken by MPI\_FINALIZE is not specified.

Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. Implementations that use the attribute delete callback on MPI\_COMM\_SELF internally should register their internal callbacks before returning from MPI\_INIT / MPI\_INIT\_THREAD, so that libraries or applications will not have portions of the MPI implementation shut down before the application-level callbacks are made. (*End of advice to implementors.*)

### 8.7.2 Determining Whether MPI Has Finished

One of the goals of MPI was to allow for layered libraries. In order for a library to do this cleanly, it needs to know if MPI is active. In MPI the function MPI\_INITIALIZED was

### Unofficial Draft for Comment Only

44 45 46

47

provided to tell if MPI had been initialized. The problem arises in knowing if MPI has been
 finalized. Once MPI has been finalized it is no longer active and cannot be restarted. A
 library needs to be able to determine this to act accordingly. To achieve this the following
 function is needed:

```
<sup>6</sup>
7 MPI_FINALIZED(flag)
```

8 OUT flag true if MPI was finalized (logical) 9 10 int MPI\_Finalized(int \*flag) 11 MPI\_Finalized(flag, ierror) 12LOGICAL, INTENT(OUT) :: flag 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI\_FINALIZED(FLAG, IERROR) 16LOGICAL FLAG 17 INTEGER IERROR 18 This routine returns true if MPI\_FINALIZE has completed. It is valid to call 19MPI\_FINALIZED before MPI\_INIT and after MPI\_FINALIZE. This function must always be 20

thread-safe, as defined in Section 12.4.

Advice to users. MPI is "active" and it is thus safe to call MPI functions if MPI\_INIT has completed and MPI\_FINALIZE has not completed. If a library has no other way of knowing whether MPI is active or not, then it can use MPI\_INITIALIZED and MPI\_FINALIZED to determine this. For example, MPI is "active" in callback functions that are invoked during MPI\_FINALIZE. (End of advice to users.)

27 28 29

30 31

32

33

34

21 22

23

24

25

26

### 8.8 Portable MPI Process Startup

A number of implementations of  $\mathsf{MPI}$  provide a startup command for  $\mathsf{MPI}$  programs that is of the form

mpirun <mpirun arguments> <program> <program arguments>

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the MPI program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard starup mechanism. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead of mpirun MPI specifies mpiexec.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an mpiexec startup command

and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called **mpiexec**, it must be of the form described below.

It is suggested that

mpiexec -n <numprocs> <program>

be at least one way to start <program> with an initial MPI\_COMM\_WORLD whose group contains <numprocs> processes. Other arguments to mpiexec may be implementation-dependent.

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that mpiexec be able to be viewed as a command-line version of MPI\_COMM\_SPAWN (See Section 10.3.4).

Analogous to MPI\_COMM\_SPAWN, we have

mpiexec -n	<maxproc< th=""><th>s&gt;</th></maxproc<>	s>
-soft	<	>
-host	<	>
-arch	<	>
-wdir	<	>
-path	<	>
-file	<	>
<comman< td=""><td>nd line&gt;</td><td></td></comman<>	nd line>	

for the case where a single command line for the application program and its arguments will suffice. See Section 10.3.4 for the meanings of these arguments. For the case corresponding to MPI\_COMM\_SPAWN\_MULTIPLE there are two possible formats:

Form A:

```
mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }
```

As with MPI\_COMM\_SPAWN, all the arguments are optional. (Even the  $-n \ge argument$  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>
```

 $41 \\ 42$ 

 $45 \\ 46$ 

1	where the lines of <i>&lt;</i> filename> are of the form separated by the colons in Form A.
2	Lines beginning with '#' are comments, and lines may be continued by terminating
3	the partial line with $\langle \cdot \rangle$ .
4	
5	<b>Example 8.11</b> Start 16 instances of myprog on the current or default machine:
6	Example 0.11 Start to instances of myprog on the current of default machine.
7	mpiexec -n 16 myprog
8	mbrexec an io mybrog
9	
10	<b>Example 8.12</b> Start 10 processes on the machine called ferrari:
11	Example 0.12 Start to processes on the machine cance refrain.
12	mpiexec -n 10 -host ferrari myprog
13	mprexec -n 10 -nost rerrari myprog
14	
	<b>Example 8.13</b> Start three copies of the same program with different command-line
15	arguments:
16	arguments.
17	mniouss munrog infils1 : munrog infils2 : munrog infils2
18	<pre>mpiexec myprog infile1 : myprog infile2 : myprog infile3</pre>
19	
20	Example 8.14 Start the ocean program on five Suns and the atmos program on 10
21	RS/6000's:
22	105/0000 5.
23	mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
24	mprexec -n 5 -arch sun oceann 10 -arch 180000 acmos
25	It is second that the implementation is this second as mothed for sheering hasts of
26	It is assumed that the implementation in this case has a method for choosing hosts of
27	the appropriate type. Their ranks are in the order specified.
28	
29	<b>Example 8.15</b> Start the ocean program on five Suns and the atmos program on 10
30	RS/6000's (Form B):
31	
32	mpiexec -configfile myfile
33	
34	where myfile contains
35	
36	-n 5 -arch sun ocean
37	-n 10 -arch rs6000 atmos
38	
39	(End of advice to implementors.)
40	
41	
42	
43	
43	
44	
45	
47 48	
-10	

# Chapter 9

# The Info Object

Many of the routines in MPI take an argument info. info is an opaque object with a handle of type MPI\_Info in C and Fortran with the mpi\_f08 module, and INTEGER in Fortran with the mpi module or the include file mpif.h. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI.

An implementation must support info objects as caches for arbitrary (key,value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI\_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI\_INFO\_GET\_NKEYS, MPI\_INFO\_GET\_NTHKEY, MPI\_INFO\_GET\_VALUELEN, and MPI\_INFO\_GET must retain all (key,value) pairs so that layered functionality can also use the Info object.

Keys have an implementation-defined maximum length of MPI\_MAX\_INFO\_KEY, which is at least 32 and at most 255. Values have an implementation-defined maximum length of MPI\_MAX\_INFO\_VAL. In Fortran, leading and trailing spaces are stripped from both. Returned values will never be larger than these maximum lengths. Both key and value are case sensitive.

*Rationale.* Keys have a maximum length because the set of known keys will always be finite and known to the implementation and because there is no reason for keys to be complex. The small maximum size allows applications to declare keys of size MPI\_MAX\_INFO\_KEY. The limitation on value sizes is so that an implementation is not forced to deal with arbitrarily long strings. (*End of rationale.*)

Advice to users. MPI\_MAX\_INFO\_VAL might be very large, so it might not be wise to declare a string of that size. (End of advice to users.)

When info is used as an argument to a nonblocking routine, it is parsed before that routine returns, so that it may be modified or freed immediately after return.

When the descriptions refer to a key or value as being a boolean, an integer, or a list, <sup>45</sup> they mean the string representation of these types. An implementation may define its own <sup>46</sup> rules for how info value strings are converted to other types, but to ensure portability, every <sup>47</sup> implementation must support the following representations. Valid values for a boolean must <sup>48</sup>

7 8 9

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33 34

35 36

37

38 39

40

41 42

43

1include the strings "true" and "false" (all lowercase). For integers, valid values must include  $\mathbf{2}$ string representations of decimal values of integers that are within the range of a standard 3 integer type in the program. (However it is possible that not every integer is a valid value 4 for a given key.) On positive numbers, + signs are optional. No space may appear between  $\mathbf{5}$ a + or - sign and the leading digit of a number. For comma separated lists, the string 6 must contain valid elements separated by commas. Leading and trailing spaces are stripped 7automatically from the types of info values described above and for each element of a comma 8 separated list. These rules apply to all info values of these types. Implementations are free 9 to specify a different interpretation for values of other info keys. 10 11 MPI\_INFO\_CREATE(info) 1213 OUT info info object created (handle) 1415int MPI\_Info\_create(MPI\_Info \*info) 16MPI\_Info\_create(info, ierror) 17TYPE(MPI\_Info), INTENT(OUT) :: info 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1920MPI\_INFO\_CREATE(INFO, IERROR) 21INTEGER INFO, IERROR 22MPI\_INFO\_CREATE creates a new info object. The newly created object contains no 23key/value pairs.  $^{24}$ 2526MPI\_INFO\_SET(info, key, value) 27info object (handle) INOUT info 2829IN key (string) key 30 IN value value (string)  $^{31}$ 32 int MPI\_Info\_set(MPI\_Info info, const char \*key, const char \*value) 33 34MPI\_Info\_set(info, key, value, ierror) 35 TYPE(MPI\_Info), INTENT(IN) :: info 36 CHARACTER(LEN=\*), INTENT(IN) :: key, value 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 MPI\_INFO\_SET(INFO, KEY, VALUE, IERROR) 39 INTEGER INFO, IERROR 40 CHARACTER\*(\*) KEY, VALUE 41 42MPI\_INFO\_SET adds the (key,value) pair to info, and overrides the value if a value for 43 the same key was previously set. key and value are null-terminated strings in C. In Fortran, 44leading and trailing spaces in key and value are stripped. If either key or value are larger 45than the allowed maximums, the errors MPI\_ERR\_INFO\_KEY or MPI\_ERR\_INFO\_VALUE are

46

47 48 raised, respectively.

MPI_INFO	_DELETE(info, key)		1
INOUT	info	info object (handle)	2
IN	key	key (string)	3
	icy .	kcy (string)	4 5
int MPI I	nfo_delete(MPI_Info info	. const char *kev)	6
		·	7
	_delete(info, key, ierror) [MPI_Info), INTENT(IN) ::		8
	CTER(LEN=*), INTENT(IN) :		9
	ER, OPTIONAL, INTENT(OUT)	•	10
MDT TNEO	DELETE (INEO VEV IEDDOD)		11 12
	DELETE(INFO, KEY, IERROR) ER INFO, IERROR	)	13
	CTER*(*) KEY		14
		unders) which forms inform If here is not defined in info	15
	ises an error of class $MPI_ERR$	value) pair from info. If key is not defined in info,	16
une cam ra			17
			18 19
MPI_INFO	_GET(info, key, valuelen, value	, flag)	20
IN	info	info object (handle)	21
IN	key	key (string)	22
IN	valuelen	length of value arg (integer)	23
OUT	value	value (string)	24
OUT	flag	true if key defined, false if not (boolean)	25 26
001	llag	true if key defined, false if not (boolean)	20
int MPT T	nfo get(MPI Info info co	onst char *key, int valuelen, char *value,	28
1110 111 1_1	int *flag)	shot onder (hoy, into variable), onder (variab,	29
MDT Trefe	-	uslus flag issues)	30
	<pre>_get(info, key, valuelen, MPI_Info), INTENT(IN) ::</pre>	-	31
	CTER(LEN=*), INTENT(IN)		32 33
	ER, INTENT(IN) :: value		34
CHARA	CTER(LEN=valuelen), INTEN	NT(OUT) :: value	35
	CAL, INTENT(OUT) :: flag		36
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	37
MPI_INFO_	GET(INFO, KEY, VALUELEN,	VALUE, FLAG, IERROR)	38
	ER INFO, VALUELEN, IERROF	2	39
	ACTER*(*) KEY, VALUE		40 41
LOGIC	CAL FLAG		41
This f	function retrieves the value as	ssociated with key in a previous call to	43
	° ,	t sets flag to true and returns the value in value,	44

MPI\_INFO\_SET. If such a key exists, it sets flag to true and returns the value in value, otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters available in value. If it is less than the actual size of the value, the value is truncated. In C, valuelen should be one less than the amount of allocated space to allow for the null terminator.

```
1
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
\mathbf{2}
3
     MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
4
5
       IN
                 info
                                              info object (handle)
6
       IN
                 key
                                              key (string)
7
       OUT
                 valuelen
                                              length of value arg (integer)
8
9
       OUT
                 flag
                                              true if key defined, false if not (boolean)
10
11
     int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
12
                     int *flag)
13
     MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
14
          TYPE(MPI_Info), INTENT(IN) :: info
15
          CHARACTER(LEN=*), INTENT(IN) :: key
16
          INTEGER, INTENT(OUT) :: valuelen
17
          LOGICAL, INTENT(OUT) ::
                                       flag
18
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
21
          INTEGER INFO, VALUELEN, IERROR
22
          LOGICAL FLAG
23
          CHARACTER*(*) KEY
24
          Retrieves the length of the value associated with key. If key is defined, valuelen is set to
25
     the length of its associated value and flag is set to true. If key is not defined, valuelen is not
26
     touched and flag is set to false. The length returned in C does not include the end-of-string
27
     character.
28
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
29
30
^{31}
     MPI_INFO_GET_NKEYS(info, nkeys)
32
       IN
                 info
33
                                              info object (handle)
34
       OUT
                 nkeys
                                              number of defined keys (integer)
35
36
     int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
37
38
     MPI_Info_get_nkeys(info, nkeys, ierror)
          TYPE(MPI_Info), INTENT(IN) ::
39
                                              info
40
          INTEGER, INTENT(OUT) :: nkeys
41
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
43
          INTEGER INFO, NKEYS, IERROR
44
45
          MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.
46
47
48
```

MPI_INFO	_GET_NTHKEY(info, n, key)		1
IN	info	info object (handle)	2
			3
IN	n	key number (integer)	4
OUT	key	key (string)	5
			6
int MPI_I	nfo_get_nthkey(MPI_Info i	nfo, int n, char *key)	7 8
MPI Info	get_nthkey(info, n, key,	ierror)	9
	MPI_Info), INTENT(IN) ::		10
INTEG	ER, INTENT(IN) :: n		11
CHARA	CTER(LEN=*), INTENT(OUT)	:: key	12
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	13
MPI INFO	GET_NTHKEY(INFO, N, KEY,	IERROR)	14
	ER INFO, N, IERROR		15
	CTER*(*) KEY		16
Thia f	unation noturns the ath define	d have in info Kava and numbered 0 N 1 where	17
		d key in info. Keys are numbered $0 \dots N - 1$ where GET_NKEYS. All keys between 0 and $N - 1$ are	18 19
	5	f a given key does not change as long as info is not	20
0	ith MPI_INFO_SET or MPI_II	0 0	21
			22
			23
MPI_INFO	_DUP(info, newinfo)		24
IN	info	info object (handle)	25
OUT	newinfo	info object (handle)	26
			27
int MPI_I	nfo_dup(MPI_Info info, MF	PI_Info *newinfo)	28 29
MPT Info	dup(info, newinfo, ierror	-)	30
	MPI_Info), INTENT(IN) ::		31
	MPI_Info), INTENT(OUT) ::		32
	ER, OPTIONAL, INTENT(OUT)		33
MDT TNEO	DUP(INFO, NEWINFO, IERROF		34
	ER INFO, NEWINFO, IERROR		35
			36
		ting info object, creating a new object, with the	37
same (key,	value) pairs and the same orde	ering of keys.	38 39
			40
MPI_INFO	_FREE(info)		41
INOUT	info	info object (handle)	42
			43
int MPI T	nfo_free(MPI_Info *info)		44
			45
	free(info, ierror)	inf.	46
	MPI_Info), INTENT(INOUT) ER, OPTIONAL, INTENT(OUT)		47
TNIEG	ER, UPIIUMAL, INIENI(UUI)	:: ierror	48

1	MPI_INFO_FREE(INFO, IERROR)
2	INTEGER INFO, IERROR
3	
4	This function frees info and sets it to MPI_INFO_NULL. The value of an info argument is
5	interpreted each time the info is passed to a routine. Changes to an info after return from
6	a routine do not affect that interpretation.
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31 32	
32	
34	
34	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

# Chapter 10

# **Process Creation and Management**

## 10.1 Introduction

MPI is primarily concerned with communication rather than process or resource management. However, it is necessary to address these issues to some degree in order to define a useful framework for communication. This chapter presents a set of MPI interfaces that allows for a variety of approaches to process management while placing minimal restrictions on the execution environment.

The MPI model for process creation allows both the creation of an initial set of processes related by their membership in a common MPI\_COMM\_WORLD and the creation and management of processes after an MPI application has been started. A major impetus for the latter form of process creation comes from the PVM [24] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

The MPI Forum decided not to address resource control because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers. Resource control can encompass a wide range of abilities, including adding and deleting nodes from a virtual parallel machine, reserving and scheduling resources, managing compute partitions of an MPP, and returning information about available resources. MPI assumes that resource control is provided externally — probably by computer vendors, in the case of tightly coupled systems, or by a third party software package when the environment is a cluster of workstations.

The reasons for including process management in MPI are both technical and practical. Important classes of message-passing applications require process control. These include task farms, serial applications with parallel modules, and problems that require a run-time assessment of the number and type of processes that should be started. On the practical side, users of workstation clusters who are migrating from PVM to MPI may be accustomed to using PVM's capabilities for process and resource management. The lack of these features would be a practical stumbling block to migration.

The following goals are central to the design of MPI process management:

- The MPI process model must apply to the vast majority of current parallel environments. These include everything from tightly integrated MPPs to heterogeneous networks of workstations.
- MPI must not take over operating system responsibilities. It should instead provide a

 $^{24}$ 

 $^{31}$ 

clean interface between an application and system software.

- MPI must guarantee communication determinism in the presense of dynamic processes, i.e., dynamic process management must not introduce unavoidable race conditions.
- MPI must not contain features that compromise performance.

The process management model addresses these issues in two ways. First, MPI remains primarily a communication library. It does not manage the parallel environment in which a parallel program executes, though it provides a minimal interface between an application and external resource and process managers.

Second, MPI maintains a consistent concept of a communicator, regardless of how its members came into existence. A communicator is never changed once created, and it is always created using deterministic collective operations.

10.2 The Dynamic Process Model

The dynamic process model allows for the creation and cooperative termination of processes after an MPI application has started. It provides a mechanism to establish communication between the newly created processes and the existing MPI application. It also provides a mechanism to establish communication between two existing MPI applications, even when one did not "start" the other.

22 23 24

32

33

40

41

42

43

44

45

46 47 48 10.2.1 Starting Processes

 $^{25}_{26}$  MPI applications may start new processes through an interface to an external process manager.

<sup>27</sup> MPI\_COMM\_SPAWN starts MPI processes and establishes communication with them,
 <sup>28</sup> returning an intercommunicator. MPI\_COMM\_SPAWN\_MULTIPLE starts several different
 <sup>30</sup> binaries (or the same binary with different arguments), placing them in the same
 <sup>31</sup> MPI\_COMM\_WORLD and returning an intercommunicator.

MPI uses the group abstraction to represent processes. A process is identified by a (group, rank) pair.

<sup>34</sup><sub>35</sub> 10.2.2 The Runtime Environment

The MPI\_COMM\_SPAWN and MPI\_COMM\_SPAWN\_MULTIPLE routines provide an interface between MPI and the *runtime environment* of an MPI application. The difficulty is that there is an enormous range of runtime environments and application requirements, and MPI must not be tailored to any particular one. Examples of such environments are:

• MPP managed by a batch queueing system. Batch queueing systems generally allocate resources before an application begins, enforce limits on resource use (CPU time, memory use, etc.), and do not allow a change in resource allocation after a job begins. Moreover, many MPPs have special limitations or extensions, such as a limit on the number of processes that may run on one processor, or the ability to gang-schedule processes of a parallel application.

9

10

11

12

13

14 15

16 17

18

19

20

21

1

- Network of workstations with PVM. PVM (Parallel Virtual Machine) allows a user to create a "virtual machine" out of a network of workstations. An application may extend the virtual machine or manage processes (create, kill, redirect output, etc.) through the PVM library. Requests to manage the machine or processes may be intercepted and handled by an external resource manager.
- Network of workstations managed by a load balancing system. A load balancing system may choose the location of spawned processes based on dynamic quantities, such as load average. It may transparently migrate processes from one machine to another when a resource becomes unavailable.
- Large SMP with Unix. Applications are run directly by the user. They are scheduled at a low level by the operating system. Processes may have special scheduling characteristics (gang-scheduling, processor affinity, deadline scheduling, processor locking, etc.) and be subject to OS resource limits (number of processes, amount of memory, etc.).

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide "operating system" services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.).

Complex interaction of an MPI application with its runtime environment should be done through an environment-specific API. An example of such an API would be the PVM task and machine management routines — pvm\_addhosts, pvm\_config, pvm\_tasks, etc., possibly modified to return an MPI (group, rank) when possible. A Condor or PBS API would be another possibility.

At some low level, obviously, MPI must be able to interact with the runtime system, but the interaction is not visible at the application level and the details of the interaction are not specified by the MPI standard.

In many cases, it is impossible to keep environment-specific information out of the MPI interface without seriously compromising MPI functionality. To permit applications to take advantage of environment-specific functionality, many MPI routines take an info argument that allows an application to specify environment-specific information. There is a tradeoff between functionality and portability: applications that make use of info are not portable.

MPI does not require the existence of an underlying "virtual machine" model, in which there is a consistent global view of an MPI application and an implicit "operating system" managing resources and processes. For instance, processes spawned by one task may not be visible to another; additional hosts added to the runtime environment by one process may not be visible in another process; tasks spawned by different processes may not be automatically distributed over available resources.

Interaction between MPI and the runtime environment is limited to the following areas:

- A process may start new processes with MPI\_COMM\_SPAWN and MPI\_COMM\_SPAWN\_MULTIPLE.
- When a process spawns a child process, it may optionally use an info argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.

375

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10 11

12

13

14

15

16 17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41 42

43

44

45

46

• An attribute MPI_UNIVERSE_SIZE (See Section $10.5.1$ ) on MPI_COMM_WORLD tells a
program how "large" the initial runtime environment is, namely how many processes
can usefully be started in all. One can subtract the size of MPI_COMM_WORLD from
this value to find out how many processes might usefully be started in addition to
those already running.

## 10.3 Process Manager Interface

10.3.1 Processes in MPI

A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a
 unique process but a process does not determine a unique (group, rank) pair, since a process
 may belong to several groups.

 $14 \\ 15$ 

19

20

21

22

23

8 9

10

### 10.3.2 Starting Processes and Establishing Communication

The following routine starts a number of MPI processes and establishes communication with them, returning an intercommunicator.

Advice to users. It is possible in MPI to start a static SPMD or MPMD application by first starting one process and having that process start its siblings with MPI\_COMM\_SPAWN. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI application. (*End of advice to users.*)

24 25 26

27

28

MPI\_COMM\_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array\_of\_errcodes)

00			
29 30	IN	command	name of program to be spawned (string, significant only at root)
31 32 33	IN	argv	arguments to command (array of strings, significant only at root)
34 35	IN	maxprocs	maximum number of processes to start (integer, sig- nificant only at root)
36 37 38 39	IN	info	a set of key-value pairs telling the runtime system where and how to start the processes (handle, signifi- cant only at root)
40 41	IN	root	rank of process in which previous arguments are examined (integer)
42 43	IN	comm	intracommunicator containing group of spawning processes (handle)
44 45 46	OUT	intercomm	intercommunicator between original group and the newly spawned group (handle)
40 47 48	Ουτ	array_of_errcodes	one code per process (array of integer)

```
1
int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
                                                                                   \mathbf{2}
              MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
                                                                                    3
              int array_of_errcodes[])
                                                                                   4
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
                                                                                    5
              array_of_errcodes, ierror)
                                                                                    6
    CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
                                                                                    7
    INTEGER, INTENT(IN) :: maxprocs, root
                                                                                    8
    TYPE(MPI_Info), INTENT(IN) ::
                                     info
                                                                                   9
    TYPE(MPI_Comm), INTENT(IN) ::
                                     comm
                                                                                   10
    TYPE(MPI_Comm), INTENT(OUT) ::
                                      intercomm
                                                                                   11
    INTEGER :: array_of_errcodes(*)
                                                                                   12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   13
                                                                                   14
MPI_COMM_SPAWN (COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
                                                                                   15
              ARRAY_OF_ERRCODES, IERROR)
                                                                                   16
    CHARACTER*(*) COMMAND, ARGV(*)
                                                                                   17
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
                                                                                   18
    IERROR
```

MPI\_COMM\_SPAWN tries to start maxprocs identical copies of the MPI program specified by command, establishing communication with them and returning an intercommunicator. The spawned processes are referred to as children. The children have their own MPI\_COMM\_WORLD, which is separate from that of the parents. MPI\_COMM\_SPAWN is collective over comm, and also may not return until MPI\_INIT has been called in the children. Similarly, MPI\_INIT in the children may not return until all parents have called MPI\_COMM\_SPAWN. In this sense, MPI\_COMM\_SPAWN in the parents and MPI\_INIT in the children form a collective operation over the union of parent and child processes. The intercommunicator returned by MPI\_COMM\_SPAWN contains the parent processes in the local group and the child processes in the remote group. The ordering of processes in the local and remote groups is the same as the ordering of the group of the comm in the parents and of MPI\_COMM\_WORLD of the children, respectively. This intercommunicator can be obtained in the children through the function MPI\_COMM\_GET\_PARENT.

Advice to users. An implementation may automatically establish communication before MPI\_INIT is called by the children. Thus, completion of MPI\_COMM\_SPAWN in the parent does not necessarily mean that MPI\_INIT has been called in the children (although the returned intercommunicator can be used immediately). (End of advice to users.)

The command argument The command argument is a string containing the name of a program to be spawned. The string is null-terminated in C. In Fortran, leading and trailing spaces are stripped. MPI does not specify how to find the executable or how the working directory is determined. These rules are implementation-dependent and should be appropriate for the runtime environment.

Advice to implementors. The implementation should use a natural rule for finding executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning

#### Unofficial Draft for Comment Only

19

20

21

22

23

24

25

26

27

28

29

30

 $^{31}$ 

32 33

34

35

36

37

38 39

40

41

42

43

 $44 \\ 45$ 

46

47

process, or might search the directories in a PATH environment variable as do Unix shells. An implementation on top of PVM would use PVM's rules for finding executables (usually in \$HOME/pvm3/bin/\$PVM\_ARCH). An MPI implementation running under POE on an IBM SP would use POE's method of finding executables. An implementation should document its rules for finding executables and determining working directories, and a high-quality implementation should give the user some control over these rules. (*End of advice to implementors.*)

- If the program named in command does not call MPI\_INIT, but instead forks a process
   that calls MPI\_INIT, the results are undefined. Implementations may allow this case to
   work but are not required to.
  - Advice to users. MPI does not say what happens if the program you start is a shell script and that shell script starts a program that calls MPI\_INIT. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. (End of advice to users.)
- 18 19

33 34

The argv argument argv is an array of strings containing arguments that are passed to the program. The first element of argv is the first argument passed to command, not, as is conventional in some contexts, the command itself. The argument list is terminated by NULL in C and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI\_ARGV\_NULL may be used in C and Fortran to indicate an empty argument list. In C this constant is the same as NULL.

```
Example 10.1 Examples of argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:
char command[] = "ocean";
char *argv[] = {"-gridfile", "ocean1.grd", NULL};
MPI_Comm_spawn(command, argv, ...);
```

```
or, if not everything is known at compile time:
```

```
35
             char *command;
36
             char **argv;
37
             command = "ocean";
38
             argv=(char **)malloc(3 * sizeof(char *));
39
             argv[0] = "-gridfile";
40
             argv[1] = "ocean1.grd";
41
             argv[2] = NULL;
42
             MPI_Comm_spawn(command, argv, ...);
43
44
     In Fortran:
45
46
47
48
```

1

2

3

4

5

6

7

8

12

13

14

15

16

CHARACTER*25 command, argv(3)	
command = ' ocean '	
<pre>argv(1) = ' -gridfile '</pre>	
argv(2) = ' ocean1.grd'	
argv(3) = ' '	
call MPI_COMM_SPAWN(command, argv,)	

Arguments are supplied to the program if this is allowed by the operating system. In C, the MPI\_COMM\_SPAWN argument argv differs from the argv argument of main in two respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the implementation and conventionally contains the name of the program (given by command). argv[1] of main corresponds to argv[0] in MPI\_COMM\_SPAWN, argv[2] of main to argv[1] of MPI\_COMM\_SPAWN, etc. Passing an argv of MPI\_ARGV\_NULL to MPI\_COMM\_SPAWN results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the name of the program. Second, argv of MPI\_COMM\_SPAWN must be null-terminated, so that its length can be determined.

If a Fortran implementation supplies routines that allow a program to obtain its arguments, the arguments may be available through that mechanism. In C, if the operating system does not support arguments appearing in **argv** of **main()**, the MPI implementation may add the arguments to the argv that is passed to MPI\_INIT.

The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn maxprocs processes, it raises an error of class MPI\_ERR\_SPAWN.

An implementation may allow the info argument to change the default behavior, such that if the implementation is unable to spawn all maxprocs processes, it may spawn a smaller number of processes instead of raising an error. In principle, the info argument may specify an arbitrary set  $\{m_i: 0 \leq m_i \leq \mathsf{maxprocs}\}\$  of allowed values for the number of processes spawned. The set  $\{m_i\}$  does not necessarily include the value maxprocs. If an implementation is able to spawn one of these allowed numbers of processes,

MPI\_COMM\_SPAWN returns successfully and the number of spawned processes, m, is given 30 by the size of the remote group of intercomm. If m is less than maxproc, reasons why the other processes were not spawned are given in array\_of\_errcodes as described below. If it is not possible to spawn one of the allowed numbers of processes, MPI\_COMM\_SPAWN raises an error of class MPI\_ERR\_SPAWN.

A spawn call with the default behavior is called *hard*. A spawn call for which fewer than maxprocs processes may be returned is called soft. See Section 10.3.4 for more information on the soft key for info.

Advice to users. By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior "spawn as many processes as possible, up to N," you should do a soft spawn, where the set of allowed values  $\{m_i\}$  is  $\{0 \dots N\}$ . 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 46handle of type MPI\_Info in C and Fortran with the mpi\_f08 module and 47INTEGER in Fortran with the mpi module or the include file mpif.h. It is a container for a 48

#### **Unofficial Draft for Comment Only**

8

9

10

11

12

13

14

15

16

17

18

19

2021

22

23

 $^{24}$ 

25

26

27

28

29

 $^{31}$ 

32

33

34

35

36

37 38

39

40

41

42

43

number of user-specified (key,value) pairs. key and value are strings (null-terminated char\*
 in C, character\*(\*) in Fortran). Routines to create and manipulate the info argument are described in Chapter 9.

For the SPAWN calls, info provides additional (and possibly implementation-dependent)
 instructions to MPI and the runtime system on how to start processes. An application may
 pass MPI\_INFO\_NULL in C or Fortran. Portable programs not requiring detailed control over
 process locations should use MPI\_INFO\_NULL.

<sup>8</sup> MPI does not specify the content of the info argument, except to reserve a number of <sup>9</sup> special key values (see Section 10.3.4). The info argument is quite flexible and could even <sup>10</sup> be used, for example, to specify the executable and its command-line arguments. In this <sup>11</sup> case the command argument to MPI\_COMM\_SPAWN could be empty. The ability to do this <sup>12</sup> follows from the fact that MPI does not specify how an executable is found, and the info <sup>13</sup> argument can tell the runtime system where to "find" the executable "" (empty string). Of <sup>14</sup> course a program that does this will not be portable across MPI implementations.

15

The root argument All arguments before the root argument are examined only on the
 process whose rank in comm is equal to root. The value of these arguments on other
 processes is ignored.

19

 $^{31}$ 

32

33 34 35

43

20The array\_of\_errcodes argument The array\_of\_errcodes is an array of length maxprocs in 21which MPI reports the status of each process that MPI was requested to start. If all maxprocs 22processes were spawned,  $array_of_errcodes$  is filled in with the value MPI\_SUCCESS. If only m23 $(0 \le m \le maxprocs)$  processes are spawned, m of the entries will contain MPI\_SUCCESS and  $^{24}$ the rest will contain an implementation-specific error code indicating the reason MPI could 25not start the process. MPI does not specify which entries correspond to failed processes. 26An implementation may, for instance, fill in error codes in one-to-one correspondence with 27a detailed specification in the info argument. These error codes all belong to the error class 28MPI\_ERR\_SPAWN if there was no error in the argument list. In C or Fortran, an application 29may pass MPI\_ERRCODES\_IGNORE if it is not interested in the error codes. 30

- Advice to implementors. MPI\_ERRCODES\_IGNORE in Fortran is a special type of constant, like MPI\_BOTTOM. See the discussion in Section 2.5.4. (End of advice to implementors.)
- <sup>36</sup> MPI\_COMM\_GET\_PARENT(parent)

parent

```
<sup>37</sup>
<sub>38</sub> OUT
```

the parent communicator (handle)

```
39
40 int MPI_Comm_get_parent(MPI_Comm *parent)
```

```
    MPI_Comm_get_parent(parent, ierror)
    TYPE(MPI_Comm) INTENT(OUT) ·· parent
```

```
TYPE(MPI_Comm), INTENT(OUT) :: parent
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
44
45 MPI_COMM_GET_PARENT(PARENT, IERROR)
```

```
45
46 INTEGER PARENT, IERROR
```

<sup>47</sup> If a process was started with MPI\_COMM\_SPAWN or MPI\_COMM\_SPAWN\_MULTIPLE,
 <sup>48</sup> MPI\_COMM\_GET\_PARENT returns the "parent" intercommunicator of the current process.

This parent intercommunicator is created implicitly inside of MPI\_INIT and is the same intercommunicator returned by SPAWN in the parents.

If the process was not spawned, MPI\_COMM\_GET\_PARENT returns MPI\_COMM\_NULL.

After the parent communicator is freed or disconnected, MPI\_COMM\_GET\_PARENT returns MPI\_COMM\_NULL.

Advice to users. MPI\_COMM\_GET\_PARENT returns a handle to a single intercommunicator. Calling MPI\_COMM\_GET\_PARENT a second time returns a handle to the same intercommunicator. Freeing the handle with MPI\_COMM\_DISCONNECT or MPI\_COMM\_FREE will cause other references to the intercommunicator to become invalid (dangling). Note that calling MPI\_COMM\_FREE on the parent communicator is not useful. (End of advice to users.)

*Rationale.* The desire of the Forum was to create a constant MPI\_COMM\_PARENT similar to MPI\_COMM\_WORLD. Unfortunately such a constant cannot be used (syntactically) as an argument to MPI\_COMM\_DISCONNECT, which is explicitly allowed. (*End of rationale.*)

### 10.3.3 Starting Multiple Executables and Establishing Communication

While MPI\_COMM\_SPAWN is sufficient for most cases, it does not allow the spawning of multiple binaries, or of the same binary with multiple sets of arguments. The following routine spawns multiple binaries or the same binary with multiple sets of arguments, establishing communication with them and placing them in the same MPI\_COMM\_WORLD.

 $\mathbf{2}$ 

 $\overline{7}$ 

#### 1MPI\_COMM\_SPAWN\_MULTIPLE(count, array\_of\_commands, array\_of\_argv, $\mathbf{2}$ array\_of\_maxprocs, array\_of\_info, root, comm, intercomm, array\_of\_errcodes) 3 4 IN count number of commands (positive integer, significant to 5MPI only at root — see advice to users) 6 IN array\_of\_commands programs to be executed (array of strings, significant 7 only at root) 8 9 IN array\_of\_argv arguments for commands (array of array of strings, 10 significant only at root) 11 IN array\_of\_maxprocs maximum number of processes to start for each com-12mand (array of integer, significant only at root) 13 IN array\_of\_info info objects telling the runtime system where and how 14to start processes (array of handles, significant only at 15root) 1617 IN rank of process in which previous arguments are exroot 18 amined (integer) 19 IN intracommunicator containing group of spawning procomm 20cesses (handle) 21OUT intercomm intercommunicator between original group and newly 22spawned group (handle) 2324OUT array\_of\_errcodes one error code per process (array of integer) 2526int MPI\_Comm\_spawn\_multiple(int count, char \*array\_of\_commands[], 27char \*\*array\_of\_argv[], const int array\_of\_maxprocs[], 28const MPI\_Info array\_of\_info[], int root, MPI\_Comm comm, 29 MPI\_Comm \*intercomm, int array\_of\_errcodes[]) 30 MPI\_Comm\_spawn\_multiple(count, array\_of\_commands, array\_of\_argv, $^{31}$ array\_of\_maxprocs, array\_of\_info, root, comm, intercomm, 32 array\_of\_errcodes, ierror) 33 INTEGER, INTENT(IN) :: count, array\_of\_maxprocs(\*), root 34 CHARACTER(LEN=\*), INTENT(IN) :: array\_of\_commands(\*) 35 CHARACTER(LEN=\*), INTENT(IN) :: array\_of\_argv(count, \*) 36 TYPE(MPI\_Info), INTENT(IN) :: array\_of\_info(\*) 37 TYPE(MPI\_Comm), INTENT(IN) :: comm 38 TYPE(MPI\_Comm), INTENT(OUT) :: intercomm 39 INTEGER :: array\_of\_errcodes(\*) 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42MPI\_COMM\_SPAWN\_MULTIPLE(COUNT, ARRAY\_OF\_COMMANDS, ARRAY\_OF\_ARGV, 43 ARRAY\_OF\_MAXPROCS, ARRAY\_OF\_INFO, ROOT, COMM, INTERCOMM, 44ARRAY\_OF\_ERRCODES, IERROR) 45INTEGER COUNT, ARRAY\_OF\_INFO(\*), ARRAY\_OF\_MAXPROCS(\*), ROOT, COMM, 46INTERCOMM, ARRAY\_OF\_ERRCODES(\*), IERROR 47 CHARACTER\*(\*) ARRAY\_OF\_COMMANDS(\*), ARRAY\_OF\_ARGV(COUNT, \*) 48

MPI\_COMM\_SPAWN\_MULTIPLE is identical to MPI\_COMM\_SPAWN except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI\_COMM\_SPAWN. For the Fortran version of array\_of\_argv, the element array\_of\_argv(i,j) is the j-th argument to command number i.

*Rationale.* This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI\_COMM\_SPAWN to sort out arguments. Note that the leading dimension of array\_of\_argv must be the same as count. Also note that Fortran rules for sequence association allow a different value in the first dimension; in this case, the sequence of array elements is interpreted by MPI\_COMM\_SPAWN\_MULTIPLE as if the sequence is stored in an array defined with the first dimension set to count. This Fortran feature allows an implementor to define MPI\_ARGVS\_NULL (see below) with fixed dimensions, e.g., (1,1), or only with one dimension, e.g., (1). (End of rationale.)

Advice to users. The argument count is interpreted by MPI only at the root, as is array\_of\_argv. Since the leading dimension of array\_of\_argv is count, a non-positive value of count at a non-root node could theoretically cause a runtime bounds check error, even though array\_of\_argv should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of count on the non-root nodes. (*End of advice to users.*)

In any language, an application may use the constant MPI\_ARGVS\_NULL (which is likely to be (char \*\*\*)0 in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of array\_of\_argv to MPI\_ARGV\_NULL is not defined. To specify arguments for some commands but not others, the commands without arguments should have a corresponding argv whose first element is null ((char \*)0 in C and empty string in Fortran). In Fortran at non-root processes, the count argument must be set to a value that is consistent with the provided array\_of\_argv although the content of these arguments has no meaning for this operation.

All of the spawned processes have the same MPI\_COMM\_WORLD. Their ranks in MPI\_COMM\_WORLD correspond directly to the order in which the commands are specified in MPI\_COMM\_SPAWN\_MULTIPLE. Assume that  $m_1$  processes are generated by the first command,  $m_2$  by the second, etc. The processes corresponding to the first command have ranks  $0, 1, \ldots, m_1-1$ . The processes in the second command have ranks  $m_1, m_1+1, \ldots, m_1+m_2-1$ . The processes in the third have ranks  $m_1 + m_2, m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 - 1$ , etc.

Advice to users. Calling MPI\_COMM\_SPAWN multiple times would create many sets of children with different MPI\_COMM\_WORLDs whereas

MPI\_COMM\_SPAWN\_MULTIPLE creates children with a single MPI\_COMM\_WORLD, so the two methods are not completely equivalent. There are also two performancerelated reasons why, if you need to spawn multiple executables, you may want to use MPI\_COMM\_SPAWN\_MULTIPLE instead of calling MPI\_COMM\_SPAWN several times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes spawned at the same time may be faster than communication between processes spawned separately. (End of advice to users.)

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

```
The array_of_errcodes argument is a 1-dimensional array of size \sum_{i=1}^{count} n_i, where n_i is
1
\mathbf{2}
     the i-th element of array_of_maxprocs. Command number i corresponds to the n_i contiguous
     slots in this array from element \sum_{j=1}^{i-1} n_j to \left[\sum_{j=1}^{i} n_j\right] - 1. Error codes are treated as for
3
4
     MPI_COMM_SPAWN.
5
6
     Example 10.2 Examples of array_of_argv in C and Fortran
7
     To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" and the program
8
      "atmos" with argument "atmos.grd" in C:
9
              char *array_of_commands[2] = {"ocean", "atmos"};
10
              char **array_of_argv[2];
11
              char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
12
              char *argv1[] = {"atmos.grd", (char *)0};
13
              array_of_argv[0] = argv0;
14
              array_of_argv[1] = argv1;
15
              MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
16
17
     Here is how you do it in Fortran:
18
19
              CHARACTER*25 commands(2), array_of_argv(2, 3)
              commands(1) = ' ocean '
20
21
              array_of_argv(1, 1) = ' -gridfile '
22
              array_of_argv(1, 2) = ' ocean1.grd'
              array_of_argv(1, 3) = ', '
23
^{24}
25
              commands(2) = ' atmos '
26
              array_of_argv(2, 1) = ' atmos.grd '
27
              array_of_argv(2, 2) = ', '
28
              call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
29
30
31
     10.3.4 Reserved Keys
32
     The following keys are reserved. An implementation is not required to interpret these keys,
33
     but if it does interpret the key, it must provide the functionality described.
34
35
     host Value is a hostname. The format of the hostname is determined by the implementation.
36
37
     arch Value is an architecture name. Valid architecture names and what they mean are
38
           determined by the implementation.
39
     wdir Value is the name of a directory on a machine on which the spawned process(es)
40
           execute(s). This directory is made the working directory of the executing process(es).
41
           The format of the directory name is determined by the implementation.
42
43
     path Value is a directory or set of directories where the implementation should look for the
44
           executable. The format of path is determined by the implementation.
45
46
     file Value is the name of a file in which additional information is specified. The format of
47
           the filename and internal format of the file are determined by the implementation.
48
```

soft Value specifies a set of numbers which are allowed values for the number of processes that MPI\_COMM\_SPAWN (et al.) may create. The format of the value is a comma-separated list of Fortran-90 triplets each of which specifies a set of integers and which together specify the set formed by the union of these sets. Negative values in this set and values greater than maxprocs are ignored. MPI will spawn the largest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant.

By Fortran-90 triplets, we mean:

by Forman-90 mplets, we mean.	9
1. a means $a$	10
2. <b>a:b</b> means $a, a + 1, a + 2,, b$	11
3. a:b:c means $a, a + c, a + 2c,, a + ck$ , where for $c > 0$ , k is the largest integer for which $a + ck \le b$ and for $c < 0$ , k is the largest integer for which $a + ck \ge b$ . If $b > a$ then c must be positive. If $b < a$ then c must be negative.	12 13 14 15
Examples:	16
1. <b>a:b</b> gives a range between $a$ and $b$	17 18
2. 0:N gives full "soft" functionality	19
3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows a power-of-two number of processes.	20 21
4. 2:10000:2 allows an even number of processes.	22
	23
5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.	24 25
10.3.5 Spawn Example	26
	27
Manager-worker Example Using MPI_COMM_SPAWN	28
/* manager */	29
#include "mpi.h"	30
<pre>int main(int argc, char *argv[])</pre>	31
{	32
<pre>int world_size, universe_size, *universe_sizep, flag;</pre>	33 34
MPI_Comm everyone; /* intercommunicator */	34
char worker_program[100];	36
MDI Init (kongo kongr).	37
MPI_Init(&argc, &argv); MPI_Comm_size(MPI_COMM_WORLD, &world_size);	38
Mri_comm_Size(Mri_comm_world, awoiid_Size),	39
if (world_size != 1) error("Top heavy with management");	40
ii (world_Size := 1) error( rop heavy with management );	41
MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,	42
<pre>kuniverse_sizep, &amp;flag);</pre>	43
if (!flag) {	44
printf("This MPI does not support UNIVERSE_SIZE. How many\n\	45
processes total?");	46
<pre>scanf("%d", &amp;universe_size);</pre>	47

} else universe\_size = \*universe\_sizep;

**Unofficial Draft for Comment Only** 

 $1 \\ 2$ 

3

4

5

6 7

8

```
1
        if (universe_size == 1) error("No room to start workers");
2
3
        /*
4
         * Now spawn the workers. Note that there is a run-time determination
5
         * of what type of worker to spawn, and presumably this calculation must
6
         * be done at run time and cannot be calculated before starting
7
         * the program. If everything is known when the application is
8
         * first started, it is generally better to start them all at once
9
         * in a single MPI_COMM_WORLD.
10
         */
11
12
        choose_worker_program(worker_program);
13
        MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
14
                  MPI_INFO_NULL, 0, MPI_COMM_SELF, & everyone,
15
                  MPI_ERRCODES_IGNORE);
16
        /*
17
         * Parallel code here. The communicator "everyone" can be used
18
         * to communicate with the spawned processes, which have ranks 0,...
19
         * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
20
         * "everyone".
21
         */
22
23
        MPI_Finalize();
^{24}
        return 0;
25
     }
26
     /* worker */
27
28
     #include "mpi.h"
29
     int main(int argc, char *argv[])
30
^{31}
     ſ
32
        int size;
33
        MPI_Comm parent;
34
        MPI_Init(&argc, &argv);
        MPI_Comm_get_parent(&parent);
35
        if (parent == MPI_COMM_NULL) error("No parent!");
36
        MPI_Comm_remote_size(parent, &size);
37
        if (size != 1) error("Something's wrong with the parent");
38
39
        /*
40
41
         * Parallel code here.
42
         * The manager is represented as the process with rank 0 in (the remote
         * group of) the parent communicator. If the workers need to communicate
43
         * among themselves, they can use MPI_COMM_WORLD.
44
         */
45
46
47
        MPI_Finalize();
48
        return 0;
```

}

## 10.4 Establishing Communication

This section provides functions that establish communication between two sets of MPI processes that do not share a communicator.

Some situations in which these functions are useful are:

- 1. Two parts of an application that are started independently need to communicate.
- 2. A visualization tool wants to attach to a running process.
- 3. A server wants to accept connections from multiple clients. Both clients and server may be parallel programs.

In each of these situations, MPI must establish communication channels where none existed before, and there is no parent/child relationship. The routines described in this section establish communication between the two sets of processes by creating an MPI intercommunicator, where the two groups of the intercommunicator are the original sets of processes.

Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. One group of processes indicates its willingness to accept connections from other groups of processes. We will call this group the (parallel) *server*, even if this is not a client/server type of application. The other group connects to the server; we will call it the *client*.

Advice to users. While the names *client* and *server* are used throughout this section, MPI does not guarantee the traditional robustness of client/server systems. The functionality described in this section is intended to allow two cooperating parts of the same application to communicate with one another. For instance, a client that gets a segmentation fault and dies, or one that does not participate in a collective operation may cause a server to crash or hang. (*End of advice to users.*)

#### 10.4.1 Names, Addresses, Ports, and All That

Almost all of the complexity in MPI client/server routines addresses the question "how does the client find out how to contact the server?" The difficulty, of course, is that there is no existing communication channel between them, yet they must somehow agree on a rendezvous point where they will establish communication.

Agreeing on a rendezvous point always involves a third party. The third party may itself provide the rendezvous point or may communicate rendezvous information from server to client. Complicating matters might be the fact that a client does not really care what server it contacts, only that it be able to get in touch with one that can handle its request.

Ideally, MPI can accommodate a wide variety of run-time systems while retaining the ability to write simple, portable code. The following should be compatible with MPI:

- The server resides at a well-known internet address host:port.
- The server prints out an address to the terminal; the user gives this address to the client program.

#### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

 $\frac{44}{45}$ 

388

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

25

30

31

32

33

36 37

- The server places the address information on a nameserver, where it can be retrieved with an agreed-upon name.
- The server to which the client connects is actually a broker, acting as a middleman between the client and the real server.

MPI does not require a nameserver, so not all implementations will be able to support all of the above scenarios. However, MPI provides an optional nameserver interface, and is compatible with external name servers.

A port\_name is a *system-supplied* string that encodes a low-level network address at which a server can be contacted. Typically this is an IP address and a port number, but an implementation is free to use any protocol. The server establishes a port\_name with the MPI\_OPEN\_PORT routine. It accepts a connection to a given port with MPI\_COMM\_ACCEPT. A client uses port\_name to connect to the server.

By itself, the port\_name mechanism is completely portable, but it may be clumsy to use because of the necessity to communicate port\_name to the client. It would be more convenient if a server could specify that it be known by an *application-supplied* service\_name so that the client could connect to that service\_name without knowing the port\_name.

An MPI implementation may allow the server to publish a (port\_name, service\_name) pair with MPI\_PUBLISH\_NAME and the client to retrieve the port name from the service name with MPI\_LOOKUP\_NAME. This allows three levels of portability, with increasing levels of functionality.

- Applications that do not rely on the ability to publish names are the most portable.
   Typically the port\_name must be transferred "by hand" from server to client.
- 2. Applications that use the MPI\_PUBLISH\_NAME mechanism are completely portable 27 among implementations that provide this service. To be portable among all imple-28 mentations, these applications should have a fall-back mechanism that can be used 29 when names are not published.
  - 3. Applications may ignore MPI's name publishing functionality and use their own mechanism (possibly system-supplied) to publish names. This allows arbitrary flexibility but is not portable.
- <sup>34</sup><sub>35</sub> 10.4.2 Server Routines

A server makes itself available with two routines. First it must call MPI\_OPEN\_PORT to establish a port at which it may be contacted. Secondly it must call MPI\_COMM\_ACCEPT to accept connections from clients.

39 40 41

38

MPI\_OPEN\_PORT(info, port\_name)

42	IN	info	implementation-specific information on how to estab-
43			lish an address (handle)
44	OUT	port_name	newly established port (string)
45	001	port_name	nowly obtablished port (bering)
46	· · NDT O		
47	int MPI_0]	pen_port(MPI_Info info, c	char *port_name)

<sup>&</sup>lt;sup>48</sup> MPI\_Open\_port(info, port\_name, ierror)

```
TYPE(MPI_Info), INTENT(IN) :: info
CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
CHARACTER*(*) PORT_NAME
INTEGER INFO, IERROR
```

This function establishes a network address, encoded in the port\_name string, at which the server will be able to accept connections from clients. port\_name is supplied by the system, possibly using information in the info argument.

MPI copies a system-supplied port name into port\_name. port\_name identifies the newly opened port and can be used by a client to contact the server. The maximum size string that may be supplied by the system is MPI\_MAX\_PORT\_NAME.

Advice to users. The system copies the port name into port\_name. The application must pass a buffer of sufficient size to hold this value. (End of advice to users.)

port\_name is essentially a network address. It is unique within the communication universe to which it belongs (determined by the implementation), and may be used by any client within that communication universe. For instance, if it is an internet (host:port) address, it will be unique on the internet. If it is a low level switch address on an IBM SP, it will be unique to that SP.

Advice to implementors. These examples are not meant to constrain implementations. A port\_name could, for instance, contain a user name or the name of a batch job, as long as it is unique within some well-defined communication domain. The larger the communication domain, the more useful MPI's client/server functionality will be. (End of advice to implementors.)

The precise form of the address is implementation-defined. For instance, an internet address may be a host name or IP address, or anything that the implementation can decode into an IP address. A port name may be reused after it is freed with MPI\_CLOSE\_PORT and released by the system.

Advice to implementors. Since the user may type in port\_name by hand, it is useful to choose a form that is easily readable and does not have embedded spaces. (End of advice to implementors.)

info may be used to tell the implementation how to establish the address. It may, and usually will, be MPI\_INFO\_NULL in order to get the implementation defaults.

a port (string)

MPI\_CLOSE\_PORT(port\_name) IN port\_name

int MPI\_Close\_port(const char \*port\_name)

```
MPI_Close_port(port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Unofficial Draft for Comment Only

1

2

3

4

5

6

7 8

9

10

11

12

13

14

15

16 17

18

19

20

21

22

23

 $^{24}$ 

25

26

27 28

29

30

31

32 33

34

35

36 37

38

39 40

41 42

43 44

45 46

47

```
1
     MPI_CLOSE_PORT(PORT_NAME, IERROR)
\mathbf{2}
          CHARACTER*(*) PORT_NAME
3
          INTEGER IERROR
4
     This function releases the network address represented by port_name.
5
6
\overline{7}
     MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
8
       IN
                 port_name
                                              port name (string, used only on root)
9
10
       IN
                 info
                                              implementation-dependent information (handle, used
11
                                              only on root)
12
       IN
                 root
                                              rank in comm of root node (integer)
13
       IN
                 comm
                                              intracommunicator over which call is collective (han-
14
                                              dle)
15
16
       OUT
                 newcomm
                                              intercommunicator with client as remote group (han-
17
                                              dle)
18
19
     int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
20
                     MPI_Comm comm, MPI_Comm *newcomm)
21
     MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
22
          CHARACTER(LEN=*), INTENT(IN) :: port_name
23
          TYPE(MPI_Info), INTENT(IN) :: info
24
          INTEGER, INTENT(IN) :: root
25
          TYPE(MPI_Comm), INTENT(IN) ::
                                              comm
26
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
30
          CHARACTER*(*) PORT_NAME
^{31}
          INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
32
          MPI_COMM_ACCEPT establishes communication with a client. It is collective over the
33
     calling communicator. It returns an intercommunicator that allows communication with the
34
     client.
35
          The port_name must have been established through a call to MPI_OPEN_PORT.
36
          info can be used to provide directives that may influence the behavior of the ACCEPT
37
     call.
38
39
     10.4.3 Client Routines
40
41
     There is only one routine on the client side.
42
43
44
45
46
47
48
```

1011 1		, root, comm, newcomm)	
IN	port_name	network address (string, used only on root)	2
IN	info	implementation-dependent information (handle, used	3 4
		only on root)	4 5
IN	root	rank in <b>comm</b> of root node (integer)	6
		( <sup>(</sup> ),	7
IN	comm	intracommunicator over which call is collective (han-	8
		dle)	9
OU	T newcomm	intercommunicator with server as remote group (han-	10
		dle)	11
			12
int	-	port_name, MPI_Info info, int root,	13
	MPI_Comm comm, MPI_C	omm *newcomm)	14
MPI_	<pre>Comm_connect(port_name, info,</pre>	root, comm, newcomm, ierror)	15
	CHARACTER(LEN=*), INTENT(IN)	:: port_name	16 17
	TYPE(MPI_Info), INTENT(IN) :: info		
	INTEGER, INTENT(IN) :: root		18 19
	TYPE(MPI_Comm), INTENT(IN) ::		20
	TYPE(MPI_Comm), INTENT(OUT) :		21
	INTEGER, OPTIONAL, INTENT(OUT)	) :: lerror	22
MPI_	COMM_CONNECT(PORT_NAME, INFO,	ROOT, COMM, NEWCOMM, IERROR)	23
	CHARACTER*(*) PORT_NAME		24
	INTEGER INFO, ROOT, COMM, NEW	COMM, IERROR	25
	This routine establishes communic	ation with a server specified by port_name. It is	26
		r and returns an intercommunicator in which the	27

This routine establishes communication with a server specified by port\_name. It is collective over the calling communicator and returns an intercommunicator in which the remote group participated in an MPI\_COMM\_ACCEPT.

If the named port does not exist (or has been closed), MPI\_COMM\_CONNECT raises an error of class MPI\_ERR\_PORT.

If the port exists, but does not have a pending MPI\_COMM\_ACCEPT, the connection attempt will eventually time out after an implementation-defined time, or succeed when the server calls MPI\_COMM\_ACCEPT. In the case of a time out, MPI\_COMM\_CONNECT raises an error of class MPI\_ERR\_PORT.

Advice to implementors. The time out period may be arbitrarily short or long. However, a high-quality implementation will try to queue connection attempts so that a server can handle simultaneous requests from several clients. A high-quality implementation may also provide a mechanism, through the info arguments to MPI\_OPEN\_PORT, MPI\_COMM\_ACCEPT, and/or MPI\_COMM\_CONNECT, for the user to control timeout and queuing behavior. (*End of advice to implementors.*)

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order they were initiated and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

port\_name is the address of the server. It must be the same as the name returned by MPI\_OPEN\_PORT on the server. Some freedom is allowed here. If there are equivalent

#### Unofficial Draft for Comment Only

1

 $44 \\ 45$ 

46

47

48

28

29

30

 $^{31}$ 

32

33

34

35 36

37

38

39

40

41

forms of port\_name, an implementation may accept them as well. For instance, if port\_name is (hostname:port), an implementation may accept (ip\_address:port) as well.

2 3 4

5

1

```
10.4.4 Name Publishing
```

The routines in this section provide a mechanism for publishing names. A (service\_name, 6 port\_name) pair is published by the server, and may be retrieved by a client using the 7 service\_name only. An MPI implementation defines the scope of the service\_name, that 8 is, the domain over which the service\_name can be retrieved. If the domain is the empty 9 set, that is, if no client can retrieve the information, then we say that name publishing 10 is not supported. Implementations should document how the scope is determined. High-11 quality implementations will give some control to users through the info arguments to name 12publishing functions. Examples are given in the descriptions of individual functions. 13

14

```
15
16
```

33

MPI\_PUBLISH\_NAME(service\_name, info, port\_name)

17	IN	service_name	a service name to associate with the port (string)
18	IN	info	implementation-specific information (handle)
19 20	IN	port_name	a port name (string)
20 21 22 23	<pre>int MPI_Publish_name(const char *service_name, MPI_Info info,</pre>		
24 25 26 27 28	TYPE CHAR	ish_name(service_name, in: (MPI_Info), INTENT(IN) :: ACTER(LEN=*), INTENT(IN) GER, OPTIONAL, INTENT(OUT)	info :: service_name, port_name
29 30 31	INTE	ISH_NAME(SERVICE_NAME, IN GER INFO, IERROR ACTER*(*) SERVICE_NAME, P(	
32			

This routine publishes the pair (port\_name, service\_name) so that an application may retrieve a system-supplied port\_name using a well-known service\_name.

The implementation must define the *scope* of a published service name, that is, the domain over which the service name is unique, and conversely, the domain over which the (port name, service name) pair may be retrieved. For instance, a service name may be unique to a job (where job is defined by a distributed operating system or batch scheduler), unique to a machine, or unique to a Kerberos realm. The scope may depend on the info argument to MPI\_PUBLISH\_NAME.

<sup>40</sup> MPI permits publishing more than one service\_name for a single port\_name. On the <sup>41</sup> other hand, if service\_name has already been published within the scope determined by info, <sup>42</sup> the behavior of MPI\_PUBLISH\_NAME is undefined. An MPI implementation may, through <sup>43</sup> a mechanism in the info argument to MPI\_PUBLISH\_NAME, provide a way to allow multiple <sup>44</sup> servers with the same service in the same scope. In this case, an implementation-defined <sup>45</sup> policy will determine which of several port names is returned by MPI\_LOOKUP\_NAME.

<sup>46</sup> Note that while service\_name has a limited scope, determined by the implementation,
 <sup>47</sup> port\_name always has global scope within the communication universe used by the imple-

MPI\_UNPUBLISH\_NAME(service\_name, info, port\_name)

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.*)

19
20
21

IN	service_name	a service name (string)	2	
IN	info	implementation-specific information (handle)	2	
IN	port_name	a port name (string)	2	
int MPI_U	npublish_name(const char const char *port_name	*service_name, MPI_Info info, e)	2	
<pre>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</pre>				
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) INTEGER INFO, IERROR CHARACTER*(*) SERVICE_NAME, PORT_NAME				

This routine unpublishes a service name that has been previously published. Attempting to unpublish a name that has not been published or has already been unpublished is erroneous and is indicated by the error class MPI\_ERR\_SERVICE.

All published names must be unpublished before the corresponding port is closed and before the publishing process exits. The behavior of MPI\_UNPUBLISH\_NAME is implementation dependent when a process tries to unpublish a name that it did not publish.

If the info argument was used with MPI\_PUBLISH\_NAME to tell the implementation 45 how to publish names, the implementation may require that info passed to 46 MPI\_UNPUBLISH\_NAME contain information to tell the implementation how to unpublish 47 a name. 48

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

1 MPI\_LOOKUP\_NAME(service\_name, info, port\_name) 2 IN service\_name a service name (string) 3 IN info implementation-specific information (handle) 4 5OUT port\_name a port name (string) 6  $\overline{7}$ int MPI\_Lookup\_name(const char \*service\_name, MPI\_Info info, 8 char \*port\_name) 9 MPI\_Lookup\_name(service\_name, info, port\_name, ierror) 10 CHARACTER(LEN=\*), INTENT(IN) :: service\_name 11 TYPE(MPI\_Info), INTENT(IN) :: info 12CHARACTER(LEN=MPI\_MAX\_PORT\_NAME), INTENT(OUT) :: port\_name 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI\_LOOKUP\_NAME(SERVICE\_NAME, INFO, PORT\_NAME, IERROR) 16CHARACTER\*(\*) SERVICE\_NAME, PORT\_NAME 17 INTEGER INFO, IERROR 18 This function retrieves a port\_name published by MPI\_PUBLISH\_NAME with 19service\_name. If service\_name has not been published, it raises an error in the error class 20MPI\_ERR\_NAME. The application must supply a port\_name buffer large enough to hold the 21largest possible port name (see discussion above under MPI\_OPEN\_PORT). 22If an implementation allows multiple entries with the same service\_name within the 23same scope, a particular **port\_name** is chosen in a way determined by the implementation.  $^{24}$ If the info argument was used with MPI\_PUBLISH\_NAME to tell the implementation 2526how to publish names, a similar info argument may be required for MPI\_LOOKUP\_NAME. 27Reserved Key Values 2810.4.5 29The following key values are reserved. An implementation is not required to interpret these 30 key values, but if it does interpret the key value, it must provide the functionality described.  $^{31}$ 32 ip\_port Value contains IP port number at which to establish a port. (Reserved for 33 MPI\_OPEN\_PORT only). 3435 ip\_address Value contains IP address at which to establish a port. If the address is not a 36 valid IP address of the host on which the MPI\_OPEN\_PORT call is made, the results 37 are undefined. (Reserved for MPI\_OPEN\_PORT only). 38 39 10.4.6 Client/Server Examples 40Simplest Example — Completely Portable. 41 42The following example shows the simplest way to use the client/server interface. It does 43 not use service names at all. 44On the server side: 454647 char myport[MPI\_MAX\_PORT\_NAME]; 48 MPI\_Comm intercomm;

/* */	1		
<pre>MPI_Open_port(MPI_INFO_NULL, myport);</pre>			
<pre>printf("port name is: %s\n", myport);</pre>	3		
	4		
<pre>MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &amp;intercomm);</pre>	5 6		
/* do something with intercomm */	7		
The server prints out the port name to the terminal and the user must type it in when starting up the client (assuming the MPI implementation supports stdin such that this			
			works). On the client side:
MDT Commentations and a	11		
<pre>MPI_Comm intercomm; char name[MPI_MAX_PORT_NAME];</pre>	12		
printf("enter port name: ");	13		
gets(name);	14		
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);	15		
	16		
Ocean/Atmosphere — Relies on Name Publishing	17		
	18		
In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere	19 20		
climate model. It assumes that the MPI implementation publishes names.	20 21		
	21		
<pre>MPI_Open_port(MPI_INFO_NULL, port_name);</pre>	23		
MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);	24		
	25		
<pre>MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &amp;intercomm);</pre>	26		
<pre>/* do something with intercomm */</pre>	27		
<pre>MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);</pre>	28		
	29		
On the direct side	30		
On the client side:	31		
<pre>MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);</pre>	32		
MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,	33 34		
<pre>&amp;intercomm);</pre>	35		
	36		
Simple Client-Server Example	37		
This is a simple example; the server accepts only a single connection at a time and serves	38		
that connection until the client requests to be disconnected. The server is a single process.	39		
Here is the server. It accepts a single connection and then processes data until it	40		
receives a message with tag 1. A message with tag 0 tells the server to exit.	41		
	42		
<pre>#include "mpi.h"</pre>	43		
int main(int argc, char *argv[])			
{ MDI Comm alignt:	45		
MPI_Comm client; MPI_Status status;	46		
char port_name[MPI_MAX_PORT_NAME];	47		
F	48		

```
1
         double buf[MAX_DATA];
\mathbf{2}
                 size, again;
         int
3
4
         MPI_Init(&argc, &argv);
5
         MPI_Comm_size(MPI_COMM_WORLD, &size);
6
         if (size != 1) error(FATAL, "Server too big");
7
         MPI_Open_port(MPI_INFO_NULL, port_name);
8
         printf("server available at %s\n", port_name);
9
         while (1) {
10
              MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
11
                                 &client);
12
              again = 1;
13
              while (again) {
14
                  MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
15
                             MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
                  switch (status.MPI_TAG) {
16
17
                       case 0: MPI_Comm_free(&client);
18
                                MPI_Close_port(port_name);
19
                                MPI_Finalize();
20
                                return 0;
21
                       case 1: MPI_Comm_disconnect(&client);
22
                                again = 0;
23
                                break;
24
                       case 2: /* do something */
25
                       . . .
26
                       default:
27
                                /* Unexpected message type */
28
                                MPI_Abort(MPI_COMM_WORLD, 1);
29
                       }
30
                  }
^{31}
              }
32
     }
33
         Here is the client.
34
35
     #include "mpi.h"
36
     int main( int argc, char **argv )
37
     {
38
         MPI_Comm server;
39
         double buf[MAX_DATA];
40
         char port_name[MPI_MAX_PORT_NAME];
41
42
         MPI_Init( &argc, &argv );
43
         strcpy( port_name, argv[1] );/* assume server's name is cmd-line arg */
44
45
         MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
46
                             &server );
47
48
```

```
while (!done) {
    tag = 2; /* Action to perform */
    MPI_Send( buf, n, MPI_DOUBLE, 0, tag, server );
    /* etc */
    }
MPI_Send( buf, 0, MPI_DOUBLE, 0, 1, server );
MPI_Comm_disconnect( &server );
MPI_Finalize();
return 0;
```

#### 10.5 Other Functionality

#### Universe Size 10.5.1

}

Many "dynamic" MPI applications are expected to exist in a static runtime environment, in which resources have been allocated before the application is run. When a user (or possibly a batch system) runs one of these quasi-static applications, she will usually specify a number of processes to start and a total number of processes that are expected. An application simply needs to know how many slots there are, i.e., how many processes it should spawn.

MPI provides an attribute on MPI\_COMM\_WORLD, MPI\_UNIVERSE\_SIZE, that allows 22 the application to obtain this information in a portable manner. This attribute indicates 23the total number of processes that are expected. In Fortran, the attribute is the integer  $^{24}$ value. In C, the attribute is a pointer to the integer value. An application typically subtracts 25the size of MPI\_COMM\_WORLD from MPI\_UNIVERSE\_SIZE to find out how many processes it 26should spawn. MPI\_UNIVERSE\_SIZE is initialized in MPI\_INIT and is not changed by MPI. If 27defined, it has the same value on all processes of MPI\_COMM\_WORLD. MPI\_UNIVERSE\_SIZE 28is determined by the application startup mechanism in a way not specified by MPI. (The 29size of MPI\_COMM\_WORLD is another example of such a parameter.)

Possibilities for how MPI\_UNIVERSE\_SIZE might be set include

- A -universe\_size argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.
- An environment variable set by the user.
- Extra information passed to MPI\_COMM\_SPAWN through the info argument.

An implementation must document how MPI\_UNIVERSE\_SIZE is set. An implementation may not support the ability to set MPI\_UNIVERSE\_SIZE, in which case the attribute MPI\_UNIVERSE\_SIZE is not set.

MPI\_UNIVERSE\_SIZE is a recommendation, not necessarily a hard limit. For instance, 43 44some implementations may allow an application to spawn 50 processes per processor, if they are requested. However, it is likely that the user only wants to spawn one process per processor.

47MPI\_UNIVERSE\_SIZE is assumed to have been specified when an application was started, 48 and is in essence a portable mechanism to allow the user to pass to the application (through

### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

5 6

7 8

9

10 11 12

13 14

15

16

17

18

19

20

21

30

3132

33 34

35

36

37 38

39

40 41

42

the MPI process startup mechanism, such as mpiexec) a piece of critical runtime information. Note that no interaction with the runtime environment is required. If the runtime environment changes size while an application is running, MPI\_UNIVERSE\_SIZE is not updated, and the application must find out about the change through direct communication with the runtime system.

6 7

8

# 10.5.2 Singleton MPI\_INIT

A high-quality implementation will allow any process (including those not started with a "parallel application" mechanism) to become an MPI process by calling MPI\_INIT. Such a process can then connect to other MPI processes using the MPI\_COMM\_ACCEPT and MPI\_COMM\_CONNECT routines, or spawn other MPI processes. MPI does not mandate this behavior, but strongly encourages it where technically feasible.

- Advice to implementors. To start MPI processes belonging to the same
   MPI\_COMM\_WORLD requires some special coordination. The processes must be started
   at the "same" time, they must have a mechanism to establish communication, etc.
   Either the user or the operating system must take special steps beyond simply starting
   processes.
- When an application enters MPI\_INIT, clearly it must be able to determine if these special steps were taken. If a process enters MPI\_INIT and determines that no special steps were taken (i.e., it has not been given the information to form an MPI\_COMM\_WORLD with other processes) it succeeds and forms a singleton MPI program, that is, one in which MPI\_COMM\_WORLD has size 1.
- In some implementations, MPI may not be able to function without an "MPI environment." For example, MPI may require that daemons be running or MPI may not be able to work at all on the front-end of an MPP. In this case, an MPI implementation may either
  - 1. Create the environment (e.g., start a daemon) or
  - 2. Raise an error if it cannot create the environment and the environment has not been started independently.
  - A high-quality implementation will try to create a singleton MPI process and not raise an error.
    - (End of advice to implementors.)

MPI\_APPNUM

<sup>38</sup> 10.5.3

30 31

32

33 34

35 36

37

39

There is a predefined attribute MPI\_APPNUM of MPI\_COMM\_WORLD. In Fortran, the attribute is an integer value. In C, the attribute is a pointer to an integer value. If a process was spawned with MPI\_COMM\_SPAWN\_MULTIPLE, MPI\_APPNUM is the command number that generated the current process. Numbering starts from zero. If a process was spawned with MPI\_COMM\_SPAWN, it will have MPI\_APPNUM equal to zero.

Additionally, if the process was not started by a spawn call, but by an implementationspecific startup mechanism that can handle multiple process specifications, MPI\_APPNUM should be set to the number of the corresponding process specification. In particular, if it is started with

mpiexec spec0 [: spec1 : spec2 : ...]

MPI\_APPNUM should be set to the number of the corresponding specification.

If an application was not spawned with MPI\_COMM\_SPAWN or

MPI\_COMM\_SPAWN\_MULTIPLE, and MPI\_APPNUM does not make sense in the context of the implementation-specific startup mechanism, MPI\_APPNUM is not set.

MPI implementations may optionally provide a mechanism to override the value of MPI\_APPNUM through the info argument. MPI reserves the following key for all SPAWN calls.

appnum Value contains an integer that overrides the default value for MPI\_APPNUM in the child.

*Rationale.* When a single application is started, it is able to figure out how many processes there are by looking at the size of MPI\_COMM\_WORLD. An application consisting of multiple SPMD sub-applications has no way to find out how many sub-applications there are and to which sub-application the process belongs. While there are ways to figure it out in special cases, there is no general mechanism. MPI\_APPNUM provides such a general mechanism. (*End of rationale.*)

#### 10.5.4 Releasing Connections

Before a client and server connect, they are independent MPI applications. An error in one does not affect the other. After establishing a connection with MPI\_COMM\_CONNECT and MPI\_COMM\_ACCEPT, an error in one may affect the other. It is desirable for a client and server to be able to disconnect, so that an error in one will not affect the other. Similarly, it might be desirable for a parent and child to disconnect, so that errors in the child do not affect the parent, or vice-versa.

- Two processes are **connected** if there is a communication path (direct or indirect) between them. More precisely:
  - 1. Two processes are connected if
    - (a) they both belong to the same communicator (inter- or intra-, including MPI\_COMM\_WORLD) or
    - (b) they have previously belonged to a communicator that was freed with MPI\_COMM\_FREE instead of MPI\_COMM\_DISCONNECT or
    - (c) they both belong to the group of the same window or filehandle.
  - 2. If A is connected to B and B to C, then A is connected to C.
- Two processes are **disconnected** (also **independent**) if they are not connected.
- By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.
- Processes which are connected, but do not share the same MPI\_COMM\_WORLD, may become disconnected (independent) if the communication path between them is broken by using MPI\_COMM\_DISCONNECT.

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

 $44 \\ 45$ 

1	The following additional rules apply to MPI routines in other chapters:					
2 3	• MPI_FINALIZE is collective over a set of connected processes.					
4 5 6 7 8	• MPI_ABORT does not abort independent processes. It may abort all processes in the caller's MPI_COMM_WORLD (ignoring its comm argument). Additionally, it may abort connected processes as well, though it makes a "best attempt" to abort only the processes in comm.					
9 10	• If a process terminates without calling MPI_FINALIZE, independent processes are not affected but the effect on connected processes is not defined.					
11 12 13 14 15 16 17 18 19 20	Advice to implementors. An MPI implementation that tolerates process failures (as defined in Chapter 15.2.1) remains in a defined state after a process has failed. In practice, it may be difficult to distinguish between a process failure and an erroneous program that terminates without calling MPI_FINALIZE: in order to satisfy the process failure semantic, an implementation may have to exhibit the behavior defined for process failures with such erroneous programs. A high quality implementation should exhibit a different behavior for erroneous programs and process failures. (End of advice to implementors.)					
21 22	MPI_COMM_DISCONNECT(comm)					
23 24	INOUT comm communicator (handle)					
25 26	<pre>int MPI_Comm_disconnect(MPI_Comm *comm)</pre>					
27 28 29 30	<pre>MPI_Comm_disconnect(comm, ierror)     TYPE(MPI_Comm), INTENT(INOUT) :: comm     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
31 32	MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR					
33 34 35	This function waits for all pending communication on <b>comm</b> to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.					
36 37 38 39	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.					
40 41 42 43	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.					
44 45 46 47 48	Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. ( <i>End of advice to users.</i> )					

*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.*)

#### 10.5.5 Another Way to Establish MPI Communication

MPI\_COMM\_JOIN(fd, intercomm)

IN	fd	socket file descriptor	
OUT	intercomm	new intercommunicator (handle)	
int MPI_C	omm_join(int fd, MPI_Comm	*intercomm)	
<pre>MPI_Comm_join(fd, intercomm, ierror)     INTEGER, INTENT(IN) :: fd</pre>			
	MPI_Comm), INTENT(OUT) ::	intercomm	
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	
	JOIN(FD, INTERCOMM, IERRO	R)	
INTEG	ER FD, INTERCOMM, IERROR		

MPI\_COMM\_JOIN is intended for MPI implementations that exist in an environment supporting the Berkeley Socket interface [45, 49]. Implementations that exist in an environment not supporting Berkeley Sockets should provide the entry point for MPI\_COMM\_JOIN and should return MPI\_COMM\_NULL.

This call creates an intercommunicator from the union of two MPI processes which are connected by a socket. MPI\_COMM\_JOIN should normally succeed if the local and remote processes have access to the same implementation-defined MPI communication universe.

Advice to users. An MPI implementation may require a specific communication medium for MPI communication, such as a shared memory segment or a special switch. In this case, it may not be possible for two processes to successfully join even if there is a socket connecting them and they are using the same MPI implementation. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to establish communication over a slow medium if its preferred one is not available. If implementations do not do this, they must document why they cannot do MPI communication over the medium used by the socket (especially if the socket is a TCP connection). (End of advice to implementors.)

fd is a file descriptor representing a socket of type SOCK\_STREAM (a two-way reliable byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must not be enabled for the socket. The socket must be in a connected state. The socket must be quiescent when MPI\_COMM\_JOIN is called (see below). It is the responsibility of the application to create the socket using standard socket API calls.

MPI\_COMM\_JOIN must be called by the process at each end of the socket. It does not return until both processes have called MPI\_COMM\_JOIN. The two processes are referred to as the local and remote processes. 

#### Unofficial Draft for Comment Only

 $^{24}$ 

1 MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing  $\mathbf{2}$ else. Upon return from MPI\_COMM\_JOIN, the file descriptor will be open and quiescent 3 (see below). 4 If MPI is unable to create an intercommunicator, but is able to leave the socket in its  $\mathbf{5}$ original state, with no pending communication, it succeeds and sets intercomm to 6 MPI\_COMM\_NULL.  $\overline{7}$ The socket must be quiescent before MPI\_COMM\_JOIN is called and after 8 MPI\_COMM\_JOIN returns. More specifically, on entry to MPI\_COMM\_JOIN, a read on the 9 socket will not read any data that was written to the socket before the remote process called 10 MPI\_COMM\_JOIN. On exit from MPI\_COMM\_JOIN, a read will not read any data that was 11written to the socket before the remote process returned from MPI\_COMM\_JOIN. It is the 12responsibility of the application to ensure the first condition, and the responsibility of the 13MPI implementation to ensure the second. In a multithreaded application, the application 14must ensure that one thread does not access the socket while another is calling 15MPI\_COMM\_JOIN, or call MPI\_COMM\_JOIN concurrently. 16Advice to implementors. MPI is free to use any available communication path(s) 17 for MPI messages in the new communicator; the socket is only used for the initial 18 19handshaking. (End of advice to implementors.) 20MPI\_COMM\_JOIN uses non-MPI communication to do its work. The interaction of 21non-MPI communication with pending MPI communication is not defined. Therefore, the 22result of calling MPI\_COMM\_JOIN on two connected processes (see Section 10.5.4 for the 23definition of connected) is undefined.  $^{24}$ The returned communicator may be used to establish MPI communication with addi-25tional processes, through the usual MPI communicator creation mechanisms. 2627282930 3132 33 34 35 36 37 38 39

## Chapter 11

# **One-Sided** Communications

### 11.1 Introduction

**Remote Memory Access (RMA)** extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. In such a case, each process can compute what data it needs to access or to update at other processes. However, the programmer may not be able to easily determine which data in a process may need to be accessed or to be updated by operations executed by a different process, and may not even know which processes may perform such updates. Thus, the transfer parameters are all available only on one side. Regular send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This distribution may require all processes to participate in a time-consuming global computation, or to poll for potential communication requests to receive and upon which to act periodically. The use of RMA communication mechanisms avoids the need for global computations or explicit polling. A generic example of this nature is the execution of an assignment of the form A = B(map), where map is a permutation vector, and A, B, and map are distributed in the same manner.

Message-passing communication achieves two effects: *communication* of data from sender to receiver and *synchronization* of sender with receiver. The RMA design separates these two functions. The following communication calls are provided:

- Remote write: MPI\_PUT, MPI\_RPUT
- Remote read: MPI\_GET, MPI\_RGET
- Remote update: MPI\_ACCUMULATE, MPI\_RACCUMULATE
- Remote read and update: MPI\_GET\_ACCUMULATE, MPI\_RGET\_ACCUMULATE, and MPI\_FETCH\_AND\_OP
- Remote atomic swap operations: MPI\_COMPARE\_AND\_SWAP

This chapter refers to an operations set that includes all remote update, remote read and update, and remote atomic swap operations as "accumulate" operations.

7 8 9

15 16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37 38

39 40

41

42

43 44

 $45 \\ 46$ 

47

1 MPI supports two fundamentally different *memory models*: separate and *unified*. The  $\mathbf{2}$ separate model makes no assumption about memory consistency and is highly portable. 3 This model is similar to that of weakly coherent memory systems: the user must impose 4 correct ordering of memory accesses through synchronization calls. The unified model can  $\mathbf{5}$ exploit cache-coherent hardware and hardware-accelerated, one-sided operations that are 6 commonly available in high-performance systems. The two different models are discussed 7in detail in Section 11.4. Both models support several synchronization calls to support 8 different synchronization styles.

<sup>9</sup> The design of the RMA functions allows implementors to take advantage of fast or <sup>10</sup> asynchronous communication mechanisms provided by various platforms, such as coherent <sup>11</sup> or noncoherent shared memory, DMA engines, hardware-supported put/get operations, and <sup>12</sup> communication coprocessors. The most frequently used RMA communication mechanisms <sup>13</sup> can be layered on top of message-passing. However, certain RMA functions might need <sup>14</sup> support for asynchronous communication agents in software (handlers, threads, etc.) in a <sup>15</sup> distributed memory environment.

<sup>16</sup> We shall denote by **origin** the process that performs the call, and by **target** the <sup>17</sup> process in which the memory is accessed. Thus, in a put operation, source=origin and <sup>18</sup> destination=target; in a get operation, source=target and destination=origin.

19 20

21

404

## 11.2 Initialization

 $^{22}_{23}$  MPI provides the following window initialization functions: MPI\_WIN\_CREATE,

MPI\_WIN\_ALLOCATE, MPI\_WIN\_ALLOCATE\_SHARED, and

<sup>24</sup> MPI\_WIN\_CREATE\_DYNAMIC, which are collective on an intracommunicator.

<sup>25</sup> MPI\_WIN\_CREATE allows each process to specify a "window" in its memory that is made <sup>26</sup> accessible to accesses by remote processes. The call returns an opaque object that represents <sup>27</sup> the group of processes that own and access the set of windows, and the attributes of each <sup>28</sup> window, as specified by the initialization call. MPI\_WIN\_ALLOCATE differs from

<sup>25</sup> MPI\_WIN\_CREATE in that the user does not pass allocated memory;

<sup>31</sup> MPI\_WIN\_ALLOCATE returns a pointer to memory allocated by the MPI implementation. <sup>31</sup> MPI\_WIN\_ALLOCATE\_SHARED differs from MPI\_WIN\_ALLOCATE in that the allocated <sup>32</sup> memory can be accessed from all processes in the window's group with direct load/store <sup>33</sup> instructions. Some restrictions may apply to the specified communicator.

<sup>34</sup> MPI\_WIN\_CREATE\_DYNAMIC creates a window that allows the user to dynamically control <sup>35</sup> which memory is exposed by the window.

- 38
- 39
- 40 41
- 42

- 44 45
- 16
- 46 47
- 48

#### Window Creation 11 2 1

11.2.1	Window Creation		1
			2
			3
MPI_W	/IN_CREATE(base, size, disp_unit	:, info, comm, win)	4
IN	base	initial address of window (choice)	5 6
IN	size	size of window in bytes (non-negative integer)	7
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	8 9
IN	info	info argument (handle)	10 11
IN	comm	intra-communicator (handle)	12
OUT	win	window object returned by the call (handle)	13
			14 15
int MF	int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,		
	MPI_Comm comm, MPI_V	-	16 17
MPT Wi	n create(base, size, disp u	unit, info, comm, win, ierror)	18
	<pre>TPE(*), DIMENSION(), ASYNC</pre>		19
	TEGER(KIND=MPI_ADDRESS_KIND		20
IN	TEGER, INTENT(IN) :: disp_	unit	21
ТҮ	PE(MPI_Info), INTENT(IN) ::	info	22
ТҮ	<pre>PE(MPI_Comm), INTENT(IN) ::</pre>	comm	23
TY	<pre>PE(MPI_Win), INTENT(OUT) ::</pre>	win	24
IN	TEGER, OPTIONAL, INTENT(OUT	C) :: ierror	25
мрт шт	N CREATE (BASE STZE DISP II	JNIT, INFO, COMM, WIN, IERROR)	26
	<pre>wype&gt; BASE(*)</pre>	MIT, INIO, COMI, WIN, ILIUROIC	27
	TEGER(KIND=MPI_ADDRESS_KIND	)) SIZE	28
	TEGER DISP_UNIT, INFO, COMM		29
			30
'T'	ns is a collective call executed	by all processes in the group of comm. It returns	31

This is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations. Each process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address base. In C, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 18.1.12). A process may elect to expose no memory by specifying size = 0.

The displacement unit argument is provided to facilitate address arithmetic in RMA operations: the target displacement argument of an RMA operation is scaled by the factor disp\_unit specified by the target process, at window creation.

Rationale. The window size is specified using an address-sized integer, rather than a basic integer type, to allow windows that span more memory than can be described with a basic integer type. (End of rationale.)

Advice to users. Common choices for disp\_unit are 1 (no scaling), and (in C syntax) 47sizeof (type), for a window that consists of an array of elements of type type. The 48

#### **Unofficial Draft for Comment Only**

44

4546

32 33

34

35

36 37

38 39

	406		CHAPTER 11.	ONE-SIDED COMMUNICATIONS
1 2 3 4		v to byte displacements		in RMA calls, and have those scaled geneous environment. ( <i>End of advice</i>
5 6 7		argument provides opti window. The followin		the runtime about the expected usage edefined:
8 9 10 11	nization window.	(i.e., MPI_WIN_LOCK This implies that this	κ, MPI_WIN_LOC	y assume that passive target synchro- K_ALL) will not be used on the given used for 3-party communication, and ronous agent activity at this process.
12 13 14		dering — controls the $11.7.2$ for details.	ordering of accum	nulate operations at the target. See
15 16 17 18 19 20	accumul same_op calls to eliminat	ate calls to the same _no_op, then the imple the same target addres	target address wi mentation will as s will use the same access for certain	tion will assume that all concurrent ill use the same operation. If set to sume that all concurrent accumulate e operation or MPI_NO_OP. This can operation types where the hardware _no_op.
21 22 23 24		l on all processes, and	*	have provided this info key with the
25 26 27 28	disp_uni			tion may assume that the argument all processes have provided this info
29 30 31 32 33	to query It is rec	the specified info argu	uments for window	scribed in Section 11.2.7 can be used vs that have been passed to a library. d info keys for each passed window.
34 35 36 37 38 39 40	windows, in le put and accu should pose n associated wit	ocation, size, displacen mulate accesses to a p o problem. The same a	nent units, and in particular process area in memory m bject. However, co	y specify completely different target fo arguments. As long as all the get, fit their specific target window this hay appear in multiple windows, each oncurrent communications to distinct,
41 42 43 44 45 46 47 48	can be a ification process impleme	in an RMA operation a target of RMA opera . For example, with the to use RMA operation entation does enforce the at cannot affect any me	is to permit the p tions and for the in his definition, a se as, knowing that ( he specified limits)	by that may be accessed from another programmer to specify what memory implementation to enforce that spec- rver process can safely allow a client (under the assumption that the MPI on the exposed memory) an error in what was explicitly exposed. ( <i>End of</i>

## Unofficial Draft for Comment Only

Advice to users. A window can be created in any part of the process memory. However, on some systems, the performance of windows in memory allocated by MPI\_ALLOC\_MEM (Section 8.2) will be better. Also, on some systems, performance is improved when window boundaries are aligned at "natural" boundaries (word, double-word, cache line, page frame, etc.). (*End of advice to users.*)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the MPI\_WIN\_CREATE call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by MPI\_ALLOC\_MEM, or by other, implementation-specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI\_WIN\_CREATE occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allocate or to specify memory regions that are preferable for use in one-sided communication. In particular, such mechanisms can be used to place static variables into such preferred regions.

Implementors should document any performance impact of window alignment. (*End of advice to implementors.*)

### 11.2.2 Window That Allocates Memory

MPI_WIN_ALLOCATE(size, c	disp_unit, i	info, comm,	baseptr, win)
--------------------------	--------------	-------------	---------------

IN	size	size of window in bytes (non-negative integer)	28
IN	disp_unit	local unit size for displacements, in bytes (positive in-	29
	disp_unit	teger)	30
		J ,	31
IN	info	info argument (handle)	32
IN	comm	intra-communicator (handle)	33
	bacaptr	initial address of minder (sheise)	34
OUT	baseptr	initial address of window (choice)	35
OUT	win	window object returned by the call (handle)	36
			37
int MPI_N	/in_allocate(MPI_Aint siz	e, int disp_unit, MPI_Info info,	38
	MPI_Comm comm, void	*baseptr, MPI_Win *win)	39
		•	40

MPI\_Win\_allocate(size, disp\_unit, info, comm, baseptr, win, ierror)
USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR
INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: size
INTEGER, INTENT(IN) :: disp\_unit
TYPE(MPI\_Info), INTENT(IN) :: info
TYPE(MPI\_Comm), INTENT(IN) :: comm
TYPE(C\_PTR), INTENT(OUT) :: baseptr
TYPE(MPI\_Win), INTENT(OUT) :: win

#### 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

41

42

43

44

45

46

47

1	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2	MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
3	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
4	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
5	
6 7	This is a collective call executed by all processes in the group of comm. On each
8	process, it allocates memory of at least size bytes, returns a pointer to it, and returns a
9	window object that can be used by all processes in <b>comm</b> to perform RMA operations. The returned memory consists of <b>size</b> bytes local to each process, starting at address <b>baseptr</b>
10	and is associated with the window as if the user called MPI_WIN_CREATE on existing
11	memory. The size argument may be different at each process and size $= 0$ is valid; however, a
12	library might allocate and expose more memory in order to create a fast, globally symmetric
13	allocation. The discussion of and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in
14	Section 8.2 also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section 8.2
15	for an explanation of the type used for <b>baseptr</b> .
16	If the Fortran compiler provides $\texttt{TYPE(C_PTR)}$ , then the following generic interface must
17	be provided in the mpi module and should be provided in mpif.h through overloading,
18	i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
19 20	BASEPTR, but with a different specific procedure name:
20	INTERFACE MPI_WIN_ALLOCATE
22	SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
23	WIN, IERROR)
24	IMPORT :: MPI_ADDRESS_KIND
25	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
26	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
27	END SUBROUTINE
28	SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
29	WIN, IERROR)
30	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
31	IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
32 33	INTEGER (KIND=MPI_ADDRESS_KIND) :: SIZE
34	TYPE(C_PTR) :: BASEPTR
35	END SUBROUTINE
36	END INTERFACE
37	
38	The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR.
39	The implied specific procedure names are described in Section 18.1.5.
40	Rationale. By allocating (potentially aligned) memory instead of allowing the user
41	to pass in an arbitrary buffer, this call can improve the performance for systems with
42	remote direct memory access. This also permits the collective allocation of memory
43	and supports what is sometimes called the "symmetric allocation" model that can be
44	more scalable (for example, the implementation can arrange to return an address for
45 46	the allocated memory that is the same on all processes). (End of rationale.)
40 47	The info argument can be used to masify kints similar to the info argument for
48	The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM.

## Unofficial Draft for Comment Only

#### 11.2.3 Window That Allocates Shared Memory

			3
MPI_WIN_	ALLOCATE_SHARED(size, dis	sp_unit, info, comm, baseptr, win)	4
IN	size	size of local window in bytes (non-negative integer)	5 6
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	7 8
IN	info	info argument (handle)	9
IN	comm	intra-communicator (handle)	10 11
OUT	baseptr	address of local allocated window segment (choice)	12
OUT	win	window object returned by the call (handle)	13
			14
int MPI_W	in_allocate_shared(MPI_Ai	int size, int disp_unit, MPI_Info info,	15 16
	MPI_Comm comm, void	*baseptr, MPI_Win *win)	16
MPI Win a	llocate shared(size. dist	o_unit, info, comm, baseptr, win, ierror)	18
	INTRINSIC :: ISO_C_BINDI	-	19
INTEG	ER(KIND=MPI_ADDRESS_KIND)	), INTENT(IN) :: size	20
INTEG	ER, INTENT(IN) :: disp_u	mit	21
TYPE(	MPI_Info), INTENT(IN) ::	info	22
TYPE(	MPI_Comm), INTENT(IN) ::	comm	23
TYPE(	C_PTR), INTENT(OUT) :: h	baseptr	24
TYPE(	<pre>MPI_Win), INTENT(OUT) ::</pre>	win	25
INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror	26
ΜΡΤ ΜΤΝ Λ	LLOCATE SHARED (STZE DIS	P_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)	27
	ER DISP_UNIT, INFO, COMM,		28
	ER (KIND=MPI_ADDRESS_KIND)	-	29
TINTEG	ERCITIND-HI T_RODRESS_KIND)		30

 $^{31}$ This is a collective call executed by all processes in the group of comm. On each 32 process, it allocates memory of at least size bytes that is shared among all processes in 33 comm, and returns a pointer to the locally allocated segment in **baseptr** that can be used 34for load/store accesses on the calling process. The locally allocated memory can be the target of load/store accesses by remote processes; the base pointers for other processes 35 can be queried using the function MPI\_WIN\_SHARED\_QUERY. The call also returns a 36 37 window object that can be used by all processes in comm to perform RMA operations. The size argument may be different at each process and size = 0 is valid. It is the user's 38 39 responsibility to ensure that the communicator comm represents a group of processes that can create a shared memory segment that can be accessed by all processes in the group. 40 41 The discussions of rationales for MPI\_ALLOC\_MEM and MPI\_FREE\_MEM in Section 8.2 42also apply to MPI\_WIN\_ALLOCATE\_SHARED; in particular, see the rationale in Section 8.2 for an explanation of the type used for **baseptr**. The allocated memory is contiguous across 4344process ranks unless the info key alloc\_shared\_noncontig is specified. Contiguous across process ranks means that the first address in the memory segment of process i is consecutive with 4546the last address in the memory segment of process i-1. This may enable the user to 47calculate remote address offsets with local information only.

1

 $\mathbf{2}$ 

If the Fortran compiler provides TYPE(C\_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI\_ADDRESS\_KIND) BASEPTR, but with a different specific procedure name:

```
6
     INTERFACE MPI_WIN_ALLOCATE_SHARED
          SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
\overline{7}
                                                  BASEPTR, WIN, IERROR)
8
              IMPORT :: MPI_ADDRESS_KIND
9
              INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
10
11
              INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
          END SUBROUTINE
12
          SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
13
                                                        BASEPTR, WIN, IERROR)
14
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
15
              IMPORT :: MPI_ADDRESS_KIND
16
              INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
17
              INTEGER(KIND=MPI_ADDRESS_KIND) ::
18
                                                      SIZE
19
              TYPE(C_PTR) :: BASEPTR
          END SUBROUTINE
20
     END INTERFACE
21
22
          The base procedure name of this overloaded function is
23
     MPI_WIN_ALLOCATE_SHARED_CPTR. The implied specific procedure names are described
^{24}
     in Section 18.1.5.
25
          The info argument can be used to specify hints similar to the info argument for
26
     MPI_WIN_CREATE, MPI_WIN_ALLOCATE, and MPI_ALLOC_MEM. The additional info
27
     key alloc_shared_noncontig allows the library to optimize the layout of the shared memory
28
     segments in memory.
29
30
           Advice to users. If the info key alloc_shared_noncontig is not set to true, the allocation
31
           strategy is to allocate contiguous memory across process ranks. This may limit the
32
           performance on some architectures because it does not allow the implementation to
33
           modify the data layout (e.g., padding to reduce access latency). (End of advice to
34
           users.)
35
36
           Advice to implementors. If the user sets the info key alloc_shared_noncontig to true,
37
           the implementation can allocate the memory requested by each process in a location
38
           that is close to this process. This can be achieved by padding or allocating memory
39
           in special memory segments. Both techniques may make the address space across
40
           consecutive ranks noncontiguous. (End of advice to implementors.)
41
42
          The consistency of load/store accesses from/to the shared memory as observed by the
43
     user program depends on the architecture. A consistent view can be created in the unified
44
     memory model (see Section 11.4) by utilizing the window synchronization functions (see
45
     Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling
46
     MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows
47
     in the separate memory model.
48
```

1

 $\mathbf{2}$ 

3

4

IN	_SHARED_QUERY(win, rank win	shared memory window object (handle)	1 2	
		* * , ,	3	
IN	rank	rank in the group of window win (non-negative integer) or MPI_PROC_NULL	4 5	
OUT	size	size of the window segment (non-negative integer)	6	
OUT	disp_unit	local unit size for displacements, in bytes (positive in- teger)	7 8 9	
OUT	baseptr	address for load/store access to window segment (choice)	9 10 11	
int MPI	Win shared querv(MPI Wir	win, int rank, MPI_Aint *size,	12 13	
_	int *disp_unit, vo:		14	
MPI_Win_	<pre>shared_query(win, rank,</pre>	size, disp_unit, baseptr, ierror)	15 16	
	INTRINSIC :: ISO_C_BIN		17	
	(MPI_Win), INTENT(IN) ::		18	
INTEGER, INTENT(IN) :: rank				
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size INTEGER, INTENT(OUT) :: disp_unit				
TYPE(C_PTR), INTENT(OUT) :: baseptr			21	
	GER, OPTIONAL, INTENT(OU	-	22	
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)			23 24	
	GER WIN, RANK, DISP_UNIT		25	
	GER (KIND=MPI_ADDRESS_KI	-	26	
			27	
		-local address for remote memory segments created This function can return different process-local ad-	28	
		on different processes. The returned memory can be	29	
		the constraints defined in Section 11.7. This func-	30	
	,	of flavor MPI_WIN_FLAVOR_SHARED. If the passed	31	
	5	AVOR_SHARED, the error MPI_ERR_RMA_FLAVOR is	32 33	
raised. V	When rank is MPI_PROC_NUL	L, the pointer, disp_unit, and size returned are the	34	
		ory segment belonging the lowest rank that specified	35	
		attached to the window specified size $= 0$ , then the	36	
	-	f MPI_ALLOC_MEM was called with size $= 0$ .	37	
		YPE(C_PTR), then the following generic interface must	38	
-	-	should be provided in mpif.h through overloading, he routine with INTEGER(KIND=MPI_ADDRESS_KIND)	39	
,	but with a different specific		40	
211011 116,	sat with a unitern specific		41	
TNTEREAC	F MPT WIN SHARED OUFRY		42	

INTERFACE MPI_WIN_SHARED_QUERY	43
SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &	
BASEPTR, IERROR)	44
IMPORT :: MPI_ADDRESS_KIND	45
	46
INTEGER WIN, RANK, DISP_UNIT, IERROR	47
INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	40
	48

Unofficial Draft for Comment Only

1 END SUBROUTINE 2 SUBROUTINE MPI\_WIN\_SHARED\_QUERY\_CPTR(WIN, RANK, SIZE, DISP\_UNIT, & 3 BASEPTR, IERROR) 4 USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR 5IMPORT :: MPI\_ADDRESS\_KIND 6 INTEGER :: WIN, RANK, DISP\_UNIT, IERROR 7 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: SIZE 8 TYPE(C\_PTR) :: BASEPTR 9 END SUBROUTINE 10 END INTERFACE 11 The base procedure name of this overloaded function is 12MPI\_WIN\_SHARED\_QUERY\_CPTR. The implied specific procedure names are described in 13 Section 18.1.5. 14151611.2.4 Window of Dynamically Attached Memory 17The MPI-2 RMA model requires the user to identify the local memory that may be a 18 target of RMA calls at the time the window is created. This has advantages for both 19 the programmer (only this memory can be updated by one-sided operations and provides 20greater safety) and the MPI implementation (special steps may be taken to make one-21sided access to such memory more efficient). However, consider implementing a modifiable 22 linked list using RMA operations; as new items are added to the list, memory must be 23allocated. In a C or C++ program, this memory is typically allocated using malloc or 24new respectively. In MPI-2 RMA, the programmer must create a window with a predefined 25amount of memory and then implement routines for allocating memory from within the 26window's memory. In addition, there is no easy way to handle the situation where the 27predefined amount of memory turns out to be inadequate. To support this model, the 28routine MPI\_WIN\_CREATE\_DYNAMIC creates a window that makes it possible to expose 29 memory without remote synchronization. It must be used in combination with the local 30 routines MPI\_WIN\_ATTACH and MPI\_WIN\_DETACH.  $^{31}$ 32 33 MPI\_WIN\_CREATE\_DYNAMIC(info, comm, win) 34IN info info argument (handle) 35 36 IN comm intra-communicator (handle) 37 OUT win window object returned by the call (handle) 38 39 int MPI\_Win\_create\_dynamic(MPI\_Info info, MPI\_Comm comm, MPI\_Win \*win) 40 41MPI\_Win\_create\_dynamic(info, comm, win, ierror) 42TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_Comm), INTENT(IN) :: 43 comm 44TYPE(MPI\_Win), INTENT(OUT) :: win 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46MPI\_WIN\_CREATE\_DYNAMIC(INFO, COMM, WIN, IERROR) 47INTEGER INFO, COMM, WIN, IERROR 48

This is a collective call executed by all processes in the group of comm. It returns a window win without memory attached. Existing process memory can be attached as described below. This routine returns a window object that can be used by these processes to perform RMA operations on attached memory. Because this window has special properties, it will sometimes be referred to as a *dynamic* window.

The info argument can be used to specify hints similar to the info argument for MPI\_WIN\_CREATE.

In the case of a window created with MPI\_WIN\_CREATE\_DYNAMIC, the target\_disp for all RMA functions is the address at the target; i.e., the effective window\_base is MPI\_BOTTOM and the disp\_unit is one. For dynamic windows, the target\_disp argument to RMA communication operations is not restricted to non-negative values. Users should use MPI\_GET\_ADDRESS at the target process to determine the address of a target memory location and communicate this address to the origin process.

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type MPI\_Aint and result in unexpected values on some platforms. The MPI\_AINT\_ADD and MPI\_AINT\_DIFF functions can be used to safely perform address arithmetic with MPI\_Aint displacements. (*End of advice to users.*)

Advice to implementors. In environments with heterogeneous data representations, care must be exercised in communicating addresses between processes. For example, it is possible that an address valid at the target process (for example, a 64-bit pointer) cannot be expressed as an address at the origin (for example, the origin uses 32-bit pointers). For this reason, a portable MPI implementation should ensure that the type MPI\_AINT (see Table 3.3) is able to store addresses from any process. (*End of advice to implementors.*)

Memory at the target cannot be accessed with this window until that memory has been attached using the function MPI\_WIN\_ATTACH. That is, in addition to using MPI\_WIN\_CREATE\_DYNAMIC to create an MPI window, the user must use MPI\_WIN\_ATTACH before any local memory may be the target of an MPI RMA operation. Only memory that is currently accessible may be attached.

MPI\_WIN\_ATTACH(win, base, size)

1011 1		(, 5120)	35
IN	win	window object (handle)	36
IN	base	initial address of memory to be attached	37
IN	size	size of memory to be attached in bytes	38
	5120	size of memory to be attached in sytes	39
• .			40
int	int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)		
MPI_	MPI_Win_attach(win, base, size, ierror)		
	TYPE(MPI_Win), INTEN	T(IN) :: win	43
	TYPE(*), DIMENSION(), ASYNCHRONOUS :: base		
	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size		
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
MPI_	WIN_ATTACH(WIN, BASE	, SIZE, IERROR)	48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5

6

7

8 9

10

11

12

13

14

15

16

17

18 19

20

21

22

23

 $^{24}$ 

25

26 27

28

29

30

 $^{31}$ 

1	INTEGER WIN, IER	ROR			
2	<type> BASE(*)</type>				
3	INTEGER (KIND=MPI_ADDRESS_KIND) SIZE				
4 5	Attaches a local memory region beginning at <b>base</b> for remote access within the given				
6	window. The memory region specified must not contain any part that is already attached				
7	to the window win, that is, attaching overlapping memory concurrently within the same				
8		he argument win must be a window that was created with			
9		NAMIC. The local memory region attached to the window consists			
10	of size bytes, starting at address base. In C, base is the starting address of a memory region.				
11	In Fortran, one can pass the first element of a memory region or a whole array, which				
12	must be 'simply contiguous' (for 'simply contiguous,' see Section 18.1.12). Multiple (but non-overlapping) memory regions may be attached to the same window.				
13	non-overtapping) memor	y regions may be attached to the same window.			
14	Rationale. Requ	iring that memory be explicitly attached before it is exposed to			
15 16	one-sided access by	y other processes can simplify implementations and improve perfor-			
17		to make memory available for RMA operations without requiring a			
18		N_CREATE call is needed for some one-sided programming models.			
19	(End of rationale.)				
20	Advice to users.	Attaching memory to a window may require the use of scarce			
21		staching large regions of memory is not recommended in portable			
22	, , , ,	ing memory to a window may fail if sufficient resources are not			
23	available; this is si	milar to the behavior of MPI_ALLOC_MEM.			
24 25	The user is also re	esponsible for ensuring that MPI_WIN_ATTACH at the target has			
26	returned before a process attempts to target that memory with an MPI RMA call.				
27 28 29	Performing an RMA operation to memory that has not been attached to a window created with MPI_WIN_CREATE_DYNAMIC is erroneous. ( <i>End of advice to users.</i> )				
30	Advice to impleme	entors. A high-quality implementation will attempt to make as			
31	much memory ava	ilable for attaching as possible. Any limitations should be docu-			
32	mented by the implementor. (End of advice to implementors.)				
33	A., 1 · .				
34		is a local operation as defined by MPI, which means that the call pletes without requiring any MPI routine to be called in any other			
35 36		e detached with the routine MPI_WIN_DETACH. After memory has			
37		ot be the target of an MPI RMA operation on that window (unless			
38	, 0	ed with MPI_WIN_ATTACH).			
39					
40	MPI_WIN_DETACH(win	hase)			
41	Ϋ́,	·			
42	IN win	window object (handle)			
43	IN base	initial address of memory to be detached			
44					
45 46	int MPI_Win_detach(M	PI_Win win, const void *base)			
47	MPI_Win_detach(win, base, ierror)				
48	TYPE(MPI_Win), I				

TYPE(*), DIMENSION(), ASYNCHRONOUS :: base	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2
	3
MPI_WIN_DETACH(WIN, BASE, IERROR)	4
INTEGER WIN, IERROR	5
<type> BASE(*)</type>	6
Detaches a previously attached memory region beginning at <b>base</b> . The arguments <b>base</b>	7

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 11.2.5.

and win must match the arguments passed to a previous call to MPI\_WIN\_ATTACH.

```
MPI_WIN_FREE(win)
INOUT win window object (handle)
int MPI_Win_free(MPI_Win *win)
MPI_Win_free(win, ierror)
TYPE(MPI_Win), INTENT(INOUT) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_WIN_FREE(WIN, IERROR)
```

```
INTEGER WIN, IERROR
```

Frees the window object win and returns a null handle (equal to MPI\_WIN\_NULL). This is a collective call executed by all processes in the group associated with win. MPI\_WIN\_FREE(win) can be invoked by a process only after it has completed its involvement in RMA communications on window win: e.g., the process has called MPI\_WIN\_FENCE, or called MPI\_WIN\_WAIT to match a previous call to MPI\_WIN\_POST or called MPI\_WIN\_COMPLETE to match a previous call to MPI\_WIN\_START or called MPI\_WIN\_UNLOCK to match a previous call to MPI\_WIN\_LOCK. The memory associated with windows created by a call to MPI\_WIN\_CREATE may be freed after the call returns. If the window was created with MPI\_WIN\_ALLOCATE, MPI\_WIN\_FREE will free the window memory that was allocated in MPI\_WIN\_ALLOCATE. If the window was created with MPI\_WIN\_ALLOCATE\_SHARED, MPI\_WIN\_FREE will free the window memory that was allocated in MPI\_WIN\_ALLOCATE\_NA

Freeing a window that was created with a call to MPI\_WIN\_CREATE\_DYNAMIC detaches all associated memory; i.e., it has the same effect as if all attached memory was detached by calls to MPI\_WIN\_DETACH.

```
47
48
```

8

9

10

11

12

13 14

15

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	Advice to implem	nentors. MPI_WIN_FREE I	equires a barr	ier synchronization: no		
2	-	rn from free until all proces	0	-		
3	win call free. This ensures that no process will attempt to access a remote window					
4	(e.g., with lock/unlock) after it was freed. The only exception to this rule is when the					
5	user sets the $no\_locks$ info key to true when creating the window. In that case, an MPI					
6	-	may free the local window wit	thout barrier s	ynchronization. (End of		
7	advice to implem	entors.)				
8						
9 10	11.2.6 Window Attri	butes				
11	The following attribute	es are cached with a window v	when the wind	ow is created.		
12	MPI_WIN_BASE	window	base address.			
13	MPI_WIN_SIZE		size, in bytes.			
14	MPI_WIN_DISP_UNIT		, .	ciated with the window.		
15	MPI_WIN_CREATE_FI	AVOR how the	window was c	reated.		
16	MPI_WIN_MODEL	memory	model for win	dow.		
17	In C calls to MDI	Win got attr(win MDI W/N	DACE Libro	e.flog)		
18		_Win_get_attr(win, MPI_WIN MPI_WIN_SIZE, &size, &flag		ællag),		
19						
20	<u> </u>	MPI_WIN_DISP_UNIT, &dis MPI_WIN_CREATE_FLAVOF		l l flog) and		
21	<u> </u>	MPI_WIN_MODEL, &memo				
22	<u> </u>		-			
23	pointer to the start of the window win, and will return in size, disp_unit, create_kind, and					
	momony model pointer			memory_model pointers to the size, displacement unit of the window, the kind of routine		
24		s to the size, displacement u	nit of the wind			
25	used to create the win	s to the size, displacement un dow, and the memory model,	nit of the wind respectively.	A detailed listing of the		
25 26	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model, the attribute value argument	nit of the wind respectively.	A detailed listing of the		
25 26 27	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model,	nit of the wind respectively.	A detailed listing of the		
25 26 27 28	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model, the attribute value argument	nit of the wind respectively.	A detailed listing of the		
25 26 27 28 29	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute	nit of the wind respectively. to MPI_WIN_	A detailed listing of the		
25 26 27 28 29 30	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE	nit of the wind respectively. to MPI_WIN_ C Type void *	A detailed listing of the		
25 26 27 28 29 30 31	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE	<pre>nit of the wind respectively. to MPI_WIN_ C Type void * MPI_Aint *</pre>	A detailed listing of the		
25 26 27 28 29 30 31 32	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT	<pre>nit of the wind respectively. to MPI_WIN_ C Type void * MPI_Aint * int *</pre>	A detailed listing of the		
25 26 27 28 29 30 31 32 33	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR	to MPI_WIN_ C Type void * MPI_Aint * int *	A detailed listing of the		
25 26 27 28 29 30 31 32 33 34	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT	<pre>nit of the wind respectively. to MPI_WIN_ C Type void * MPI_Aint * int *</pre>	A detailed listing of the		
25 26 27 28 29 30 31 32 33 34 35	used to create the win type of the pointer in	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR	to MPI_WIN_ C Type void * MPI_Aint * int *	A detailed listing of the		
25 26 27 28 29 30 31 32 33 34 35 36	used to create the win type of the pointer in MPI_WIN_SET_ATTR	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR	<pre>nit of the wind respectively. to MPI_WIN_ Void * MPI_Aint * int * int * int *</pre>	A detailed listing of the _GET_ATTR and		
25 26 27 28 29 30 31 32 33 34 35 36 37	used to create the win type of the pointer in MPI_WIN_SET_ATTR	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL	<pre>nit of the wind respectively. to MPI_WIN_ Void * MPI_Aint * int * int * int *</pre>	A detailed listing of the _GET_ATTR and		
25 26 27 28 29 30 31 32 33 34 35 36 37 38	used to create the win type of the pointer in MPI_WIN_SET_ATTR Table 11.1: C types o MPI_WIN_SET_ATTR.	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to	nit of the wind         respectively.         to MPI_WIN_         C Type         void *         MPI_Aint *         int *         int *         o MPI_WIN_GE	A detailed listing of the GET_ATTR and		
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39	used to create the win type of the pointer in MPI_WIN_SET_ATTR Table 11.1: C types o MPI_WIN_SET_ATTR. In Fortran, calls to	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to p MPI_WIN_GET_ATTR(win,	<pre>nit of the wind respectively. to MPI_WIN_ C Type void * MPI_Aint * int * int * int * MPI_WIN_GE MPI_WIN_BAS</pre>	A detailed listing of the GET_ATTR and		
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40	used to create the win type of the pointer in MPI_WIN_SET_ATTR Table 11.1: C types o MPI_WIN_SET_ATTR. In Fortran, calls to MPI_WIN_GET_ATTR	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to p MPI_WIN_GET_ATTR(win, (win, MPI_WIN_SIZE, size, fla	<pre>nit of the wind respectively. to MPI_WIN_ void * MPI_Aint * int * int * o MPI_WIN_GE MPI_WIN_BAS g, ierror),</pre>	A detailed listing of the _GET_ATTR and		
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	used to create the win type of the pointer in MPI_WIN_SET_ATTR Table 11.1: C types o MPI_WIN_SET_ATTR. In Fortran, calls to MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to p MPI_WIN_GET_ATTR(win, (win, MPI_WIN_SIZE, size, fla (win, MPI_WIN_DISP_UNIT,	nit of the wind         respectively.         to MPI_WIN_         C Type         void *         MPI_Aint *         int *         int *         o MPI_WIN_GE         MPI_WIN_BAS         g, ierror),         disp_unit, flag,	A detailed listing of the GET_ATTR and		
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	Table 11.1: C types o MPI_WIN_SET_ATTR In Fortran, calls to MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to D MPI_WIN_GET_ATTR(win, (win, MPI_WIN_SIZE, size, fla (win, MPI_WIN_DISP_UNIT, (win, MPI_WIN_CREATE_FLA	<pre>nit of the wind respectively. to MPI_WIN_ void * MPI_Aint * int * int * o MPI_WIN_GE MPI_WIN_BAS g, ierror), disp_unit, flag, WOR, create_k</pre>	A detailed listing of the GET_ATTR and GET_ATTR and ET_ATTR and SE, base, flag, ierror), ierror), ind, flag, ierror), and		
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	used to create the win type of the pointer in MPI_WIN_SET_ATTR Table 11.1: C types o MPI_WIN_SET_ATTR. In Fortran, calls to MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to p MPI_WIN_GET_ATTR(win, (win, MPI_WIN_SIZE, size, fla (win, MPI_WIN_DISP_UNIT, (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_CREATE_FLA	nit of the wind         respectively.         to MPI_WIN_         Void *         MPI_Aint *         int *         int *         o MPI_WIN_GB         MPI_WIN_BAS         g, ierror),         disp_unit, flag,         WOR, create_k         mory_model, flag	A detailed listing of the _GET_ATTR and		
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	Table 11.1: C types o MPI_WIN_SET_ATTR MPI_WIN_SET_ATTR In Fortran, calls to MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR base, size, disp_unit, c	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to MPI_WIN_GET_ATTR(win, (win, MPI_WIN_SIZE, size, fla (win, MPI_WIN_DISP_UNIT, (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_MODEL, menore reate_kind, and memory_model	nit of the wind         respectively.         to MPI_WIN_         Void *         MPI_Aint *         int *         int *         o MPI_WIN_GE         MPI_WIN_BAS         g, ierror),         disp_unit, flag,         WOR, create_k         mory_model, flage         el the (integen	A detailed listing of the GET_ATTR and ET_ATTR and ET_ATTR and SE, base, flag, ierror), ierror), ind, flag, ierror), and ag, ierror) will return in r representation of) the		
25 26 27 28 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	Table 11.1: C types o MPI_WIN_SET_ATTR MPI_WIN_SET_ATTR In Fortran, calls to MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR base, size, disp_unit, c	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to p MPI_WIN_GET_ATTR(win, (win, MPI_WIN_SIZE, size, fla (win, MPI_WIN_DISP_UNIT, (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_CREATE_FLA	nit of the wind         respectively.         to MPI_WIN_         Void *         MPI_Aint *         int *         int *         o MPI_WIN_GE         MPI_WIN_BAS         g, ierror),         disp_unit, flag,         WOR, create_k         mory_model, flage         el the (integen	A detailed listing of the GET_ATTR and ET_ATTR and ET_ATTR and SE, base, flag, ierror), ierror), ind, flag, ierror), and ag, ierror) will return in r representation of) the		
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	Table 11.1: C types o MPI_WIN_SET_ATTR In Fortran, calls to MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR base, size, disp_unit, c base address, the size,	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to MPI_WIN_GET_ATTR(win, (win, MPI_WIN_SIZE, size, fla (win, MPI_WIN_DISP_UNIT, (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_MODEL, menore reate_kind, and memory_model	nit of the wind         respectively.         to MPI_WIN_ <b>C Type</b> void *         MPI_Aint *         int *         int *         o MPI_WIN_GE         MPI_WIN_GE         MPI_WIN_BAS         g, ierror),         disp_unit, flag,         VOR, create_k         mory_model, flag         el the (integen         vindow win, the	A detailed listing of the GET_ATTR and ET_ATTR and ET_ATTR and SE, base, flag, ierror), ierror), ind, flag, ierror), and ag, ierror) will return in r representation of) the		
25 26 27 28 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	Table 11.1: C types o MPI_WIN_SET_ATTR In Fortran, calls to MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR MPI_WIN_GET_ATTR base, size, disp_unit, c base address, the size,	s to the size, displacement un dow, and the memory model, the attribute value argument is shown in Table 11.1. Attribute MPI_WIN_BASE MPI_WIN_DISP_UNIT MPI_WIN_CREATE_FLAVOR MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL f attribute value argument to p MPI_WIN_GET_ATTR(win, (win, MPI_WIN_SIZE, size, fla (win, MPI_WIN_DISP_UNIT, (win, MPI_WIN_DISP_UNIT, (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_CREATE_FLA (win, MPI_WIN_MODEL, memory_mode the displacement unit of the v I the memory model, respecti	nit of the wind         respectively.         to MPI_WIN_ <b>C Type</b> void *         MPI_Aint *         int *         int *         o MPI_WIN_GE         MPI_WIN_GE         MPI_WIN_BAS         g, ierror),         disp_unit, flag,         VOR, create_k         mory_model, flag         el the (integen         vindow win, the	A detailed listing of the GET_ATTR and ET_ATTR and ET_ATTR and SE, base, flag, ierror), ierror), ind, flag, ierror), and ag, ierror) will return in r representation of) the		

MPI_WIN_FLAVOR_CREATE	Window was created with MPI_WIN_CREATE.	1
MPI_WIN_FLAVOR_ALLOCATE	Window was created with	2
	MPI_WIN_ALLOCATE.	3
MPI_WIN_FLAVOR_DYNAMIC	Window was created with	4
	MPI_WIN_CREATE_DYNAMIC.	5
MPI_WIN_FLAVOR_SHARED	Window was created with	6
	MPI_WIN_ALLOCATE_SHARED.	7
		8

The values of memory\_model are MPI\_WIN\_SEPARATE and MPI\_WIN\_UNIFIED. The meaning of these is described in Section 11.4.

In the case of windows created with MPI\_WIN\_CREATE\_DYNAMIC, the base address is MPI\_BOTTOM and the size is 0. In C, pointers are returned, and in Fortran, the values are returned, for the respective attributes. (The window attribute access functions are defined in Section 6.7.3.) The value returned for an attribute on a window is constant over the lifetime of the window.

The other "window attribute," namely the group of processes attached to the window, can be retrieved using the call below.

MPI\_WIN\_GET\_GROUP(win, group) 19 20IN win window object (handle) 21OUT group group of processes which share access to the window 22 (handle) 23 $^{24}$ int MPI\_Win\_get\_group(MPI\_Win win, MPI\_Group \*group) 2526MPI\_Win\_get\_group(win, group, ierror) 27TYPE(MPI\_Win), INTENT(IN) :: win 28 TYPE(MPI\_Group), INTENT(OUT) :: group 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI\_WIN\_GET\_GROUP(WIN, GROUP, IERROR) 31INTEGER WIN, GROUP, IERROR 32

MPI\_WIN\_GET\_GROUP returns a duplicate of the group of the communicator used to create the window associated with win. The group is returned in group.

#### 11.2.7 Window Info

Hints specified via info (see Section 9) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or use system resources more efficiently. However, hints do not change the semantics of any MPI interfaces. In other words, an implementation is free to ignore all hints. Hints are specified on a per window basis, in window creation functions and MPI\_WIN\_SET\_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI\_WIN\_SET\_INFO there will be no effect on previously set or default hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In

#### Unofficial Draft for Comment Only

9

10

11

12

13

14

15

16

17 18

33

34

35 36

37 38

39

40

41

42

43

44

45

46 47

```
1
           general, unsupported hints should simply be ignored. Needless to say, no hint can be
\mathbf{2}
           mandatory. However, for each hint used by a specific implementation, a default value
3
           must be provided when the user does not specify a value for the hint. (End of advice
4
           to implementors.)
5
6
7
     MPI_WIN_SET_INFO(win, info)
8
9
       INOUT
                                              window object (handle)
                 win
10
       IN
                 info
                                              info object (handle)
11
12
     int MPI_Win_set_info(MPI_Win win, MPI_Info info)
13
14
     MPI_Win_set_info(win, info, ierror)
15
          TYPE(MPI_Win), INTENT(IN) :: win
16
          TYPE(MPI_Info), INTENT(IN) ::
                                              info
17
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
18
     MPI_WIN_SET_INFO(WIN, INFO, IERROR)
19
          INTEGER WIN, INFO, IERROR
20
21
          MPI_WIN_SET_INFO sets new values for the hints of the window associated with win.
22
     The call is collective on the group of win. The info object may be different on each process,
23
     but any info entries that an implementation requires to be the same on all processes must
^{24}
     appear with the same value in each process's info object.
25
26
                             Some info items that an implementation can use when it creates
           Advice to users.
27
           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
28
           creation call. (End of advice to users.)
29
30
^{31}
32
     MPI_WIN_GET_INFO(win, info_used)
33
                                              window object (handle)
34
       IN
                 win
35
       OUT
                 info_used
                                              new info object (handle)
36
37
     int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
38
39
     MPI_Win_get_info(win, info_used, ierror)
40
          TYPE(MPI_Win), INTENT(IN) :: win
41
          TYPE(MPI_Info), INTENT(OUT) :: info_used
42
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
43
     MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
44
          INTEGER WIN, INFO_USED, IERROR
45
46
          MPI_WIN_GET_INFO returns a new info object containing the hints of the window
47
     associated with win. The current setting of all hints actually used by the system related to
```

this window is returned in info\_used. If no such hints exist, a handle to a newly created

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

 $\frac{44}{45}$ 

46

47

48

info object is returned that contains no key/value pair. The user is responsible for freeing info\_used via MPI\_INFO\_FREE.

Advice to users. The info object returned in info\_used will contain all hints currently active for this window. This set of hints may be greater or smaller than the set of hints specified when the window was created, as the system may not recognize some hints set by the user, and may recognize other hints that the user has not set. (End of advice to users.)

## 11.3 Communication Calls

MPI supports the following RMA communication calls: MPI\_PUT and MPI\_RPUT transfer data from the caller memory (origin) to the target memory; MPI\_GET and MPI\_RGET transfer data from the target memory to the caller memory; MPI\_ACCUMULATE and MPI\_RACCUMULATE update locations in the target memory, e.g., by adding to these locations values sent from the caller memory; MPI\_GET\_ACCUMULATE,

MPI\_RGET\_ACCUMULATE, and MPI\_FETCH\_AND\_OP perform atomic read-modify-write and return the data before the accumulate operation; and MPI\_COMPARE\_AND\_SWAP performs a remote atomic compare and swap operation. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, at the origin or both the origin and the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 11.5. Transfers can also be completed with calls to flush routines; see Section 11.5.4 for details. For the MPI\_RPUT, MPI\_RGET, MPI\_RACCUMULATE, and MPI\_RGET\_ACCUMULATE calls, the transfer can be locally completed by using the MPI test or wait operations described in Section 3.7.3.

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call until the operation completes at the origin.

The resulting data values, or outcome, of concurrent conflicting accesses to the same memory locations is undefined; if a location is updated by a put or accumulate operation, then the outcome of loads or other RMA operations is undefined until the updating operation has completed at the target. There is one exception to this rule; namely, the same location can be updated by several concurrent accumulate calls, the outcome being as if these updates occurred in some order. In addition, the outcome of concurrent load/store and RMA updates to the same memory location is undefined. These restrictions are described in more detail in Section 11.7.

The calls use general datatype arguments to specify communication buffers at the origin and at the target. Thus, a transfer operation may also gather data at the source and scatter it at the destination. However, all arguments specifying both communication buffers are provided by the caller.

For all RMA calls, the target process may be identical with the origin process; i.e., a process may use an RMA operation to move data in its memory.

*Rationale.* The choice of supporting "self-communication" is the same as for messagepassing. It simplifies some coding, and is very useful with accumulate operations, to allow atomic updates of local variables. (*End of rationale.*)

#### Unofficial Draft for Comment Only

MPI\_PROC\_NULL is a valid target rank in all MPI RMA communication calls. The effect  $\mathbf{2}$ 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 4 synchronization method that started the epoch.

### 11.3.1 Put

The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target process. The obvious difference is that all arguments are provided by one call — the call executed by the origin process.

12MPI\_PUT(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, 13 target\_datatype, win) 14

15	IN	origin_addr	initial address of origin buffer (choice)
16	IN	origin_count	number of entries in origin buffer (non-negative inte-
17			ger)
18 19	IN	origin_datatype	datatype of each entry in origin buffer (handle)
20	IN	target_rank	rank of target (non-negative integer)
21	IN	target_disp	displacement from start of window to target buffer
22			(non-negative integer)
23	IN	target_count	number of entries in target buffer (non-negative inte-
24			ger)
25			
26	IN	target_datatype	datatype of each entry in target buffer (handle)
27	IN	win	window object used for communication (handle)

```
29
     int MPI_Put(const void *origin_addr, int origin_count,
30
                   MPI_Datatype origin_datatype, int target_rank,
                   MPI_Aint target_disp, int target_count,
32
```

```
MPI_Datatype target_datatype, MPI_Win win)
```

MPI\_Put(origin\_addr, origin\_count, origin\_datatype, target\_rank, 34 target\_disp, target\_count, target\_datatype, win, ierror) 35TYPE(\*), 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\_Win), INTENT(IN) :: win 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42

MPI\_PUT(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, 43 TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, IERROR) 44<type> ORIGIN\_ADDR(\*) 45INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 46INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 47 TARGET\_DATATYPE, WIN, IERROR 48

1

3

56

7

8

9

10 11

28

31

Transfers origin\_count successive entries of the type specified by the origin\_datatype, starting at address origin\_addr on the origin node, to the target node specified by the win, target\_rank pair. The data are written in the target buffer at address target\_addr =window\_base + target\_disp  $\times$  disp\_unit, where window\_base and disp\_unit are the base address and window displacement unit specified at window initialization, by the target process.

The target buffer is specified by the arguments target\_count and target\_datatype.

The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin\_addr, origin\_count, origin\_datatype, target\_rank, tag, comm, and the target process executed a receive operation with arguments target\_addr, target\_count, target\_datatype, source, tag, comm, where target\_addr is the target buffer address computed as explained above, the values of tag are arbitrary valid matching tag values, and comm is a communicator for the group of win.

The communication must satisfy the same constraints as for a similar message-passing communication. The target\_datatype may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window or in attached memory in a dynamic window.

The target\_datatype argument is a handle to a datatype object defined at the origin process. However, this object is interpreted at the target process: the outcome is as if 19the target datatype object was defined at the target process by the same sequence of calls 20used to define it at the origin process. The target datatype must contain only relative 21displacements, not absolute addresses. The same holds for get and accumulate operations.

Advice to users. The target\_datatype argument is a handle to a datatype object that is defined at the origin process, even though it defines a data layout in the target process memory. This causes no problems in a homogeneous environment, or in a heterogeneous environment if only portable datatypes are used (portable datatypes are defined in Section 2.4).

The performance of a put transfer can be significantly affected, on some systems, by the choice of window location and the shape and location of the origin and target buffer: transfers to a target window in memory allocated by MPI\_ALLOC\_MEM or MPI\_WIN\_ALLOCATE may be much faster on shared memory systems; transfers from contiguous buffers will be faster on most, if not all, systems; the alignment of the communication buffers may also impact performance. (End of advice to users.)

Advice to implementors. A high-quality implementation will attempt to prevent remote accesses to memory outside the window that was exposed by the process. This is important both for debugging purposes and for protection with client-server codes that use RMA. That is, a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an MPI exception at the origin call if an out-of-bound situation occurs. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (End of advice to implementors.)

- 4445
- 46
- 47

1

 $\mathbf{2}$ 

3

4

5

6  $\overline{7}$ 

8 9

10 11

12

13

14

15

16

17

18

22

23

 $^{24}$ 

25

26

2728

29

30

 $^{31}$ 

32

33

34 35

36

37

38

39

40

41

42

```
422
                                         CHAPTER 11. ONE-SIDED COMMUNICATIONS
1
     11.3.2 Get
\mathbf{2}
3
4
     MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count,
5
                    target_datatype, win)
6
       OUT
                 origin_addr
                                             initial address of origin buffer (choice)
7
8
                 origin_count
       IN
                                             number of entries in origin buffer (non-negative inte-
9
                                             ger)
10
       IN
                 origin_datatype
                                             datatype of each entry in origin buffer (handle)
11
                 target_rank
       IN
                                             rank of target (non-negative integer)
12
       IN
                 target_disp
                                             displacement from window start to the beginning of
13
                                             the target buffer (non-negative integer)
14
15
       IN
                 target_count
                                             number of entries in target buffer (non-negative inte-
16
                                             ger)
17
       IN
                 target_datatype
                                             datatype of each entry in target buffer (handle)
18
19
       IN
                 win
                                             window object used for communication (handle)
20
21
     int MPI_Get(void *origin_addr, int origin_count,
22
                    MPI_Datatype origin_datatype, int target_rank,
23
                    MPI_Aint target_disp, int target_count,
24
                    MPI_Datatype target_datatype, MPI_Win win)
25
     MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
26
                    target_disp, target_count, target_datatype, win, ierror)
27
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
28
          INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
29
          TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
30
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
31
          TYPE(MPI_Win), INTENT(IN) :: win
32
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
33
34
     MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
35
                     TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
36
          <type> ORIGIN_ADDR(*)
37
          INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
38
          INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
39
          TARGET_DATATYPE, WIN, IERROR
40
          Similar to MPI_PUT, except that the direction of data transfer is reversed. Data
41
     are copied from the target memory to the origin. The origin_datatype may not specify
42
     overlapping entries in the origin buffer. The target buffer must be contained within the
43
     target window or within attached memory in a dynamic window, and the copied data must
44
     fit, without truncation, in the origin buffer.
45
46
47
48
```

## 11.3.3 Examples for Communication Calls

These examples show the use of the MPI\_GET function. As all MPI RMA communication functions are nonblocking, they must be completed. In the following, this is accomplished with the routine MPI\_WIN\_FENCE, introduced in Section 11.5.

**Example 11.1** We show how to implement the generic indirect assignment A = B(map), where A, B, and map have the same distribution, and map is a permutation. To simplify, we assume a block distribution with equal size blocks.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER otype(p), oindex(m),
                               & ! used to construct origin datatypes
     ttype(p), tindex(m),
                             & ! used to construct target datatypes
     count(p), total(p),
                               &
     disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
! This part does the work that depends on the locations of B.
! Can be reused while this does not change
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                         &
                     comm, win, ierr)
! This part does the work that depends on the value of map and
! the locations of the arrays.
! Can be reused while these do not change
! Compute number of entries to be received from each process
DO i=1,p
  count(i) = 0
END DO
DO i=1,m
  j = map(i)/m+1
  count(j) = count(j)+1
END DO
total(1) = 0
DO i=2,p
  total(i) = total(i-1) + count(i-1)
END DO
```

1 2

3

4

5 6

7

8

9 10

11

12 13

 $14 \\ 15$ 

16

17

18

19

20 21

22

23 24

25

26

27

28

29 30

31

32

33 34

35 36

37

38

39

40

41

42

43 44

45

46

47

```
1
     DO i=1,p
\mathbf{2}
       count(i) = 0
3
     END DO
4
\mathbf{5}
     ! compute origin and target indices of entries.
6
     ! entry i at current process is received from location
7
     ! k at process (j-1), where map(i) = (j-1)*m + (k-1),
8
     ! j = 1...p and k = 1...m
9
10
     DO i=1,m
^{11}
       j = map(i)/m+1
12
       k = MOD(map(i), m) + 1
13
       count(j) = count(j)+1
14
       oindex(total(j) + count(j)) = i
15
       tindex(total(j) + count(j)) = k
16
     END DO
17
18
     ! create origin and target datatypes for each get operation
19
     DO i=1,p
20
       CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
21
                                             oindex(total(i)+1:total(i)+count(i)), &
22
                                             MPI_REAL, otype(i), ierr)
23
       CALL MPI_TYPE_COMMIT(otype(i), ierr)
^{24}
       CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
25
                                             tindex(total(i)+1:total(i)+count(i)), &
26
                                             MPI_REAL, ttype(i), ierr)
27
       CALL MPI_TYPE_COMMIT(ttype(i), ierr)
28
     END DO
29
30
     ! this part does the assignment itself
^{31}
     CALL MPI_WIN_FENCE(0, win, ierr)
32
     disp_aint = 0
33
     DO i=1,p
34
       CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
35
     END DO
36
     CALL MPI_WIN_FENCE(0, win, ierr)
37
38
     CALL MPI_WIN_FREE(win, ierr)
39
     DO i=1,p
40
       CALL MPI_TYPE_FREE(otype(i), ierr)
41
       CALL MPI_TYPE_FREE(ttype(i), ierr)
42
     END DO
43
     RETURN
44
     END
45
46
     Example 11.2
47
48
```

A simpler version can be written that does not require that a datatype be built for the target buffer. But, one then needs a separate get call for each entry, as illustrated below. This code is much simpler, but usually much less efficient, for large arrays.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                        &
                    comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  disp_aint = MOD(map(i),m)
  CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

#### 11.3.4 Accumulate Functions

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather than replacing it. This will allow, for example, the accumulation of a sum by having all involved processes add their contributions to the sum variable in the memory of one process. The accumulate functions have slightly different semantics with respect to overlapping data accesses than the put and get functions; see Section 11.7 for details.

18

19

20

21

22

30

 $^{31}$ 

32

33

34

35

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10 11

1	Accumulat	e Function	
2 3			
4 5	MPI_ACCU	JMULATE(origin_addr, origin_ target_count, target_dat	.count, origin_datatype, target_rank, target_disp, atype, op, win)
6 7	IN	origin_addr	initial address of buffer (choice)
8	IN	origin_count	number of entries in buffer (non-negative integer)
9	IN	origin_datatype	datatype of each entry (handle)
10 11	IN	target_rank	rank of target (non-negative integer)
11 12 13	IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)
14 15	IN	target_count	number of entries in target buffer (non-negative inte- ger)
16 17	IN	target_datatype	datatype of each entry in target buffer (handle)
18	IN	ор	reduce operation (handle)
19	IN	win	window object (handle)
22 23 24 25 26 27 28 29 30 31 32 33 34 35	MPI_Accum TYPE( INTEG TYPE( INTEG TYPE( INTEG	<pre>MPI_Datatype origin_ MPI_Aint target_disp MPI_Datatype target_ mulate(origin_addr, origi target_disp, target_ (*), DIMENSION(), INTEN ER, INTENT(IN) :: origi MPI_Datatype), INTENT(IN ER(KIND=MPI_ADDRESS_KIND (MPI_Op), INTENT(IN) :: (MPI_Op), INTENT(IN) :: ER, OPTIONAL, INTENT(OUT)</pre>	<pre>_datatype, MPI_Op op, MPI_Win win) n_count, origin_datatype, target_rank, _count, target_datatype, op, win, ierror) T(IN), ASYNCHRONOUS :: origin_addr n_count, target_rank, target_count ) :: origin_datatype, target_datatype ), INTENT(IN) :: target_disp op win ) :: ierror</pre>
36 37 38 39 40 41 42 43 44 45 46 47 48	<type INTEG INTEG TARGE Accur origin_data offset targe op. This overwriting Any o</type 	TARGET_DISP, TARGET_ >> ORIGIN_ADDR(*) ER(KIND=MPI_ADDRESS_KIND ER ORIGIN_COUNT, ORIGIN_ T_DATATYPE, OP, WIN, IER nulate the contents of the origin type) to the buffer specified et_disp, in the target window s is like MPI_PUT except that g it. f the predefined operations for	DATATYPE, TARGET_RANK, TARGET_COUNT, ROR in buffer (as defined by origin_addr, origin_count, and by arguments target_count and target_datatype, at specified by target_rank and win, using the operation is data is combined into the target area instead of r MPI_REDUCE can be used. User-defined functions
48	cannot be	used. For example, if op is M	PI_SUM, each element of the origin buffer is added

to the corresponding element in the target, replacing the former value in the target.

Each datatype argument must be a predefined datatype or a derived datatype, where all basic components are of the same predefined datatype. Both datatype arguments must be constructed from the same predefined datatype. The operation **op** applies to elements of that predefined type. The parameter **target\_datatype** must not specify overlapping entries, and the target buffer must fit in the target window.

A new predefined operation, MPI\_REPLACE, is defined. It corresponds to the associative function f(a, b) = b; i.e., the current value in the target memory is replaced by the value supplied by the origin.

MPI\_REPLACE can be used only in MPI\_ACCUMULATE, MPI\_RACCUMULATE, MPI\_GET\_ACCUMULATE, MPI\_FETCH\_AND\_OP, and MPI\_RGET\_ACCUMULATE, but not in collective reduction operations such as MPI\_REDUCE.

Advice to users. MPI\_PUT is a special case of MPI\_ACCUMULATE, with the operation MPI\_REPLACE. Note, however, that MPI\_PUT and MPI\_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

**Example 11.3** We want to compute  $B(j) = \sum_{map(i)=j} A(i)$ . The arrays A, B, and map are distributed in the same manner. We write the simple version.

```
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, win, ierr, disp_int
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
size = m * realextent
disp_int = realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL, &
                    comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  disp_aint = MOD(map(i),m)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL,
                                                                       &
                      MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

This code is identical to the code in Example 11.2, except that a call to get has been <sup>46</sup> replaced by a call to accumulate. (Note that, if map is one-to-one, the code computes <sup>47</sup>  $B = A(map^{-1})$ , which is the reverse assignment to the one computed in that previous <sup>48</sup>

#### Unofficial Draft for Comment Only

1

2

3

4

5

6 7

8

9

10 11

12 13

14

15

16 17 18

19

20 21

22

23

 $^{24}$ 

25

26 27

28

29

30

31

32 33

34

35

36

37

38

39

40

41 42

43

44

example.) In a similar manner, we can replace in Example 11.1, the call to get by a call to
 accumulate, thus performing the computation with only one communication between any
 two processes.

4 5

6

Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 11.7 for details). The predefined operation MPI\_REPLACE provides fetch-and-set behavior.

```
    <sup>13</sup> MPI_GET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr,
result_count, result_datatype, target_rank, target_disp, target_count,
target_datatype, op, win)
```

17	IN	origin_addr	initial address of buffer (choice)
18 19	IN	origin_count	number of entries in origin buffer (non-negative integer)
20 21	IN	origin_datatype	datatype of each entry in origin buffer (handle)
22	OUT	result_addr	initial address of result buffer (choice)
23 24	IN	result_count	number of entries in result buffer (non-negative integer)
25 26	IN	result_datatype	datatype of each entry in result buffer (handle)
27	IN	target_rank	rank of target (non-negative integer)
28 29	IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)
30 31 32	IN	target_count	number of entries in target buffer (non-negative integer)
33	IN	target_datatype	datatype of each entry in target buffer (handle)
34	IN	ор	reduce operation (handle)
35 36	IN	win	window object (handle)
37 38 39 40 41 42	int MPI_C	MPI_Datatype origin_ int result_count, MP int target_rank, MPI	<pre>*origin_addr, int origin_count, datatype, void *result_addr, PI_Datatype result_datatype, Aint target_disp, int target_count, datatype, MPI_Op op, MPI_Win win)</pre>
43 44 45 46	TYPE	<pre>result_count, result target_count, target (*), DIMENSION(), INTEN</pre>	rigin_count, origin_datatype, result_addr, _datatype, target_rank, target_disp, _datatype, op, win, ierror) T(IN), ASYNCHRONOUS :: origin_addr
47 48	TYPE(	(*), DIMENSION(), ASYNC	HRUNUUS :: result_addr

INTEGER, INTENT(IN) :: origin\_count, result\_count, target\_rank, target\_count TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype, result\_datatype INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp TYPE(MPI\_Op), INTENT(IN) :: op TYPE(MPI\_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_GET\_ACCUMULATE(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, RESULT\_ADDR, RESULT\_COUNT, RESULT\_DATATYPE, TARGET\_RANK, TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, IERROR) <type> ORIGIN\_ADDR(\*), RESULT\_ADDR(\*) INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, RESULT\_COUNT, RESULT\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, TARGET\_DATATYPE, OP, WIN, IERROR

Accumulate origin\_count elements of type origin\_datatype from the origin buffer ( origin\_addr) to the buffer at offset target\_disp, in the target window specified by target\_rank and win, using the operation op and return in the result buffer result\_addr the content of the target buffer before the accumulation, 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 <sup>24</sup> datatype argument must be a predefined datatype or a derived datatype where all basic <sup>25</sup> 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 <sup>27</sup> predefined type. target\_datatype must not specify overlapping entries, and the target buffer <sup>28</sup> must fit in the target window or in attached memory in a dynamic window. The operation <sup>29</sup> is executed atomically for each basic datatype; see Section 11.7 for details. <sup>30</sup>

Any of the predefined operations for MPI\_REDUCE, as well as MPI\_NO\_OP or MPI\_REPLACE can be specified as op. User-defined functions cannot be used. A new predefined operation, MPI\_NO\_OP, is defined. It corresponds to the associative function f(a, b) = a; i.e., the current value in the target memory is returned in the result buffer at the origin and no operation is performed on the target buffer. When MPI\_NO\_OP is specified as the operation, the origin\_addr, origin\_count, and origin\_datatype arguments are ignored. MPI\_NO\_OP can be used only in MPI\_GET\_ACCUMULATE, MPI\_RGET\_ACCUMULATE, and MPI\_FETCH\_AND\_OP. MPI\_NO\_OP cannot be used in MPI\_ACCUMULATE, MPI\_RACCUMULATE, or collective reduction operations, such as MPI\_REDUCE and others.

Advice to users. MPI\_GET is similar to MPI\_GET\_ACCUMULATE, with the operation MPI\_NO\_OP. Note, however, that MPI\_GET and MPI\_GET\_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

#### Fetch and Op Function

The generic functionality of MPI\_GET\_ACCUMULATE might limit the performance of fetchand-increment or fetch-and-add calls that might be supported by special hardware oper-

#### 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

31

32

33

34

35

36

37

38

39

40

41

42

43 44 45

46

47

	430	Cl	IAPTER 11. ON	NE-SIDED COMMUNICATIONS
1 2 3		PI_FETCH_AND_OP thus allo he functionality of MPI_GET_	*	elementation of a commonly used
4 5 6	MPI_FETC	:H_AND_OP(origin_addr, resu	t_addr, datatype,	<pre>target_rank, target_disp, op, win)</pre>
7	IN	origin_addr	initial address of	buffer (choice)
8	OUT	result_addr	initial address of	result buffer (choice)
9 10 11	IN	datatype	datatype of the er fers (handle)	ntry in origin, result, and target buf-
12	IN	target_rank	rank of target (no	on-negative integer)
13 14 15	IN	target_disp	displacement from get buffer (non-ne	n start of window to beginning of tar- egative integer)
16	IN	ор	reduce operation	(handle)
17	IN	win	window object (h	andle)
22 23 24 25 26 27 28 29 30 31 32 33	TYPE ( TYPE ( TYPE ( INTEG INTEG TYPE ( TYPE ( INTEG	<pre>MPI_Op op, MPI_Win w _and_op(origin_addr, resp target_disp, op, win *), DIMENSION(), INTEN *), DIMENSION(), ASYNCH MPI_Datatype), INTENT(IN ER, INTENT(IN) :: targe ER(KIND=MPI_ADDRESS_KIND) MPI_Op), INTENT(IN) :: 0 MPI_Op), INTENT(IN) :: 0 ER, OPTIONAL, INTENT(OUT)</pre>	alt_addr, datat , ierror) C(IN), ASYNCHRO IRONOUS :: res :: datatype c_rank , INTENT(IN) : pp win :: ierror	NOUS :: origin_addr ult_addr : target_disp
34 35 36 37 38 39 40 41	<type INTEG INTEG Accum buffer at c the operation</type 	offset target_disp, in the targetion op and return in the result	, IERROR) ADDR(*) TARGET_DISP OP, WIN, IERR Matatype from the t window specifie	
42 43 44 45 46 47 48	The or predefined specified a	operations for MPI_REDUCE	, as well as MPI_ annot be used. T	ddr) must be disjoint. Any of the NO_OP or MPI_REPLACE, can be 'he datatype argument must be a y.

Compare an	d Swap Function	1	
compared t	-	nich is atomically replaced by a third value only if $4$	3 4 5
MPI_COMF	PARE_AND_SWAP(origin_add target_disp, win)	r, compare_addr, result_addr, datatype, target_rank,	8
IN	origin_addr	initial address of buffer (choice)	
IN	compare_addr	initial address of compare buffer (choice) 1	1
OUT	result_addr	initial address of result buffer (choice)	
IN	datatype	datatype of the element in all buffers (handle)	
IN	target_rank	rank of target (non-negative integer)	
IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)	7
IN	win	window object (handle)	
TYPE(; TYPE(; TYPE(; TYPE(I INTEGI INTEGI TYPE(I	MPI_Aint target_disp re_and_swap(origin_addr, target_rank, target_	<pre>compare_addr, result_addr, datatype, disp, win, ierror) T(IN), ASYNCHRONOUS :: origin_addr T(IN), ASYNCHRONOUS :: compare_addr HRONOUS :: result_addr ) :: datatype t_rank ), INTENT(IN) :: target_disp win</pre>	3 4 5 6 7 8 9 0 1 2 3
<type INTEGI INTEGI This fu compare_ad target_rank origin_addr the target one of the f</type 	TARGET_RANK, TARGET_ > ORIGIN_ADDR(*), COMPARI ER(KIND=MPI_ADDRESS_KIND) ER DATATYPE, TARGET_RANK unction compares one element dr with the buffer at offset t and win and replaces the val if the compare buffer and th is returned in the buffer resu following categories of predefi	E_ADDR(*), RESULT_ADDR(*) 3 ) TARGET_DISP 3	6 7 8 9 0 1 2 3 4 5

 $({\sf origin\_addr} \ {\rm and} \ {\sf result\_addr}) \ {\rm must} \ {\rm be} \ {\rm disjoint}.$ 

47

1	11.3.5	Request-based RMA Comm	nunication Operations
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19	Request- with the functions a passive Upon MPI_ERR other fiel are under requests, in function MPI_REG RMA req The MPI_WIN MPI_WIN ever, user clean up	RMA operations and test of described in Section 3.7.3. target epoch (see Section 1 n returning from a complet OR field in the associated sta ds of status and the result fined. It is valid to mix di collective requests, I/O req ons that enable multiple cor QUEST_FREE or MPI_CANG uests are not persistent. end of the epoch, or expl J_FLUSH, MPI_WIN_FLUSH J_FLUSH_LOCAL_ALL, also rs must still wait or test on t any resources associated with	operations allow the user to associate a request handle or wait for the completion of these requests using the Request-based RMA operations are only valid within
20	complete	locally.	
21			
22 23 24	MPI_RPL	JT(origin_addr, origin_count, target_datatype, win,	origin_datatype, target_rank, target_disp, target_count, request)
24 25	IN	origin_addr	initial address of origin buffer (choice)
26 27	IN	origin_count	number of entries in origin buffer (non-negative integer)
28 29	IN	origin_datatype	datatype of each entry in origin buffer (handle)
29 30	IN	target_rank	rank of target (non-negative integer)
31 32	IN	target_disp	displacement from start of window to target buffer (non-negative integer)
33 34	IN	target_count	number of entries in target buffer (non-negative integer)
35 36	IN	target_datatype	datatype of each entry in target buffer (handle)
37	IN	win	window object used for communication (handle)
38 39	OUT	request	RMA request (handle)
40 41 42 43 44 45 46 47		MPI_Datatype orig MPI_Aint target_d MPI_Datatype targ MPI_Request *requ c(origin_addr, origin_co	n_addr, int origin_count, in_datatype, int target_rank, isp, int target_count, et_datatype, MPI_Win win, est) ount, origin_datatype, target_rank, et_count, target_datatype, win, request,
48		ierror)	_ , 0,,,,,,,,,,,,,,,,,,,,,,,,,

TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 1  $\mathbf{2}$ INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 3 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 4 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp TYPE(MPI\_Win), INTENT(IN) :: win 56 TYPE(MPI\_Request), INTENT(OUT) :: request 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 MPI\_RPUT(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, 9 TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, REQUEST, 10 IERROR) 11 <type> ORIGIN\_ADDR(\*) 12INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 13 INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 14TARGET\_DATATYPE, WIN, REQUEST, IERROR 1516MPI\_RPUT is similar to MPI\_PUT (Section 11.3.1), except that it allocates a commu-

nication request object and associates it with the request handle (the argument request). The completion of an MPI\_RPUT operation (i.e., after the corresponding test or wait) indicates that the sender is now free to update the locations in the origin buffer. It does not indicate that the data is available at the target window. If remote completion is required, MPI\_WIN\_FLUSH, MPI\_WIN\_FLUSH\_ALL, MPI\_WIN\_UNLOCK, or MPI\_WIN\_UNLOCK\_ALL can be used.

MPI\_RGET(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, win, request)

OUT	origin_addr	initial address of origin buffer (choice)	27
IN	origin_count	number of entries in origin buffer (non-negative inte-	28 29
		ger)	29 30
IN	origin_datatype	datatype of each entry in origin buffer (handle)	31
IN	target_rank	rank of target (non-negative integer)	32
IN	target_disp	displacement from window start to the beginning of	33
		the target buffer (non-negative integer)	$\frac{34}{35}$
IN	target_count	number of entries in target buffer (non-negative inte- ger)	36 37
IN	target_datatype	datatype of each entry in target buffer (handle)	38
IN	win	window object used for communication (handle)	$\frac{39}{40}$
OUT	request	RMA request (handle)	40 41
	,	• • • /	42

int MPI\_Rget(void \*origin\_addr, int origin\_count,

```
MPI_Datatype origin_datatype, int target_rank,44MPI_Aint target_disp, int target_count,45MPI_Datatype target_datatype, MPI_Win win,46MPI_Request *request)47
```

Unofficial Draft for Comment Only

17

18

19

20

21

22

23 24

25

26

$\frac{1}{2}$	MPI_Rget		_count, origin_datatype, target_rank,
3		ierror)	rget_count, target_datatype, win, request,
4	ТҮРГ		ASYNCHRONOUS :: origin_addr
5			prigin_count, target_rank, target_count
6			<pre>IT(IN) :: origin_datatype, target_datatype</pre>
7			_KIND), INTENT(IN) :: target_disp
8		E(MPI_Win), INTENT(IN)	
9		E(MPI_Request), INTENT	-
10 11	INTE	EGER, OPTIONAL, INTEN	C(OUT) :: ierror
12	MPI_RGET	CORIGIN_ADDR, ORIGIN_	COUNT, ORIGIN_DATATYPE, TARGET_RANK,
13			RGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
14		IERROR)	
15		De> ORIGIN_ADDR(*)	VIND) TADGET DIGD
16		EGER(KIND=MPI_ADDRESS EGER ORIGIN COUNT ORI	IGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
17 18		GET_DATATYPE, WIN, REC	
18			
20			GET (Section 11.3.2), except that it allocates a commu- iates it with the request handle (the argument request)
21		1 U	completion. The completion of an MPI_RGET operation
22			le in the origin buffer. If origin_addr points to memory
23	attached	to a window, then the da	ta becomes available in the private copy of this window.
24			
25			
26	MPI_RAC	CUMULATE(origin_addr,	origin_count, origin_datatype, target_rank, target_disp,
26 27	MPI_RAC		origin_count, origin_datatype, target_rank, target_disp, :t_datatype, op, win, request)
27 28	MPI_RAC		
27		target_count, targe	t_datatype, op, win, request)
27 28 29	IN	target_count, targe	t_datatype, op, win, request) initial address of buffer (choice)
27 28 29 30 31 32	IN IN	target_count, targe origin_addr origin_count	t_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer)
27 28 29 30 31 32 33 34	IN IN IN	target_count, target origin_addr origin_count origin_datatype	t_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer) datatype of each entry in origin buffer (handle)
27 28 29 30 31 32 33 34 35	IN IN IN IN	target_count, target origin_addr origin_count origin_datatype target_rank target_disp	<pre>ht_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer) datatype of each entry in origin buffer (handle) rank of target (non-negative integer) displacement from start of window to beginning of tar-</pre>
27 28 29 30 31 32 33 34	IN IN IN IN	target_count, target origin_addr origin_count origin_datatype target_rank	<pre>ht_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer) datatype of each entry in origin buffer (handle) rank of target (non-negative integer) displacement from start of window to beginning of tar- get buffer (non-negative integer)</pre>
27 28 29 30 31 32 33 34 35 36	IN IN IN IN	target_count, target origin_addr origin_count origin_datatype target_rank target_disp	<pre>ht_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer) datatype of each entry in origin buffer (handle) rank of target (non-negative integer) displacement from start of window to beginning of tar- get buffer (non-negative integer) number of entries in target buffer (non-negative inte- </pre>
27 28 29 30 31 32 33 34 35 36 37 38 39	IN IN IN IN IN	target_count, target origin_addr origin_count origin_datatype target_rank target_disp target_count	<pre>ht_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer) datatype of each entry in origin buffer (handle) rank of target (non-negative integer) displacement from start of window to beginning of tar- get buffer (non-negative integer) number of entries in target buffer (non-negative inte- ger)</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39 40	IN IN IN IN IN	target_count, target origin_addr origin_count origin_datatype target_rank target_disp target_count target_datatype	<ul> <li>t_datatype, op, win, request)</li> <li>initial address of buffer (choice)</li> <li>number of entries in buffer (non-negative integer)</li> <li>datatype of each entry in origin buffer (handle)</li> <li>rank of target (non-negative integer)</li> <li>displacement from start of window to beginning of target buffer (non-negative integer)</li> <li>number of entries in target buffer (non-negative integer)</li> <li>datatype of each entry in target buffer (handle)</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39	IN IN IN IN IN IN	<pre>target_count, target origin_addr origin_count origin_datatype target_rank target_disp target_count target_datatype op</pre>	<ul> <li>initial address of buffer (choice)</li> <li>number of entries in buffer (non-negative integer)</li> <li>datatype of each entry in origin buffer (handle)</li> <li>rank of target (non-negative integer)</li> <li>displacement from start of window to beginning of target buffer (non-negative integer)</li> <li>number of entries in target buffer (non-negative integer)</li> <li>datatype of each entry in target buffer (handle)</li> <li>reduce operation (handle)</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	IN IN IN IN IN IN IN	target_count, target origin_addr origin_count origin_datatype target_rank target_disp target_count target_datatype op win	<ul> <li>initial address of buffer (choice)</li> <li>number of entries in buffer (non-negative integer)</li> <li>datatype of each entry in origin buffer (handle)</li> <li>rank of target (non-negative integer)</li> <li>displacement from start of window to beginning of target buffer (non-negative integer)</li> <li>number of entries in target buffer (non-negative integer)</li> <li>datatype of each entry in target buffer (handle)</li> <li>reduce operation (handle)</li> <li>window object (handle)</li> </ul>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 41 42 43 44	IN IN IN IN IN IN IN OUT	<pre>target_count, target origin_addr origin_count origin_datatype target_rank target_disp target_count target_datatype op win request _Raccumulate(const voit</pre>	<pre>ht_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer) datatype of each entry in origin buffer (handle) rank of target (non-negative integer) displacement from start of window to beginning of tar- get buffer (non-negative integer) number of entries in target buffer (non-negative inte- ger) datatype of each entry in target buffer (handle) reduce operation (handle) window object (handle) RMA request (handle) id *origin_addr, int origin_count,</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	IN IN IN IN IN IN IN OUT	<pre>target_count, target origin_addr origin_count origin_datatype target_rank target_disp target_count target_datatype op win request _Raccumulate(const vo: MPI_Datatype or</pre>	<pre>ht_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer) datatype of each entry in origin buffer (handle) rank of target (non-negative integer) displacement from start of window to beginning of tar- get buffer (non-negative integer) number of entries in target buffer (non-negative inte- ger) datatype of each entry in target buffer (handle) reduce operation (handle) window object (handle) RMA request (handle) id *origin_addr, int origin_count, igin_datatype, int target_rank,</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	IN IN IN IN IN IN IN OUT	<pre>target_count, target origin_addr origin_count origin_datatype target_rank target_disp target_count target_datatype op win request _Raccumulate(const voi MPI_Datatype or MPI_Aint target</pre>	<pre>ht_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer) datatype of each entry in origin buffer (handle) rank of target (non-negative integer) displacement from start of window to beginning of tar- get buffer (non-negative integer) number of entries in target buffer (non-negative inte- ger) datatype of each entry in target buffer (handle) reduce operation (handle) window object (handle) RMA request (handle) id *origin_addr, int origin_count, igin_datatype, int target_rank, _disp, int target_count,</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	IN IN IN IN IN IN IN OUT	<pre>target_count, target origin_addr origin_count origin_datatype target_rank target_disp target_count target_datatype op win request _Raccumulate(const voi MPI_Datatype or MPI_Aint target</pre>	<pre>ht_datatype, op, win, request) initial address of buffer (choice) number of entries in buffer (non-negative integer) datatype of each entry in origin buffer (handle) rank of target (non-negative integer) displacement from start of window to beginning of tar- get buffer (non-negative integer) number of entries in target buffer (non-negative inte- ger) datatype of each entry in target buffer (handle) reduce operation (handle) window object (handle) RMA request (handle) id *origin_addr, int origin_count, igin_datatype, int target_rank, _disp, int target_count, rget_datatype, MPI_Op op, MPI_Win win,</pre>

Unofficial Draft for Comment Only

MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,	1
<pre>target_disp, target_count, target_datatype, op, win, request,</pre>	2
ierror)	3
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr	4
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count	5
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype	6
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	7
TYPE(MPI_Op), INTENT(IN) :: op	8
TYPE(MPI_Win), INTENT(IN) :: win	9
TYPE(MPI_Request), INTENT(OUT) :: request	10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	11
MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	12
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	13
IARGET_DISF, IARGET_COUNT, IARGET_DATATIFE, OF, WIN, REQUEST, IERROR)	14
<pre>type&gt; ORIGIN_ADDR(*)</pre>	15
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	16
	17
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	18
TARGET_DATATYPE, OP, WIN, REQUEST, IERROR	19
MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section 11.3.4), except that	20
it allocates a communication request object and associates it with the request handle (the	21

it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI\_RACCUMULATE operation indicates that the origin buffer is free to be updated. It does not indicate that the operation has completed at the target window.

1MPI\_RGET\_ACCUMULATE(origin\_addr, origin\_count, origin\_datatype, result\_addr,  $\mathbf{2}$ result\_count, result\_datatype, target\_rank, target\_disp, target\_count, 3 target\_datatype, op, win, request) 4 IN origin\_addr initial address of buffer (choice) 5IN origin\_count number of entries in origin buffer (non-negative inte-6 ger) 7 8 IN origin\_datatype datatype of each entry in origin buffer (handle) 9 OUT result\_addr initial address of result buffer (choice) 10 IN result\_count number of entries in result buffer (non-negative inte-11 ger) 1213IN result\_datatype datatype of each entry in result buffer (handle) 14IN target\_rank rank of target (non-negative integer) 15IN target\_disp displacement from start of window to beginning of tar-16 get buffer (non-negative integer) 1718 IN target\_count number of entries in target buffer (non-negative inte-19 ger) 20IN target\_datatype datatype of each entry in target buffer (handle) 21IN ор reduce operation (handle) 22 23IN win window object (handle)  $^{24}$ OUT RMA request (handle) request 2526int MPI\_Rget\_accumulate(const void \*origin\_addr, int origin\_count, 27MPI\_Datatype origin\_datatype, void \*result\_addr, 28int result\_count, MPI\_Datatype result\_datatype, 29 int target\_rank, MPI\_Aint target\_disp, int target\_count, 30 MPI\_Datatype target\_datatype, MPI\_Op op, MPI\_Win win,  $^{31}$ MPI\_Request \*request) 32 33MPI\_Rget\_accumulate(origin\_addr, origin\_count, origin\_datatype, 34 result\_addr, result\_count, result\_datatype, target\_rank, 35 target\_disp, target\_count, target\_datatype, op, win, request, 36 ierror) 37 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 38 TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: result\_addr 39 INTEGER, INTENT(IN) :: origin\_count, result\_count, target\_rank, 40 target\_count 41 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype, 42result\_datatype 43 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 44 TYPE(MPI\_Op), INTENT(IN) :: op 45TYPE(MPI\_Win), INTENT(IN) :: win 46 TYPE(MPI\_Request), INTENT(OUT) :: request 47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

MPI\_RGET\_ACCUMULATE is similar to MPI\_GET\_ACCUMULATE (Section 11.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI\_RGET\_ACCUMULATE operation indicates that the data is available in the result buffer and the origin buffer is free to be updated. It does not indicate that the operation has been completed at the target window.

# 11.4 Memory Model

The memory semantics of RMA are best understood by using the concept of *public* and 19private window copies. We assume that systems have a public memory region that is 20addressable by all processes (e.g., the shared memory in shared memory machines or the 21exposed main memory in distributed memory machines). In addition, most machines have 22 fast 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 35accesses 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 40 in process memory, and public copies of other overlapping windows. This is illustrated in 41 Figure 11.1. 42

In the RMA unified model, public and private copies are identical and updates via put or accumulate calls are eventually observed by load operations without additional RMA calls. A store access to a window is eventually visible to remote get or accumulate calls without additional RMA calls. These stronger semantics of the RMA unified model allow the user to omit some synchronization calls and potentially improve performance. 47

 $^{48}$ 

### Unofficial Draft for Comment Only

1 2

3

4

5

6 7

8 9

10

11

12

13

14

15 16 17

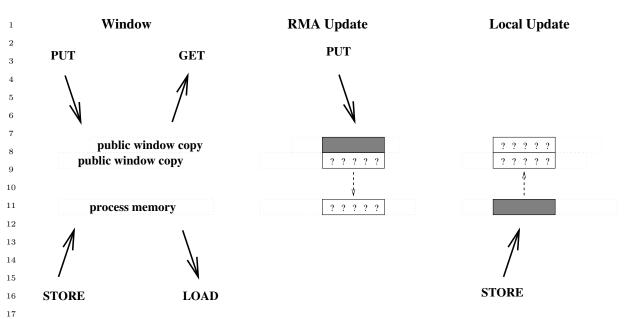


Figure 11.1: Schematic description of the public/private window operations in the MPI\_WIN\_SEPARATE memory model for two overlapping windows.

Advice to users. If accesses in the RMA unified model are not synchronized (with locks or flushes, see Section 11.5.3), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (*End of advice to users.*)

The memory model for a particular RMA window can be determined by accessing the attribute MPI\_WIN\_MODEL. If the memory model is the unified model, the value of this attribute is MPI\_WIN\_UNIFIED; otherwise, the value is MPI\_WIN\_SEPARATE.

# 11.5 Synchronization Calls

RMA communications fall in two categories:

• active target communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.

• **passive target communication**, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

# Unofficial Draft for Comment Only

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI\_PUT, MPI\_GET or MPI\_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

1. The MPI\_WIN\_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates with many others.

This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI\_WIN\_FENCE. A process can access windows at all processes in the group of win during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.

2. The four functions MPI\_WIN\_START, MPI\_WIN\_COMPLETE, MPI\_WIN\_POST, and MPI\_WIN\_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly RMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.

These calls are used for active target communication. An access epoch is started <sup>42</sup> at the origin process by a call to MPI\_WIN\_START and is terminated by a call to <sup>43</sup> MPI\_WIN\_COMPLETE. The start call has a group argument that specifies the group <sup>44</sup> of target processes for that epoch. An exposure epoch is started at the target process <sup>45</sup> by a call to MPI\_WIN\_POST and is completed by a call to MPI\_WIN\_WAIT. The post <sup>46</sup> call has a group argument that specifies the set of origin processes for that epoch. <sup>47</sup>

#### 48

# Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8 9

10

11

12

13

14

15

16

17

18

19 20

21

22

23

24

25

26

27

28

29 30

 $^{31}$ 

32 33

34

35

36

37

38

39

40

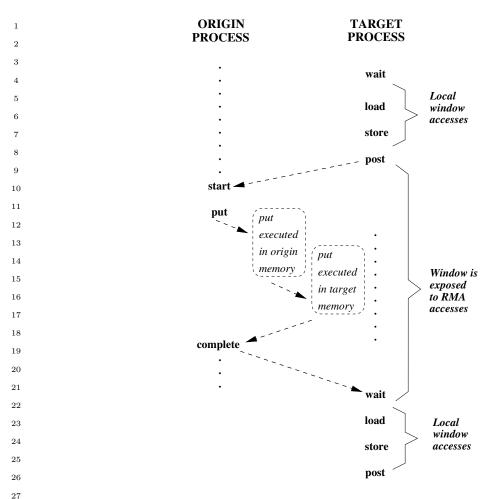


Figure 11.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

29 30 31

32

33 34

35

36 37

38

39

40

28

3. Finally, shared lock access is provided by the functions MPI\_WIN\_LOCK,

MPI\_WIN\_LOCK\_ALL, MPI\_WIN\_UNLOCK, and MPI\_WIN\_UNLOCK\_ALL.

MPI\_WIN\_LOCK and MPI\_WIN\_UNLOCK also provide exclusive lock capability. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a "billboard" model, where processes can, at random times, access or update different parts of the billboard.

These four calls provide passive target communication. An access epoch is started by a call to MPI\_WIN\_LOCK or MPI\_WIN\_LOCK\_ALL and terminated by a call to MPI\_WIN\_UNLOCK or MPI\_WIN\_UNLOCK\_ALL, respectively.

Figure 11.2 illustrates the general synchronization pattern for active target communi-41 cation. The synchronization between **post** and **start** ensures that the put call of the origin 42process does not start until the target process exposes the window (with the **post** call); 43 the target process will expose the window only after preceding local accesses to the window 44have completed. The synchronization between complete and wait ensures that the put call 45of the origin process completes before the window is unexposed (with the wait call). The 4647target process will execute following local accesses to the target window only after the wait returned. 48

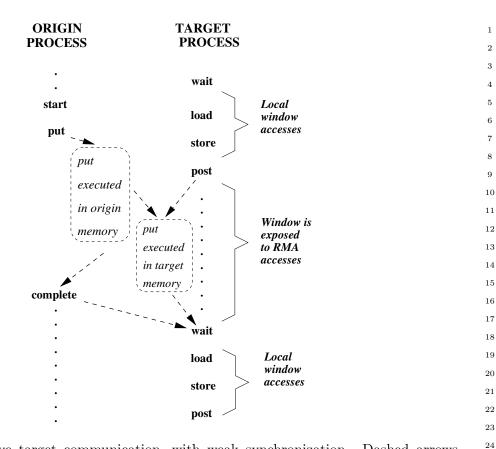


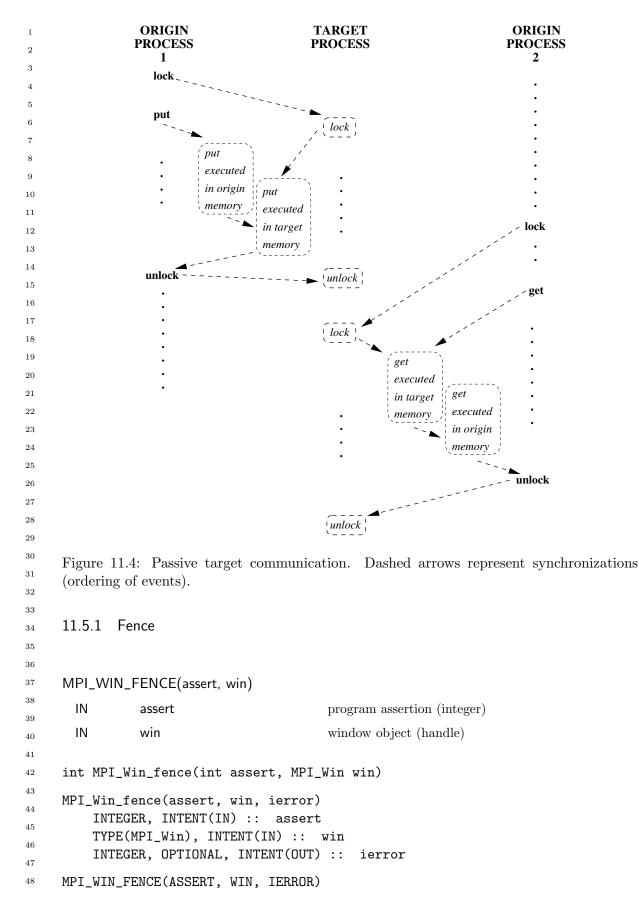
Figure 11.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

Figure 11.2 shows operations occurring in the natural temporal order implied by the synchronizations: the post occurs before the matching start, and complete occurs before the matching wait. However, such strong synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow weak synchronization, as illustrated in Figure 11.3. The access to the target window is delayed until the window is exposed, after the post. However the start may complete earlier; the put and complete may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 11.4 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The lock and unlock calls ensure that the two RMA accesses do not occur concurrently. However, they do *not* ensure that the put by origin 1 will precede the get by origin 2.

*Rationale.* RMA does not define fine-grained mutexes in memory (only logical coarsegrained process locks). MPI provides the primitives (compare and swap, accumulate, send/receive, etc.) needed to implement high-level synchronization operations. (*End* of rationale.)

## Unofficial Draft for Comment Only



Unofficial Draft for Comment Only

#### INTEGER ASSERT, WIN, IERROR

The MPI call MPI\_WIN\_FENCE(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI\_WIN\_FENCE has been called by the target process.

The call completes an RMA access epoch if it was preceded by another fence call and the local process issued RMA communication calls on win between these two calls. The call completes an RMA exposure epoch if it was preceded by another fence call and the local window was the target of RMA accesses between these two calls. The call starts an RMA access epoch if it is followed by another fence call and by RMA communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and the local window is the target of RMA accesses between these two fence calls. Thus, the fence call is equivalent to calls to a subset of post, start, complete, wait.

A fence call usually entails a barrier synchronization: a process completes a call to MPI\_WIN\_FENCE only after all other processes in the group entered their matching call. However, a call to MPI\_WIN\_FENCE that is known not to end any epoch (in particular, a call with assert equal to MPI\_MODE\_NOPRECEDE) does not necessarily act as a barrier.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.5.5. A value of assert = 0 is always valid.

Advice to users. Calls to MPI\_WIN\_FENCE should both precede and follow calls to RMA communication functions that are synchronized with fence calls. (*End of advice to users.*)

#### 11.5.2 General Active Target Synchronization

MPI\_WIN\_START(group, assert, win)

IN	group	group of target processes (handle)	33
IN	assert	program assertion (integer)	34
IN	win	window object (handle)	35
		J ( )	36
int MD	T Win atomt (MDT	_Group group, int assert, MPI_Win win)	37
IIIC MF.	I_WIH_Start(MFI	_Group group, int assert, MFI_win win)	38
MPI_Wi	n_start(group,	assert, win, ierror)	39
TY	PE(MPI_Group),	INTENT(IN) :: group	40
IN	FEGER, INTENT(I	N) :: assert	41
TY	PE(MPI_Win), IN	TENT(IN) :: win	42
IN	TEGER, OPTIONAL	, INTENT(OUT) :: ierror	43
NOT UT			44
_		ASSERT, WIN, IERROR)	45
TN.	TEGER GROUP, AS	SERT, WIN, IERROR	46

Starts an RMA access epoch for win. RMA calls issued on win during this epoch must 47 access only windows at processes in group. Each process in group must issue a matching 48

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

1call to MPI\_WIN\_POST. RMA accesses to each target window will be delayed, if necessary,  $\mathbf{2}$ until the target process executed the matching call to MPI\_WIN\_POST. MPI\_WIN\_START 3 is allowed to block until the corresponding MPI\_WIN\_POST calls are executed, but is not 4 required to. 5The assert argument is used to provide assertions on the context of the call that may 6 be used for various optimizations. This is described in Section 11.5.5. A value of assert =  $\overline{7}$ 0 is always valid. 8 9 MPI\_WIN\_COMPLETE(win) 10 11IN window object (handle) win 1213int MPI\_Win\_complete(MPI\_Win win) 14MPI\_Win\_complete(win, ierror) 15TYPE(MPI\_Win), INTENT(IN) :: win 16INTEGER, OPTIONAL, INTENT(OUT) :: 17ierror 18 MPI\_WIN\_COMPLETE(WIN, IERROR) 19INTEGER WIN, IERROR 2021Completes an RMA access epoch on win started by a call to MPI\_WIN\_START. All RMA communication calls issued on win during this epoch will have completed at the origin 22 when the call returns. 23MPI\_WIN\_COMPLETE enforces completion of preceding RMA calls at the origin, but  $^{24}$ not at the target. A put or accumulate call may not have completed at the target when it 2526has completed at the origin. Consider the sequence of calls in the example below. 2728Example 11.4 2930 MPI\_Win\_start(group, flag, win);  $^{31}$ MPI\_Put(..., win); 32 MPI\_Win\_complete(win); 33 34The call to MPI\_WIN\_COMPLETE does not return until the put call has completed 35 at the origin; and the target window will be accessed by the put operation only after the 36 call to MPI\_WIN\_START has matched a call to MPI\_WIN\_POST by the target process. 37 This still leaves much choice to implementors. The call to MPI\_WIN\_START can block 38 until the matching call to MPI\_WIN\_POST occurs at all target processes. One can also 39 have implementations where the call to MPI\_WIN\_START is nonblocking, but the call to 40MPI\_PUT blocks until the matching call to MPI\_WIN\_POST occurs; or implementations 41 where the first two calls are nonblocking, but the call to MPI\_WIN\_COMPLETE blocks 42until the call to MPI\_WIN\_POST occurred; or even implementations where all three calls 43can complete before any target process has called MPI\_WIN\_POST — the data put must

<sup>44</sup> be buffered, in this last case, so as to allow the put to complete at the origin ahead of its <sup>45</sup> completion at the target. However, once the call to MPI\_WIN\_POST is issued, the sequence <sup>46</sup> above must complete, without further dependencies.

MPI_WIN_POST(group, assert, win)			
IN group	group of origin processes (handle)	2	
IN assert	program assertion (integer)	$\frac{3}{4}$	
IN win	window object (handle)	5	
IN WIII	window object (nandle)	6	
int MDI Win negt (MDI Crown mour	int accort MDI Vin vin)	7	
<pre>int MPI_Win_post(MPI_Group group,</pre>	int assert, MPI_win Win)	8	
<pre>MPI_Win_post(group, assert, win, :</pre>	ierror)	9	
<pre>TYPE(MPI_Group), INTENT(IN) :</pre>		10	
INTEGER, INTENT(IN) :: asser		11	
TYPE(MPI_Win), INTENT(IN) ::		12	
INTEGER, OPTIONAL, INTENT(OUT)	) :: ierror	13	
MPI_WIN_POST(GROUP, ASSERT, WIN, 1	IERROR)	14	
INTEGER GROUP, ASSERT, WIN, I	ERROR	15 16	
Starts an $RMA$ exposure epoch for the	a local window associated with win Only processes	16	
Starts an RMA exposure epoch for the local window associated with win. Only processes in group should access the window with RMA calls on win during this epoch. Each process			
in group must issue a matching call to MPI_WIN_START. MPI_WIN_POST does not block.			
0		20	
		21	
MPI_WIN_WAIT(win)		22	
IN win	window object (handle)	23	
		24	
int MPI_Win_wait(MPI_Win win)		25	
MPI_Win_wait(win, ierror)		26	
TYPE(MPI_Win), INTENT(IN) ::	win	27	
INTEGER, OPTIONAL, INTENT(OUT)		28	
		29 30	
MPI_WIN_WAIT(WIN, IERROR)		31	
INTEGER WIN, IERROR		32	
Completes an RMA exposure epoch	started by a call to $MPI_WIN_POST$ on win. This	33	
call matches calls to $MPI\_WIN\_COMPLE$	<b>TE</b> (win) issued by each of the origin processes that	34	

completes an RMA exposure epoch started by a call to MPI\_WIN\_POST on WM. This call matches calls to MPI\_WIN\_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. The call to MPI\_WIN\_WAIT will block until all matching calls to MPI\_WIN\_COMPLETE have occurred. This guarantees that all these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

Figure 11.5 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start, put or complete calls may occur ahead of the matching post calls.

 $45 \\ 46$ 

```
PROCESS 0
                                   PROCESS 1
                                                     PROCESS 2
                                                                        PROCESS 3
1
2
3
4
                  start(1.2)
                                                                           start(2)
5
6
                    put(1)
                                                                           put(2)
7
                    put(2)
8
9
                  complete()
                                                                         complete()
10
11
12
                                      wait()
                                                        wait()
13
14
      Figure 11.5: Active target communication. Dashed arrows represent synchronizations and
15
     solid arrows represent data transfer.
16
17
      MPI_WIN_TEST(win, flag)
18
19
       IN
                 win
                                               window object (handle)
20
        OUT
                 flag
                                               success flag (logical)
21
22
      int MPI_Win_test(MPI_Win win, int *flag)
23
^{24}
     MPI_Win_test(win, flag, ierror)
25
          TYPE(MPI_Win), INTENT(IN) ::
                                              win
26
          LOGICAL, INTENT(OUT) :: flag
27
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                    ierror
28
     MPI_WIN_TEST(WIN, FLAG, IERROR)
29
          INTEGER WIN, IERROR
30
          LOGICAL FLAG
^{31}
32
          This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if all accesses
33
      to the local window by the group to which it was exposed by the corresponding
34
      MPI_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE
35
      calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned
36
      immediately. The effect of return of MPI_WIN_TEST with flag = true is the same as the
37
      effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible
38
      effect.
39
          MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once
40
      the call has returned flag = true, it must not be invoked anew, until the window is posted
41
      anew.
42
          Assume that window win is associated with a "hidden" communicator wincomm, used
43
      for communication by the processes of win. The rules for matching of post and start calls
44
      and for matching complete and wait calls can be derived from the rules for matching sends
45
      and receives, by considering the following (partial) model implementation.
46
47
      MPI_WIN_POST(group,0,win) initiates a nonblocking send with tag tag0 to each process
48
           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.
- **MPI\_WIN\_COMPLETE(win)** initiates a nonblocking send with tag tag1 to each process in the group of the preceding start call. No need to wait for the completion of these sends.
- **MPI\_WIN\_WAIT(win)** initiates a nonblocking receive with tag **tag1** from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice versa.

*Rationale.* The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph  $G = \langle V, E \rangle$ , where  $V = \{0, \ldots, n-1\}$  and  $ij \in E$  if origin process *i* accesses the window at target process *j*. Then each process *i* issues a call to MPI\_WIN\_POST(*ingroup*<sub>i</sub>, ...), followed by a call to MPI\_WIN\_START(*outgroup*<sub>i</sub>,...), where *outgroup*<sub>i</sub> =  $\{j : ij \in E\}$  and *ingroup*<sub>i</sub> =

 $\{j : ji \in E\}$ . A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait.

Note that each process may call with a group argument that has different members. (*End of advice to users.*)

## 11.5.3 Lock

MPI\_WIN\_LOCK(lock\_type, rank, assert, win)

IN	lock_type	either $MPI\_LOCK\_EXCLUSIVE$ or	41
		MPI_LOCK_SHARED (state)	42
IN	rank	rank of locked window (non-negative integer)	43
IN	assert	program assertion (integer)	44
IIN	asser 1	program assertion (integer)	45
IN	win	window object (handle)	46
			47

int MPI\_Win\_lock(int lock\_type, int rank, int assert, MPI\_Win win)

#### **Unofficial Draft for Comment Only**

 $^{31}$ 

```
1
     MPI_Win_lock(lock_type, rank, assert, win, ierror)
\mathbf{2}
          INTEGER, INTENT(IN) :: lock_type, rank, assert
3
          TYPE(MPI_Win), INTENT(IN) :: win
4
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
5
     MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
6
          INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
7
8
          Starts an RMA access epoch. The window at the process with rank rank can be accessed
9
     by RMA operations on win during that epoch. Multiple RMA access epochs (with calls
10
     to MPI_WIN_LOCK) can occur simultaneously; however, each access epoch must target a
11
     different process.
12
13
     MPI_WIN_LOCK_ALL(assert, win)
14
15
       IN
                                            program assertion (integer)
                 assert
16
       IN
                 win
                                             window object (handle)
17
18
     int MPI_Win_lock_all(int assert, MPI_Win win)
19
20
     MPI_Win_lock_all(assert, win, ierror)
21
          INTEGER, INTENT(IN) :: assert
22
          TYPE(MPI_Win), INTENT(IN) :: win
23
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
24
     MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
25
          INTEGER ASSERT, WIN, IERROR
26
27
         Starts an RMA access epoch to all processes in win, with a lock type of
28
     MPI_LOCK_SHARED. During the epoch, the calling process can access the window memory on
29
     all processes in win by using RMA operations. A window locked with MPI_WIN_LOCK_ALL
30
     must be unlocked with MPI_WIN_UNLOCK_ALL. This routine is not collective — the ALL
^{31}
     refers to a lock on all members of the group of the window.
32
33
           Advice to users.
                             There may be additional overheads associated with using
34
           MPI_WIN_LOCK and MPI_WIN_LOCK_ALL concurrently on the same window. These
35
          overheads could be avoided by specifying the assertion MPI_MODE_NOCHECK when
36
           possible (see Section 11.5.5). (End of advice to users.)
37
38
39
     MPI_WIN_UNLOCK(rank, win)
40
41
       IN
                                            rank of window (non-negative integer)
                 rank
42
       IN
                                             window object (handle)
                 win
43
44
     int MPI_Win_unlock(int rank, MPI_Win win)
45
46
     MPI_Win_unlock(rank, win, ierror)
47
          INTEGER, INTENT(IN) :: rank
48
          TYPE(MPI_Win), INTENT(IN) ::
                                          win
```

## Unofficial Draft for Comment Only

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
	2
MPI_WIN_UNLOCK(RANK, WIN, IERROR)	3
INTEGER RANK, WIN, IERROR	4
Completes an RMA access epoch started by a call to MPI_WIN_LOCK on window win.	5
RMA operations issued during this period will have completed both at the origin and at the	6
target when the call returns.	7
	8
	9
MPI_WIN_UNLOCK_ALL(win)	10
IN win window object (handle)	11
	12
int MPI_Win_unlock_all(MPI_Win win)	13
	14
MPI_Win_unlock_all(win, ierror)	15
TYPE(MPI_Win), INTENT(IN) :: win	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
MPI_WIN_UNLOCK_ALL(WIN, IERROR)	18
INTEGER WIN, IERROR	19
	20
Completes a shared RMA access epoch started by a call to $MPI_WIN_LOCK_ALL$ on	21
window win. RMA operations issued during this epoch will have completed both at the	22

window win. RMA operations issued during this epoch will have completed both at the origin and at the target when the call returns.

Locks are used to protect accesses to the locked target window effected by RMA calls issued between the lock and unlock calls, and to protect load/store accesses to a locked local or shared memory window executed between the lock and unlock calls. Accesses that are protected by an exclusive lock will not be concurrent at the window site with other accesses to the same window that are lock protected. Accesses that are protected by a shared lock will not be concurrent at the window site with accesses protected by an exclusive lock to the same window.

It is erroneous to have a window locked and exposed (in an exposure epoch) concurrently. For example, a process may not call MPI\_WIN\_LOCK to lock a target window if the target process has called MPI\_WIN\_POST and has not yet called MPI\_WIN\_WAIT; it is erroneous to call MPI\_WIN\_POST while the local window is locked.

*Rationale.* An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

Advice to users. Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (End of advice to users.)

47 48

23 24

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35 36

37

38

39

40

41

42

43 44

45

46

#### Unofficial Draft for Comment Only

1 2 3 4 5	lock calls MPI_WI	s to windows in N_ALLOCATE (	memory allocat (Section $11.2.2$ ), o	of RMA communication that is synchronized by ed by MPI_ALLOC_MEM (Section 8.2), or attached with MPI_WIN_ATTACH ably only in such memory.
5 6 7 8 9 10 11	is m imj alle nat	not shared may plemented more ocated memory cural to impose	require an asyn easily, and can ac It can be avoide	of passive target communication when memory chronous software agent. Such an agent can be chieve better performance, if restricted to specially ed altogether if shared memory is used. It seems allows one to use shared memory for third party machines.
12 13	(Ei	nd of rationale.	)	
14 15	Con	sider the seque	nce of calls in the	example below.
16	Exampl	e 11.5		
17 18 19 20	MPI_Put	_lock(MPI_LOC (, rank, . _unlock(rank,	, win);	ank, assert, win);
21 22 23 24 25 26 27 28 29	the origin MPI_WII two calls update o However block un	n and at the ta $N_LOCK$ may b may not block f the target win , if the call to $N$ til the lock is a	rget. This still le lock until an excl a, while MPI_WIN dow is then postp MPI_WIN_LOCK i	not return until the put transfer has completed at aves much freedom to implementors. The call to usive lock on the window is acquired; or, the first LUNLOCK blocks until a lock is acquired — the boned until the call to MPI_WIN_UNLOCK occurs. s used to lock a local window, then the call must lock may protect local load/store accesses to the
30 31	11.5.4	Flush and Syno	:	
32 33	All flush	and sync funct	ions can be called	l only within passive target epochs.
34 35	MPI_WI	N_FLUSH(rank,	win)	
36	IN	rank		rank of target window (non-negative integer)
37 38	IN	win		window object (handle)
39 40	int MPI	_Win_flush(in	t rank, MPI_Win	n win)
41 42 43 44 45 46 47	INT TYPI INT MPI_WIN	-	IN) :: rank NTENT(IN) :: L, INTENT(OUT) WIN, IERROR)	
48				

MPI\_WIN\_FLUSH completes all outstanding RMA operations initiated by the calling process to the target rank on the specified window. The operations are completed both at the origin and at the target.

MPI_W	IN_FLUSH_ALL(wi	in)	5 6
IN	win	window object (handle)	7
			8
int MP	I_Win_flush_all(	(MPT Win win)	9
			10
	n_flush_all(win,		11
	PE(MPI_Win), INT		12
ΞN	IEGER, UPIIUNAL,	, INTENT(OUT) :: ierror	13
MPI_WI	N_FLUSH_ALL(WIN,	, IERROR)	14
IN	TEGER WIN, IERRO	JR	15 16
All	RMA operations is	ssued by the calling process to any target on the specified window	17
	-	e specified window will have completed both at the origin and at	18
-	get when this call 1	• • • •	19
	5		20
			21
MPI_W	IN_FLUSH_LOCAL	(rank, win)	22
IN	rank	rank of target window (non-negative integer)	23
IN	win	window object (handle)	24
			25
int MP	I_Win_flush_loca	al(int rank, MPI_Win win)	26
			27
		ank, win, ierror)	28
	TEGER, INTENT(IN		29 30
	PE(MPI_Win), INT	<pre>FENT(IN) :: win , INTENT(OUT) :: ierror</pre>	31
TN	ILGER, OFIIONAL,	, INIENI(UUI) IEIIUI	32
		ANK, WIN, IERROR)	33
IN	TEGER RANK, WIN	, IERROR	34
Lo	cally completes at	the origin all outstanding RMA operations initiated by the calling	35
	<i>v</i> •	ess specified by rank on the specified window. For example, after	36
		e user may reuse any buffers provided to put, get, or accumulate	37
operati			38
			39
	IN_FLUSH_LOCAL	$\Delta I I (win)$	40
			41
IN	win	window object (handle)	42
			43 44
int MP	I_Win_flush_loca	al_all(MPI_Win win)	44
MPI_Wi	n_flush_local_al	ll(win, ierror)	46
		TENT(IN) :: win	47
IN	TEGER, OPTIONAL,	, INTENT(OUT) :: ierror	48

**Unofficial Draft for Comment Only** 

1

 $\frac{2}{3}$ 

1 MPI\_WIN\_FLUSH\_LOCAL\_ALL(WIN, IERROR)  $\mathbf{2}$ INTEGER WIN, IERROR 3 All RMA operations issued to any target prior to this call in this window will have 4 completed at the origin when MPI\_WIN\_FLUSH\_LOCAL\_ALL returns. 56  $\overline{7}$ MPI\_WIN\_SYNC(win) 8 IN window object (handle) win 9 10 int MPI\_Win\_sync(MPI\_Win win) 1112MPI\_Win\_sync(win, ierror) 13 TYPE(MPI\_Win), INTENT(IN) :: win 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516MPI\_WIN\_SYNC(WIN, IERROR) 17INTEGER WIN, IERROR 18 The call MPI\_WIN\_SYNC synchronizes the private and public window copies of win. 19For the purposes of synchronizing the private and public window, MPI\_WIN\_SYNC has the 20effect of ending and reopening an access and exposure epoch on the window (note that it 21does not actually end an epoch or complete any pending MPI RMA operations). 222311.5.5 Assertions  $^{24}$ 25The assert argument in the calls MPI\_WIN\_POST, MPI\_WIN\_START, MPI\_WIN\_FENCE, 26MPI\_WIN\_LOCK, and MPI\_WIN\_LOCK\_ALL is used to provide assertions on the context of 27the call that may be used to optimize performance. The assert argument does not change 28 program semantics if it provides correct information on the program — it is erroneous to 29provide incorrect information. Users may always provide assert = 0 to indicate a general 30 case where no guarantees are made. 31Advice to users. Many implementations may not take advantage of the information 32 in assert; some of the information is relevant only for noncoherent shared memory ma-33 chines. Users should consult their implementation's manual to find which information 34 is useful on each system. On the other hand, applications that provide correct asser-35tions whenever applicable are portable and will take advantage of assertion specific 36 optimizations whenever available. (End of advice to users.) 37 38 Implementations can always ignore the Advice to implementors. 39 assert argument. Implementors should document which assert values are significant 40 on their implementation. (End of advice to implementors.) 41 42assert is the bit-vector OR of zero or more of the following integer constants: 43 MPI\_MODE\_NOCHECK, MPI\_MODE\_NOSTORE, MPI\_MODE\_NOPUT, 44MPI\_MODE\_NOPRECEDE, and MPI\_MODE\_NOSUCCEED. The significant options are listed 45below for each call. 46 47Advice to users. C/C++ users can use bit vector or () to combine these constants; 48 Fortran 90 users can use the bit-vector IOR intrinsic. Alternatively, Fortran users can

## Unofficial Draft for Comment Only

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:

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

- MPI\_MODE\_NOCHECK the matching calls to MPI\_WIN\_START have not yet occurred on any origin processes when the call to MPI\_WIN\_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
- MPI\_MODE\_NOSTORE the local window was not updated by stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.
- MPI\_MODE\_NOPUT the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.

## MPI\_WIN\_FENCE:

- MPI\_MODE\_NOSTORE the local window was not updated by stores (or local get or receive calls) since last synchronization.
- MPI\_MODE\_NOPUT the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
- MPI\_MODE\_NOPRECEDE the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.
- MPI\_MODE\_NOSUCCEED the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.

## MPI\_WIN\_LOCK, MPI\_WIN\_LOCK\_ALL:

MPI\_MODE\_NOCHECK — no other process holds, or will attempt to acquire, a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

Advice to users. Note that the nostore and noprecede flags provide information on 46 what happened *before* the call; the noput and nosucceed flags provide information on 47 what will happen *after* the call. (*End of advice to users.*) 48

## Unofficial Draft for Comment Only

1 2

3

4 5

6

 $\overline{7}$ 

8

9

10

# 11.5.6 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the datatype argument of a MPI\_PUT call can be freed as soon as the call returns, even though the communication may not be complete.

As in message-passing, datatypes must be committed before they can be used in RMA communication.

7 8 9

10

11

17

18 19

20

24

48

1

 $\mathbf{2}$ 

3

4

5

6

# 11.6 Error Handling

11.6.1 Error Handlers

Errors occurring during calls to routines that create MPI windows (e.g., MPI\_WIN\_CREATE (...,comm,...)) cause the error handler currently associated with comm to be invoked. All other RMA calls have an input win argument. When an error occurs during such a call, the error handler currently associated with win is invoked.

The default error handler associated with win is MPI\_ERRORS\_ARE\_FATAL. Users may change this default by explicitly associating a new error handler with win (see Section 8.3).

11.6.2 Error Classes

The error classes for one-sided communication are defined in Table 11.2. RMA routines may
 (and almost certainly will) use other MPI error classes, such as MPI\_ERR\_OP or
 MPI\_ERR\_RANK.

24		
25	MPI_ERR_WIN	invalid win argument
26	MPI_ERR_BASE	invalid base argument
27	MPI_ERR_SIZE	invalid size argument
28	MPI_ERR_DISP	invalid disp argument
29	MPI_ERR_LOCKTYPE	invalid locktype argument
30	MPI_ERR_ASSERT	invalid assert argument
31	MPI_ERR_RMA_CONFLICT	conflicting accesses to window
32	MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls
33	MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case
34		of a window created with
35		MPI_WIN_CREATE_DYNAMIC, target memory is not
36		attached)
37	MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource
38		exhaustion)
39	MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the
40		group of the specified communicator cannot expose
41		shared memory)
42	MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called
43		function
44		
45	Table 11.9. Free	classes in one-sided communication routines
46	Table 11.2: Effor	classes in one-sided communication routilles
47		

# 11.7 Semantics and Correctness

The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is visible when the get operation is complete at the origin (or earlier); the update performed by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier). The rules also specify the latest time at which an update of one window copy becomes visible in another overlapping copy.

- An RMA operation is completed at the origin by the ensuing call to MPI\_WIN\_COMPLETE, MPI\_WIN\_FENCE, MPI\_WIN\_FLUSH, MPI\_WIN\_FLUSH\_ALL, MPI\_WIN\_FLUSH\_LOCAL, MPI\_WIN\_FLUSH\_LOCAL\_ALL, MPI\_WIN\_UNLOCK, or MPI\_WIN\_UNLOCK\_ALL that synchronizes this access at the origin.
- 2. If an RMA operation is completed at the origin by a call to MPI\_WIN\_FENCE then the operation is completed at the target by the matching call to MPI\_WIN\_FENCE by the target process.
- 3. If an RMA operation is completed at the origin by a call to MPI\_WIN\_COMPLETE then the operation is completed at the target by the matching call to MPI\_WIN\_WAIT by the target process.
- 4. If an RMA operation is completed at the origin by a call to MPI\_WIN\_UNLOCK, MPI\_WIN\_UNLOCK\_ALL, MPI\_WIN\_FLUSH(rank=target), or MPI\_WIN\_FLUSH\_ALL, then the operation is completed at the target by that same call.
- 5. An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to MPI\_WIN\_POST, MPI\_WIN\_FENCE, MPI\_WIN\_UNLOCK, MPI\_WIN\_UNLOCK\_ALL, or MPI\_WIN\_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update of a location in a private window in process memory becomes visible without additional RMA calls.
- 6. An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to MPI\_WIN\_WAIT, MPI\_WIN\_FENCE, MPI\_WIN\_LOCK, MPI\_WIN\_LOCK\_ALL, or MPI\_WIN\_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update by a put or accumulate call to a public window copy eventually becomes visible in the private copy in process memory without additional RMA calls.

The MPI\_WIN\_FENCE or MPI\_WIN\_WAIT call that completes the transfer from public copy to private copy (6) is the same call that completes the put or accumulate operation in the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then the update of the public window copy is complete as soon as the updating process executed MPI\_WIN\_UNLOCK or MPI\_WIN\_UNLOCK\_ALL. In the RMA separate memory model, the update of a private copy in the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed in the RMA separate memory model until the process executes a suitable 

# Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

synchronization call, while they must complete in the RMA unified model without additional
 synchronization calls. If fence or post-start-complete-wait synchronization is used, updates
 to a public window copy can be delayed in both memory models until the window owner
 executes a synchronization call. When passive target synchronization is used, it is necessary
 to update the public window copy even if the window owner does not execute any related
 synchronization call.

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI\_WIN\_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI\_WIN\_FENCE(0, win2) makes these updates visible in the public copy of win2.

13 The behavior of some MPI RMA operations may be *undefined* in certain situations. For 14example, the result of several origin processes performing concurrent MPI\_PUT operations 15to the same target location is undefined. In addition, the result of a single origin process 16performing multiple MPI\_PUT operations to the same target location within the same 17access epoch is also undefined. The result at the target may have all of the data from one 18 of the MPI\_PUT operations (the "last" one, in some sense), bytes from some of each of the 19operations, or something else. In MPI-2, such operations were *erroneous*. That meant that 20an MPI implementation was permitted to signal an MPI exception. Thus, user programs or 21tools that used MPI RMA could not portably permit such operations, even if the application 22code could function correctly with such an undefined result. In MPI-3, these operations are 23not erroneous, but do not have a defined behavior.

 $^{24}$ Rationale. As discussed in [6], requiring operations such as overlapping puts to 25be erroneous makes it difficult to use MPI RMA to implement programming models— 26such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further, 27while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any 28 implementation that enforces this rule, as it would require significant overhead. Thus, 29 relaxing this condition does not impact existing implementations or applications. (End 30 of rationale.) 31

Advice to implementors. Overlapping accesses are undefined. However, to assist users in debugging code, implementations may wish to provide a mode in which such operations are detected and reported to the user. Note, however, that in MPI-3, such operations must not generate an MPI exception. (*End of advice to implementors.*)

A program with a well-defined outcome in the MPI\_WIN\_SEPARATE memory model must obey the following rules.

- S1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update becomes visible in the private window copy in process memory.
- S2. A location in a window must not be accessed as a target of an RMA operation once
   an update to that location has started, until the update becomes visible in the public
   window copy. There is one exception to this rule, in the case where the same variable
   is updated by two concurrent accumulates with the same predefined datatype, on
   the same window. Additional restrictions on the operation apply, see the info key
   accumulate\_ops in Section 11.2.1.

## Unofficial Draft for Comment Only

32

33

34

35

36

37

38 39

40

41

S3. A put or accumulate must not access a target window once a store or a put or accumulate update to another (overlapping) target window has started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a store to process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.

*Rationale.* The last constraint on correct RMA accesses may seem unduly restrictive, as it forbids concurrent accesses to nonoverlapping locations in a window. The reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may be needed for locations that were updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library would have to track precisely which locations in a window were updated by a put or accumulate call. The additional overhead of maintaining such information is considered prohibitive. (*End of rationale.*)

Note that MPI\_WIN\_SYNC may be used within a passive target epoch to synchronize the private and public window copies (that is, updates to one are made visible to the other).

In the MPI\_WIN\_UNIFIED memory model, the rules are simpler because the public and private windows are the same. However, there are restrictions to avoid concurrent access to the same memory locations by different processes. The rules that a program with a well-defined outcome must obey in this case are:

- U1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update is complete, subject to the following special case.
- U2. Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory of the target, the data remains until replaced by another update. This permits polling on a location for a change from zero to non-zero or for a particular value, but not polling and comparing the relative magnitude of values. Users are cautioned that polling on one memory location and then accessing a different memory location has defined behavior only if the other rules given here and in this chapter are followed.

Advice to users. Some compiler optimizations can result in code that maintains the sequential semantics of the program, but violates this rule by introducing temporary values into locations in memory. Most compilers only apply such transformations under very high levels of optimization and users should be aware that such aggressive optimization may produce unexpected results. (*End of advice to users.*)

U3. Updating a location in the window with a store operation that is also the target of a remote read (but not update) is valid (not erroneous) but the precise result

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9

10

11

12

13

14

15

16

17 18

19

20

21

22

23

24 25

26

27

28 29

30

 $^{31}$ 

32

33

34

35

36

37

38

39 40

41

42

43

44

45

 $46 \\ 47$ 

will depend on the behavior of the implementation. Store updates will appear in memory, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory, the data remains until replaced by another update. This permits updates to memory with store operations without requiring an RMA epoch. Users are cautioned that remote accesses to a window that is updated by the local process has defined behavior only if the other rules given here and elsewhere in this chapter are followed.

- U4. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started and until the update completes at the target. There is one exception to this rule: in the case where the same location is updated by two concurrent accumulates with the same predefined datatype on the same window. Additional restrictions on the operation apply; see the info key accumulate\_ops in Section 11.2.1.
- 15U5. A put or accumulate must not access a target window once a store, put, or accumulate 16update to another (overlapping) target window has started on the same location in 17 the target window and until the update completes at the target window. Conversely, 18 a store operation to a location in a window must not start once a put or accumulate 19 update to the same location in that target window has started and until the put or 20accumulate update completes at the target. 21

In the unified memory model, in the case where the window is Advice to users. in shared memory, MPI\_WIN\_SYNC can be used to order store operations and make store updates to the window visible to other processes and threads. Use of this routine is necessary to ensure portable behavior when point-to-point, collective, or shared memory synchronization is used in place of an RMA synchronization routine. MPI\_WIN\_SYNC should be called by the writer before the non-RMA synchronization operation and by the reader after the non-RMA synchronization, as shown in Example 11.21. (End of advice to users.)

- 30 A program that violates these rules has undefined behavior. 31
  - Advice to users. A user can write correct programs by following the following rules:
  - fence: During each period between fence calls, each window is either updated by put or accumulate calls, or updated by stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.
- post-start-complete-wait: A window should not be updated with store operations 39 40 while posted if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted 42(with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window 43 is posted. 44
- 45With the post-start synchronization, the target process can tell the origin process 46that its window is now ready for RMA access; with the complete-wait synchro-47 nization, the origin process can tell the target process that it has finished its 48 RMA accesses to the window.

# **Unofficial Draft for Comment Only**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

22

23

 $^{24}$ 

2526

27

28

29

32

33

34

35

36

37

38

- **lock:** Updates to the window are protected by exclusive locks if they may conflict. Nonconflicting accesses (such as read-only accesses or accumulate accesses) are protected by shared locks, both for load/store accesses and for RMA accesses.
- changing window or synchronization mode: One can change synchronization mode, or change the window used to access a location that belongs to two overlapping windows, when the process memory and the window copy are guaranteed to have the same values. This is true after a local call to MPI\_WIN\_FENCE, if RMA accesses to the window are synchronized with fences; after a local call to MPI\_WIN\_WAIT, if the accesses are synchronized with post-start-completewait; after the call at the origin (local or remote) to MPI\_WIN\_UNLOCK or MPI\_WIN\_UNLOCK\_ALL if the accesses are synchronized with locks.

In addition, a process should not access the local buffer of a get operation until the operation is complete, and should not update the local buffer of a put or accumulate operation until that operation is complete.

The RMA synchronization operations define when updates are guaranteed to become visible in public and private windows. Updates may become visible earlier, but such behavior is implementation dependent. (*End of advice to users.*)

The semantics are illustrated by the following examples:

**Example 11.6** The following example demonstrates updating a memory location inside a window for the separate memory model, according to Rule 5. The MPI\_WIN\_LOCK and MPI\_WIN\_UNLOCK calls around the store to X in process B are necessary to ensure consistency between the public and private copies of the window.

		26
Process A:	Process B:	27
	window location X	28
		29
	MPI_Win_lock(EXCLUSIVE,B)	30
	store X /* local update to private copy of B */	31
	MPI_Win_unlock(B)	32
	<pre>/* now visible in public window copy */</pre>	33
		34
MPI_Barrier	MPI_Barrier	35
		36
MPI_Win_lock(EXCLUSIVE,B)		37
MPI_Get(X) /* ok, read fro	m public window */	38
MPI_Win_unlock(B)		39

**Example 11.7** In the RMA unified model, although the public and private copies of the windows are synchronized, caution must be used when combining load/stores and multi-process synchronization. Although the following example appears correct, the compiler or hardware may delay the store to X after the barrier, possibly resulting in the MPI\_GET returning an incorrect value of X.

 $\mathbf{2}$ 

 $\overline{7}$ 

```
1
     Process A:
                                Process B:
\mathbf{2}
                                window location X
3
4
                                store X /* update to private & public copy of B */
\mathbf{5}
     MPI_Barrier
                                MPI_Barrier
6
     MPI_Win_lock_all
7
     MPI_Get(X) /* ok, read from window */
8
     MPI_Win_flush_local(B)
9
     /* read value in X */
10
     MPI_Win_unlock_all
11
     MPI_BARRIER provides process synchronization, but not memory synchronization. The
12
     example could potentially be made safe through the use of compiler- and hardware-specific
13
     notations to ensure the store to X occurs before process B enters the MPI_BARRIER. The
14
     use of one-sided synchronization calls, as shown in Example 11.6, also ensures the correct
15
16
     result.
17
     Example 11.8 The following example demonstrates the reading of a memory location
18
     updated by a remote process (Rule 6) in the RMA separate memory model. Although the
19
     MPI_WIN_UNLOCK on process A and the MPI_BARRIER ensure that the public copy on
20
     process B reflects the updated value of X, the call to MPI_WIN_LOCK by process B is
21
     necessary to synchronize the private copy with the public copy.
22
23
     Process A:
                                    Process B:
24
                                    window location X
25
26
     MPI_Win_lock(EXCLUSIVE,B)
27
     MPI_Put(X) /* update to public window */
28
     MPI_Win_unlock(B)
29
30
     MPI_Barrier
                                    MPI_Barrier
31
32
                                    MPI_Win_lock(EXCLUSIVE,B)
33
                                     /* now visible in private copy of B */
34
                                     load X
35
                                    MPI_Win_unlock(B)
36
37
     Note that in this example, the barrier is not critical to the semantic correctness. The
38
     use of exclusive locks guarantees a remote process will not modify the public copy after
39
     MPI_WIN_LOCK synchronizes the private and public copies. A polling implementation
40
     looking for changes in X on process B would be semantically correct. The barrier is required
41
     to ensure that process A performs the put operation before process B performs the load of
42
     Х.
43
     Example 11.9 Similar to Example 11.7, the following example is unsafe even in the unified
44
     model, because the load of X can not be guaranteed to occur after the MPI_BARRIER. While
45
     Process B does not need to explicitly synchronize the public and private copies through
46
     MPI_WIN_LOCK as the MPI_PUT will update both the public and private copies of the
47
     window, the scheduling of the load could result in old values of X being returned. Compiler
```

and hardware specific notations could ensure the load occurs after the data is updated, or explicit one-sided synchronization calls can be used to ensure the proper result.

Process A:	Process B: window location X
MPI_Win_lock_all	
MPI_Put(X) /* update to with	ndow */
MPI_Win_flush(B)	
MPI_Barrier	MPI_Barrier
	load X
MPI_Win_unlock_all	

**Example 11.10** The following example further clarifies Rule 5. MPI\_WIN\_LOCK and MPI\_WIN\_LOCK\_ALL do *not* update the public copy of a window with changes to the private copy. Therefore, there is no guarantee that process A in the following sequence will see the value of X as updated by the local store by process B before the lock.

Deve e e e e A		19
Process A:	Process B:	20
	window location X	21
	stone V (thurdate to primete some of D th)	22
	store X /* update to private copy of B */	23
	MPI_Win_lock(SHARED,B)	24
MPI_Barrier	MPI_Barrier	25
		26
MPI_Win_lock(SHARED,B)		27
MPI_Get(X) /* X may be the	X before the store */	28
MPI_Win_unlock(B)		29
	MPI_Win_unlock(B)	30
	<pre>/* update on X now visible in public window */</pre>	31

The addition of an MPI\_WIN\_SYNC before the call to MPI\_BARRIER by process B would guarantee process A would see the updated value of X, as the public copy of the window would be explicitly synchronized with the private copy.

**Example 11.11** Similar to the previous example, Rule 5 can have unexpected implications for general active target synchronization with the RMA separate memory model. It is *not* guaranteed that process B reads the value of X as per the local update by process A, because neither MPI\_WIN\_WAIT nor MPI\_WIN\_COMPLETE calls by process A ensure visibility in the public window copy.

Process A:	Process B:
window location X	
window location Y	
store Y	
MPI_Win_post(A,B) /* Y vis	ible in public window */
MPI_Win_start(A)	MPI_Win_start(A)

Unofficial Draft for Comment Only

```
1
\mathbf{2}
     store X /* update to private window */
3
4
     MPI_Win_complete
                                    MPI_Win_complete
\mathbf{5}
     MPI_Win_wait
6
     /* update on X may not yet visible in public window */
7
8
     MPI_Barrier
                                    MPI_Barrier
9
10
                                     MPI_Win_lock(EXCLUSIVE,A)
11
                                     MPI_Get(X) /* may return an obsolete value */
12
                                     MPI_Get(Y)
13
                                     MPI_Win_unlock(A)
14
     To allow process B to read the value of X stored by A the local store must be replaced by
15
     a local MPI_PUT that updates the public window copy. Note that by this replacement X
16
     may become visible in the private copy of process A only after the MPI_WIN_WAIT call in
17
     process A. The update to Y made before the MPI_WIN_POST call is visible in the public
18
     window after the MPI_WIN_POST call and therefore process B will read the proper value
19
     of Y. The MPI_GET(Y) call could be moved to the epoch started by the MPI_WIN_START
20
     operation, and process B would still get the value stored by process A.
21
22
     Example 11.12 The following example demonstrates the interaction of general active
23
     target synchronization with local read operations with the RMA separate memory model.
24
     Rules 5 and 6 do not guarantee that the private copy of X at process B has been updated
25
     before the load takes place.
26
27
     Process A:
                                     Process B:
28
                                     window location X
29
30
     MPI_Win_lock(EXCLUSIVE,B)
^{31}
     MPI_Put(X) /* update to public window */
32
     MPI_Win_unlock(B)
33
34
     MPI_Barrier
                                     MPI_Barrier
35
36
                                     MPI_Win_post(B)
37
                                     MPI_Win_start(B)
38
39
                                     load X /* access to private window */
40
                                             /* may return an obsolete value */
41
42
                                     MPI_Win_complete
43
                                     MPI_Win_wait
44
45
     To ensure that the value put by process A is read, the local load must be replaced with a
46
     local MPI_GET operation, or must be placed after the call to MPI_WIN_WAIT.
47
48
```

#### 11.7.1 Atomicity

The outcome of concurrent accumulate operations to the same location with the same predefined datatype is as if the accumulates were done at that location in some serial order. Additional restrictions on the operation apply; see the info key accumulate\_ops in Section 11.2.1. Concurrent accumulate operations with different origin and target pairs are not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to an accumulate operation cannot be accessed by a load or an RMA call other than accumulate until the accumulate operation has completed (at the target). Different interleavings can lead to different results only to the extent that computer arithmetics are not truly associative or commutative. The outcome of accumulate operations with overlapping types of different sizes or target displacements is undefined.

#### 11.7.2 Ordering

Accumulate calls enable element-wise atomic read and write to remote memory locations. MPI specifies ordering between accumulate operations from one process to the same (or overlapping) memory locations at another process on a per-datatype granularity. The default ordering is strict ordering, which guarantees that overlapping updates from the same source to a remote location are committed in program order and that reads (e.g., with MPI\_GET\_ACCUMULATE) and writes (e.g., with MPI\_ACCUMULATE) are executed and committed in program order. Ordering only applies to operations originating at the same origin that access overlapping target memory regions. MPI does not provide any guarantees for accesses or updates from different origin processes to overlapping target memory regions.

26The default strict ordering may incur a significant performance penalty. MPI specifies 27the info key accumulate\_ordering to allow relaxation of the ordering semantics when specified 28 to any window creation function. The values for this key are as follows. If set to none, 29then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA 30 in MPI-2 but is not the default in MPI-3. The key can be set to a comma-separated list 31 of required access orderings at the target. Allowed values in the comma-separated list 32 are rar, war, raw, and waw for read-after-read, write-after-read, read-after-write, and write-33 after-write ordering, respectively. These indicate whether operations of the specified type 34 complete in the order they were issued. For example, raw means that any writes must 35complete at the target before subsequent reads. These ordering requirements apply only to 36 operations issued by the same origin process and targeting the same target process. The 37 default value for accumulate\_ordering is rar,raw,war,waw, which implies that writes complete at 38 the target in the order in which they were issued, reads complete at the target before any 39 writes that are issued after the reads, and writes complete at the target before any reads 40 that are issued after the writes. Any subset of these four orderings can be specified. For 41 example, if only read-after-read and write-after-write ordering is required, then the value 42of the accumulate\_ordering key could be set to rar, waw. The order of values is not significant. 43

Note that the above ordering semantics apply only to accumulate operations, not put and get. Put and get within an epoch are unordered.

45 46

44

- 47
- 48

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14 15

16 17

18

19

20

21

22

23

 $^{24}$ 

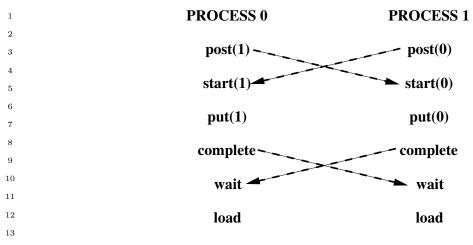


Figure 11.6: Symmetric communication

## 11.7.3 Progress

14

15 16

17

<sup>18</sup> One-sided communication has the same progress requirements as point-to-point communi-<sup>19</sup> cation: once a communication is enabled it is guaranteed to complete. RMA calls must have <sup>20</sup> local semantics, except when required for synchronization with other RMA calls.

21There is some fuzziness in the definition of the time when a RMA communication becomes enabled. This fuzziness provides to the implementor more flexibility than with 22point-to-point communication. Access to a target window becomes enabled once the corre-23sponding synchronization (such as MPI\_WIN\_FENCE or MPI\_WIN\_POST) has executed. On  $^{24}$ 25the origin process, an RMA communication may become enabled as soon as the correspond-26ing put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. Once the communication is enabled both at the origin and at the target, the 27communication must complete. 28

<sup>29</sup> Consider the code fragment in Example 11.4. Some of the calls may block if the target
 <sup>30</sup> window is not posted. However, if the target window is posted, then the code fragment
 <sup>31</sup> must complete. The data transfer may start as soon as the put call occurs, but may be
 <sup>32</sup> delayed until the ensuing complete call occurs.

Consider the code fragment in Example 11.5. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code illustrated in Figure 11.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transferred.

Assume, in the last example, that the order of the post and start calls is reversed at
each process. Then, the code may deadlock, as each process may block on the start call,
waiting for the matching post to occur. Similarly, the program will deadlock if the order of
the complete and wait calls is reversed at each process.

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice versa. Consider the code illustrated in Figure 11.7. This code will deadlock: the wait

### Unofficial Draft for Comment Only

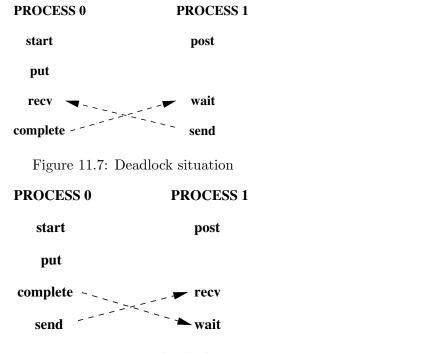


Figure 11.8: No deadlock

of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until process 1 calls send. Consider, on the other hand, the code illustrated in Figure 11.8. This code will not deadlock. Once process 1 calls post, then the sequence start, put, complete on process 0 can proceed to completion. Process 0 will reach the send call, allowing the receive call of process 1 to complete.

*Rationale.* MPI implementations must guarantee that a process makes progress on all enabled communications it participates in, while blocked on an MPI call. This is true for send-receive communication and applies to RMA communication as well. Thus, in the example in Figure 11.8, the put and complete calls of process 0 should complete while process 1 is blocked on the receive call. This may require the involvement of process 1, e.g., to transfer the data put, while it is blocked on the receive call.

A similar issue is whether such progress must occur while a process is busy comput-ing, or blocked in a non-MPI call. Suppose that in the last example the send-receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not spec-ify whether deadlock is avoided. Suppose that the blocking receive of process 1 is replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this am-biguity is unfortunate, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious. (End of rationale.) 

## Unofficial Draft for Comment Only

 $^{24}$ 

## 11.7.4 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (End of advice to users.)

A coherence problem exists between variables kept in registers and the memory values of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI\_WIN\_UNIFIED.

The problem is illustrated by the following code:

12	-		
13	Source of Process 1	Source of Process 2	Executed in Process 2
14	bbbb = 777	buff = 999	reg_A:=999
15	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
16	call MPI_PUT(bbbb		stop appl.thread
17	into buff of process 2)		buff:=777 in PUT handler
18			continue appl.thread
19	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
20		ccc = buff	ccc:=reg_A
21			

In this example, variable **buff** is allocated in the register **reg\_A** and therefore **ccc** will have the old value of **buff** and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 18.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in modules or COMMON blocks. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 18.1.10–18.1.20. Sections 18.1.17 to 18.1.17 discuss several solutions for the problem in this example.

# 11.8 Examples

34 35 36

37

38

39

**Example 11.13** The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array A, which contains the origin and target buffers of the put calls.

```
40
^{41}
     while(!converged(A)){
42
       update(A);
43
       MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
44
       for(i=0; i < toneighbors; i++)</pre>
45
         MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
46
                                 todisp[i], 1, totype[i], win);
47
       MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
48
     }
```

1

 $\mathbf{2}$ 

3

4 5

6

 $\overline{7}$ 

8

9

10

11

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

The same code could be written with get rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

**Example 11.14** Same generic example, with more computation/communication overlap. We assume that the update phase is broken into two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither uses nor provides communicated data, is updated.

The get communication can be concurrent with the core update, since they do not access the same locations, and the local update of the origin buffer by the get call can be concurrent with the local update of the core by the update\_core call. In order to get similar overlap with put communication we would need to use separate windows for the core and for the boundary. This is required because we do not allow local stores to be concurrent with puts on the same, or on overlapping, windows.

**Example 11.15** Same code as in Example 11.13, rewritten using post-start-complete-wait.

**Example 11.16** Same example, with split phases, as in Example 11.14.

```
...
while(!converged(A)){
    update_boundary(A);
    MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
    MPI_Win_start(fromgroup, 0, win);
```

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

```
1
       for(i=0; i < fromneighbors; i++)</pre>
\mathbf{2}
         MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
3
                          fromdisp[i], 1, fromtype[i], win);
4
       update_core(A);
5
       MPI_Win_complete(win);
6
       MPI_Win_wait(win);
7
     }
8
9
     Example 11.17 A checkerboard, or double buffer communication pattern, that allows
10
     more computation/communication overlap. Array A0 is updated using values of array A1,
11
     and vice versa. We assume that communication is symmetric: if process A gets data from
12
     process B, then process B gets data from process A. Window wini consists of array Ai.
13
14
     . . .
15
     if (!converged(A0,A1))
16
       MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
17
     MPI_Barrier(comm0);
18
     /* the barrier is needed because the start call inside the
19
     loop uses the nocheck option */
20
     while(!converged(A0, A1)){
21
       /* communication on AO and computation on A1 */
22
       update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
23
       MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
24
       for(i=0; i < fromneighbors; i++)</pre>
25
         MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
26
                     fromdisp0[i], 1, fromtype0[i], win0);
27
       update1(A1); /* local update of A1 that is
28
                         concurrent with communication that updates A0 */
29
       MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
30
       MPI_Win_complete(win0);
31
       MPI_Win_wait(win0);
32
33
       /* communication on A1 and computation on A0 */
34
       update2(A0, A1); /* local update of A0 that depends on A1 (and A0) */
35
       MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
36
       for(i=0; i < fromneighbors; i++)</pre>
37
         MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
38
                      fromdisp1[i], 1, fromtype1[i], win1);
39
       update1(A0); /* local update of A0 that depends on A0 only,
40
                       concurrent with communication that updates A1 */
41
       if (!converged(A0,A1))
42
         MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
43
       MPI_Win_complete(win1);
44
       MPI_Win_wait(win1);
45
     }
46
         A process posts the local window associated with win0 before it completes RMA accesses
47
```

468

47 A process posts the local window associated with win0 before it completes RMA accesses 48 to the remote windows associated with win1. When the wait(win1) call returns, then all neighbors of the calling process have posted the windows associated with win0. Conversely, when the wait(win0) call returns, then all neighbors of the calling process have posted the windows associated with win1. Therefore, the nocheck option can be used with the calls to MPI\_WIN\_START.

Put calls can be used, instead of get calls, if the area of array AO (resp. A1) used by the update(A1, AO) (resp. update(AO, A1)) call is disjoint from the area modified by the RMA communication. On some systems, a put call may be more efficient than a get call, as it requires information exchange only in one direction.

In the next several examples, for conciseness, the expression

z = MPI\_Get\_accumulate(...)

means to perform an MPI\_GET\_ACCUMULATE with the result buffer (given by result\_addr in the description of MPI\_GET\_ACCUMULATE) on the left side of the assignment, in this case, z. This format is also used with MPI\_COMPARE\_AND\_SWAP.

**Example 11.18** The following example implements a naive, non-scalable counting semaphore. The example demonstrates the use of MPI\_WIN\_SYNC to manipulate the public copy of X, as well as MPI\_WIN\_FLUSH to complete operations without ending the access epoch opened with MPI\_WIN\_LOCK\_ALL. To avoid the rules regarding synchronization of the public and private copies of windows, MPI\_ACCUMULATE and MPI\_GET\_ACCUMULATE are used to write to or read from the local public copy.

		22
Process A:	Process B:	23
MPI_Win_lock_all	MPI_Win_lock_all	24
window location X		25
X=2		26
MPI_Win_sync		27
MPI_Barrier	MPI_Barrier	28
		29
MPI_Accumulate(X, MPI_SUM, -1)	MPI_Accumulate(X, MPI_SUM, -1)	30
		31
stack variable z	stack variable z	32
do	do	33
<pre>z = MPI_Get_accumulate(X,</pre>	<pre>z = MPI_Get_accumulate(X,</pre>	34
MPI_NO_OP, 0)	MPI_NO_OP, 0)	35
MPI_Win_flush(A)	MPI_Win_flush(A)	36
while(z!=0)	while(z!=0)	37
		38
MPI_Win_unlock_all	MPI_Win_unlock_all	39
		40

**Example 11.19** Implementing a critical region between two processes (Peterson's algorithm). Despite their appearance in the following example, MPI\_WIN\_LOCK\_ALL and MPI\_WIN\_UNLOCK\_ALL are not collective calls, but it is frequently useful to start shared access epochs to all processes from all other processes in a window. Once the access epochs are established, accumulate communication operations and flush and sync synchronization operations can be used to read from or write to the public copy of the window.

 $\mathbf{2}$ 

 $\mathbf{5}$ 

```
1
     Process A:
                                              Process B:
\mathbf{2}
                                              window location Y
     window location X
3
     window location T
4
\mathbf{5}
     MPI_Win_lock_all
                                              MPI_Win_lock_all
6
     X=1
                                              Y=1
7
     MPI_Win_sync
                                              MPI_Win_sync
8
     MPI_Barrier
                                              MPI_Barrier
9
     MPI_Accumulate(T, MPI_REPLACE, 1)
                                              MPI_Accumulate(T, MPI_REPLACE, 0)
10
     stack variables t,y
                                               stack variable t,x
11
     t=1
                                              t=0
12
     y=MPI_Get_accumulate(Y,
                                              x=MPI_Get_accumulate(X,
13
        MPI_NO_OP, 0)
                                                  MPI_NO_OP, 0)
14
     while(y==1 && t==1) do
                                              while(x==1 && t==0) do
15
       y=MPI_Get_accumulate(Y,
                                                 x=MPI_Get_accumulate(X,
16
          MPI_NO_OP, 0)
                                                    MPI_NO_OP, 0)
17
       t=MPI_Get_accumulate(T,
                                                 t=MPI_Get_accumulate(T,
18
                                                    MPI_NO_OP, 0)
          MPI_NO_OP, 0)
19
       MPI_Win_flush_all
                                                 MPI_Win_flush(A)
20
     done
                                              done
21
     // critical region
                                              // critical region
22
     MPI_Accumulate(X, MPI_REPLACE, 0)
                                              MPI_Accumulate(Y, MPI_REPLACE, 0)
23
     MPI_Win_unlock_all
                                              MPI_Win_unlock_all
^{24}
```

 Example 11.20 Implementing a critical region between multiple processes with compare and swap. The call to MPI\_WIN\_SYNC is necessary on Process A after local initialization of A to guarantee the public copy has been updated with the initialization value found in the private copy. It would also be valid to call MPI\_ACCUMULATE with MPI\_REPLACE to directly initialize the public copy. A call to MPI\_WIN\_FLUSH would be necessary to assure A in the public copy of Process A had been updated before the barrier.

```
Process A:
                                              Process B...:
32
     MPI_Win_lock_all
                                              MPI_Win_lock_all
33
     atomic location A
34
     A=0
35
     MPI_Win_sync
36
     MPI_Barrier
                                              MPI_Barrier
37
     stack variable r=1
                                               stack variable r=1
38
     while(r != 0) do
                                              while(r != 0) do
39
       r = MPI_Compare_and_swap(A, 0, 1)
                                                r = MPI_Compare_and_swap(A, 0, 1)
40
       MPI_Win_flush(A)
                                                MPI_Win_flush(A)
41
     done
                                               done
42
     // critical region
                                              // critical region
43
     r = MPI_Compare_and_swap(A, 1, 0)
                                              r = MPI_Compare_and_swap(A, 1, 0)
44
     MPI_Win_unlock_all
                                              MPI_Win_unlock_all
45
46
```

Example 11.21 The following example demonstrates the proper synchronization in the
 <sup>48</sup> unified memory model when a data transfer is implemented with load and store in the

#### **Unofficial Draft for Comment Only**

Process A

case of windows in shared memory (instead of MPI\_PUT or MPI\_GET) and the synchronization between processes is performed using point-to-point communication. The synchronization between processes must be supplemented with a memory synchronization through calls to MPI\_WIN\_SYNC, which act locally as a processor-memory barrier. In Fortran, if MPI\_ASYNC\_PROTECTS\_NONBLOCKING is .FALSE. or the variable X is not declared as ASYNCHRONOUS, reordering of the accesses to the variable X must be prevented with MPI\_F\_SYNC\_REG operations. (No equivalent function is needed in C.)

The variable X is contained within a shared memory window and X corresponds to the same memory location at both processes. The MPI\_WIN\_SYNC operation performed by process A ensures completion of the load/store operations issued by process A. The MPI\_WIN\_SYNC operation performed by process B ensures that process A's updates to X are visible to process B.

Process B

		14
		15
MPI_WIN_LOCK_ALL(	MPI_WIN_LOCK_ALL(	16
MPI_MODE_NOCHECK,win)	MPI_MODE_NOCHECK,win)	17
		18
DO	DO	19
X=		20
		21
MPI_F_SYNC_REG(X)		22
MPI_WIN_SYNC(win)		23
MPI_SEND	MPI_RECV	24
	MPI_WIN_SYNC(win)	25
	MPI_F_SYNC_REG(X)	26
		27
	print X	28
		29
	MPI_F_SYNC_REG(X)	30
MPI_RECV	MPI_SEND	31
MPI_F_SYNC_REG(X)		32
END DO	END DO	33
		34
MPI_WIN_UNLOCK_ALL(win)	MPI_WIN_UNLOCK_ALL(win)	35

**Example 11.22** The following example shows how request-based operations can be used to overlap communication with computation. Each process fetches, processes, and writes the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used to allow up to M communication operations to overlap with computation.

int	i, j;
MPI_Win	win;
MPI_Request	<pre>put_req[M] = { MPI_REQUEST_NULL };</pre>
MPI_Request	get_req;
double	*baseptr;
double	data[M][N];

 $\mathbf{2}$ 

 $\overline{7}$ 

```
1
     MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
\mathbf{2}
       MPI_COMM_WORLD, &baseptr, &win);
3
4
     MPI_Win_lock_all(0, win);
5
6
     for (i = 0; i < NSTEPS; i++) {</pre>
7
      if (i<M)
8
         j=i;
9
      else
10
         MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
11
12
      MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
13
                 &get_req);
14
      MPI_Wait(&get_req,MPI_STATUS_IGNORE);
15
      compute(i, data[j], ...);
16
      MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
17
                 &put_req[j]);
18
     }
19
20
     MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
21
     MPI_Win_unlock_all(win);
22
23
     Example 11.23 The following example constructs a distributed shared linked list using
24
     dynamic windows. Initially process 0 creates the head of the list, attaches it to the window,
25
     and broadcasts the pointer to all processes. All processes then concurrently append N new
26
     elements to the list. When a process attempts to attach its element to the tail of the
27
     list it may discover that its tail pointer is stale and it must chase ahead to the new tail
28
     before the element can be attached. This example requires some modification to work in
29
     an environment where the layout of the structures is different on different processes.
30
^{31}
     . . .
32
     #define NUM_ELEMS 10
33
34
     #define LLIST_ELEM_NEXT_RANK ( offsetof(llist_elem_t, next) + \
35
                                         offsetof(llist_ptr_t, rank) )
36
     #define LLIST_ELEM_NEXT_DISP ( offsetof(llist_elem_t, next) + \
37
                                         offsetof(llist_ptr_t, disp) )
38
39
     /* Linked list pointer */
40
     typedef struct {
41
       MPI_Aint disp;
42
       int
                  rank;
43
     } llist_ptr_t;
44
45
     /* Linked list element */
46
     typedef struct {
47
       llist_ptr_t next;
48
```

```
1
  int value;
                                                                                     \mathbf{2}
} llist_elem_t;
                                                                                     3
                                                                                     4
const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM, -1 };
                                                                                     5
                                                                                     6
/* List of locally allocated list elements. */
                                                                                     7
static llist_elem_t **my_elems = NULL;
                                                                                     8
static int my_elems_size = 0;
static int my_elems_count = 0;
                                                                                     9
                                                                                    10
                                                                                    11
/* Allocate a new shared linked list element */
MPI_Aint alloc_elem(int value, MPI_Win win) {
                                                                                    12
                                                                                    13
 MPI_Aint disp;
 llist_elem_t *elem_ptr;
                                                                                    14
                                                                                    15
  /* Allocate the new element and register it with the window */
                                                                                    16
                                                                                    17
 MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
                                                                                    18
  elem_ptr->value = value;
                                                                                    19
  elem_ptr->next = nil;
                                                                                    20
 MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
                                                                                    21
  /* Add the element to the list of local elements so we can free
                                                                                    22
                                                                                    23
     it later. */
                                                                                    24
  if (my_elems_size == my_elems_count) {
                                                                                    25
    my_elems_size += 100;
                                                                                    26
    my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
  }
                                                                                    27
 my_elems[my_elems_count] = elem_ptr;
                                                                                    28
                                                                                    29
 my_elems_count++;
                                                                                    30
                                                                                    31
 MPI_Get_address(elem_ptr, &disp);
                                                                                    32
 return disp;
                                                                                    33
}
                                                                                    34
int main(int argc, char *argv[]) {
                                                                                    35
                                                                                    36
  int
                 procid, nproc, i;
                                                                                    37
 MPI_Win
                llist_win;
                                                                                    38
  llist_ptr_t
                head_ptr, tail_ptr;
                                                                                    39
 MPI_Init(&argc, &argv);
                                                                                    40
                                                                                    41
                                                                                    42
 MPI_Comm_rank(MPI_COMM_WORLD, &procid);
 MPI_Comm_size(MPI_COMM_WORLD, &nproc);
                                                                                    43
                                                                                    44
 MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
                                                                                    45
                                                                                    46
                                                                                    47
  /* Process 0 creates the head node */
                                                                                    48
  if (procid == 0)
```

```
1
         head_ptr.disp = alloc_elem(-1, llist_win);
2
3
       /* Broadcast the head pointer to everyone */
4
       head_ptr.rank = 0;
5
       MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
6
       tail_ptr = head_ptr;
7
8
       /* Lock the window for shared access to all targets */
9
       MPI_Win_lock_all(0, llist_win);
10
11
       /* All processes concurrently append NUM_ELEMS elements to the list */
12
       for (i = 0; i < NUM_ELEMS; i++) {</pre>
13
         llist_ptr_t new_elem_ptr;
14
         int success;
15
16
         /* Create a new list element and attach it to the window */
17
         new_elem_ptr.rank = procid;
18
         new_elem_ptr.disp = alloc_elem(procid, llist_win);
19
20
         /* Append the new node to the list. This might take multiple
21
            attempts if others have already appended and our tail pointer
22
            is stale. */
23
         do {
24
           llist_ptr_t next_tail_ptr = nil;
25
26
           MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
27
                (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
28
               MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
29
               llist_win);
30
31
           MPI_Win_flush(tail_ptr.rank, llist_win);
32
           success = (next_tail_ptr.rank == nil.rank);
33
34
           if (success) {
35
             MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
36
                 MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
37
                 MPI_AINT, MPI_REPLACE, llist_win);
38
39
             MPI_Win_flush(tail_ptr.rank, llist_win);
40
             tail_ptr = new_elem_ptr;
41
42
           } else {
43
             /* Tail pointer is stale, fetch the displacement. May take
44
                multiple tries if it is being updated. */
45
             do {
46
               MPI_Get_accumulate( NULL, 0, MPI_AINT, &next_tail_ptr.disp,
47
                    1, MPI_AINT, tail_ptr.rank,
48
                    MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
```

Unofficial Draft for Comment Only

```
1
               1, MPI_AINT, MPI_NO_OP, llist_win);
                                                                                           \mathbf{2}
                                                                                           3
           MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                           4
        } while (next_tail_ptr.disp == nil.disp);
                                                                                           5
        tail_ptr = next_tail_ptr;
      }
                                                                                           6
                                                                                           7
    } while (!success);
 }
                                                                                           8
                                                                                           9
                                                                                           10
 MPI_Win_unlock_all(llist_win);
 MPI_Barrier( MPI_COMM_WORLD );
                                                                                           11
                                                                                          12
 /* Free all the elements in the list */
                                                                                          13
 for ( ; my_elems_count > 0; my_elems_count--) {
                                                                                          14
    MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
                                                                                          15
                                                                                          16
    MPI_Free_mem(my_elems[my_elems_count-1]);
                                                                                           17
 }
                                                                                           18
 MPI_Win_free(&llist_win);
                                                                                           19
. . .
                                                                                          20
                                                                                          21
                                                                                          22
                                                                                          23
                                                                                          ^{24}
                                                                                          25
                                                                                           26
                                                                                          27
                                                                                          28
                                                                                          29
                                                                                          30
                                                                                          31
                                                                                          32
                                                                                          33
                                                                                          34
                                                                                          35
                                                                                          36
                                                                                          37
                                                                                          38
                                                                                          39
                                                                                           40
                                                                                          ^{41}
                                                                                          42
                                                                                          43
                                                                                          44
                                                                                           45
                                                                                           46
                                                                                           47
                                                                                           48
```

# Chapter 12

# **External Interfaces**

### 12.1 Introduction

This chapter begins with calls used to create **generalized requests**, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. These calls can be used to layer new functionality on top of MPI. Next, Section 12.3 deals with setting the information found in status. This functionality is needed for generalized requests.

The chapter continues, in Section 12.4, with a discussion of how threads are to be handled in MPI. Although thread compliance is not required, the standard specifies how threads are to work if they are provided.

# 12.2 Generalized Requests

The goal of generalized requests is to allow users to define new nonblocking operations. Such an outstanding nonblocking operation is represented by a (generalized) request. A fundamental property of nonblocking operations is that progress toward the completion of this operation occurs asynchronously, i.e., concurrently with normal program execution. Typically, this requires execution of code concurrently with the execution of the user code, e.g., in a separate thread or in a signal handler. Operating systems provide a variety of mechanisms in support of concurrent execution. MPI does not attempt to standardize or to replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as MPI\_WAIT or MPI\_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

*Rationale.* It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

1 2 3 4 5 6 7 8 9 10	the MPI is plication. formed by MPI_GREC tion" state user. A new	implementation, and the For a generalized requer the application; therefor QUEST_COMPLETE whe us of generalized requests w generalized request is st	
11	MPI_GRE	QUEST_START(query_fn,	free_fn, cancel_fn, extra_state, request)
12 13	IN	query_fn	callback function invoked when request status is queried (function)
14 15 16	IN	free_fn	callback function invoked when request is freed (func- tion)
17 18	IN	cancel_fn	callback function invoked when request is cancelled (function)
19	IN	extra_state	extra state
20 21	OUT	request	generalized request (handle)
23 24 25 26	<pre>int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,</pre>		
27 28 29 30 31 32 33 34 35 36 37 38 39	PROCI PROCI PROCI INTE TYPE INTE MPI_GREQU INTE EXTE	ierror) EDURE(MPI_Grequest_que EDURE(MPI_Grequest_can GER(KIND=MPI_ADDRESS_M (MPI_Request), INTENT GER, OPTIONAL, INTENT UEST_START(QUERY_FN, M IERROR) GER REQUEST, IERROR RNAL QUERY_FN, FREE_F	(OUT) :: ierror FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST, N, CANCEL_FN
40 41 42 43 44 45 46 47 48	Adva requ The c The s	ests, in C and Fortran. ( call starts a generalized re syntax and meaning of th	t a generalized request is of the same type as regular

starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined state for the request.	$\frac{1}{2}$
In C, the query function is	3
<pre>typedef int MPI_Grequest_query_function(void *extra_state,</pre>	4
MPI_Status *status);	5
in Fortron with the mni f08 module	6
in Fortran with the mpi_f08 module ABSTRACT INTERFACE	7
SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)	8
TYPE(MPI_Status) :: status	9
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	10
INTEGER :: ierror	11
	12
in Fortran with the mpi module and mpif.h	13
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)	14
INTEGER STATUS(MPI_STATUS_SIZE), IERROR	15
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	16
The query_fn function computes the status that should be returned for the generalized	17 18
request. The status also includes information about successful/unsuccessful cancellation of	18
the request (result to be returned by MPI_TEST_CANCELLED).	20
The query_fn callback is invoked by the MPI_{WAIT TEST}{ANY SOME ALL} call that	20
completed the generalized request associated with this callback. The callback function is	22
also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when	23
the call occurs. In both cases, the callback is passed a reference to the corresponding	24
status variable passed by the user to the MPI call; the status set by the callback function	25
is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or	26
$MPI_STATUSES_IGNORE$ to the $MPI$ function that causes <code>query_fn</code> to be called, then $MPI$	27
will pass a valid status object to query_fn, and this status will be ignored upon return of the	28
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE	29
is called on the request; it may be invoked several times for the same generalized request,	30
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also	31
that a call to MPI_{WAIT TEST}{SOME ALL} may cause multiple invocations of query_fn	32
callback functions, one for each generalized request that is completed by the MPI call. The	33
order of these invocations is not specified by MPI.	34
In C, the free function is	35
<pre>typedef int MPI_Grequest_free_function(void *extra_state);</pre>	36
in Fortran with the mpi_f08 module	37
ABSTRACT INTERFACE	38
SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)	39
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state	40
INTEGER :: ierror	41 42
in Fortran with the mpi module and mpif.h	42
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)	43
INTEGER IERROR	45
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	46
The free for function is involved to clean up user allocated recourses when the merelined	47
The free_fn function is invoked to clean up user-allocated resources when the generalized	48

15

16

17

18

19

20

21

22

23

24

2526

34

1 request is freed.

 $\mathbf{2}$ The free\_fn callback is invoked by the MPI\_{WAIT|TEST}{ANY|SOME|ALL} call that 3 completed the generalized request associated with this callback. free\_fn is invoked after 4 the call to query\_fn for the same request. However, if the MPI call completed multiple  $\mathbf{5}$ generalized requests, the order in which free\_fn callback functions are invoked is not specified 6 by MPI.

7 The free\_fn callback is also invoked for generalized requests that are freed by a call 8 to MPI\_REQUEST\_FREE (no call to MPI\_{WAIT|TEST}{ANY|SOME|ALL} will occur for 9 such a request). In this case, the callback function will be called either in the MPI call 10 MPI\_REQUEST\_FREE(request), or in the MPI call MPI\_GREQUEST\_COMPLETE(request), 11whichever happens last, i.e., in this case the actual freeing code is executed as soon as both 12calls MPI\_REQUEST\_FREE and MPI\_GREQUEST\_COMPLETE have occurred. The request 13is not deallocated until after free\_fn completes. Note that free\_fn will be invoked only once 14per request by a correct program.

Advice to users. Calling MPI\_REQUEST\_FREE(request) will cause the request handle to be set to MPI\_REQUEST\_NULL. This handle to the generalized request is no longer valid. However, user copies of this handle are valid until after free\_fn completes since MPI does not deallocate the object until then. Since free\_fn is not called until after MPI\_GREQUEST\_COMPLETE, the user copy of the handle can be used to make this call. Users should note that MPI will deallocate the object after free\_fn executes. At this point, user copies of the request handle no longer point to a valid request. MPI will not set user copies to MPI\_REQUEST\_NULL in this case, so it is up to the user to avoid accessing this stale handle. This is a special case in which MPI defers deallocating the object until a later time that is known by the user. (End of advice to users.)

In C, the cancel function is

27typedef int MPI\_Grequest\_cancel\_function(void \*extra\_state, int complete); 28

29in Fortran with the mpi\_f08 module

30 ABSTRACT INTERFACE

SUBROUTINE MPI\_Grequest\_cancel\_function(extra\_state, complete, ierror) 3132 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state 33 LOGICAL :: complete

> INTEGER :: ierror

35in Fortran with the mpi module and mpif.h 36

```
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
37
```

INTEGER IERROR

```
38
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

39 LOGICAL COMPLETE 40

41 The cancel\_fn function is invoked to start the cancelation of a generalized request. 42It is called by MPI\_CANCEL(request). MPI passes complete=true to the callback function 43if MPI\_GREQUEST\_COMPLETE was already called on the request, and 44complete=false otherwise.

45

All callback functions return an error code. The code is passed back and dealt with as 46appropriate for the error code by the MPI function that invoked the callback function. For 47example, if error codes are returned then the error code returned by the callback function 48will be returned by the MPI function that invoked the callback function. In the case of

# **Unofficial Draft for Comment Only**

an MPI\_{WAIT|TEST}{ANY} call that invokes both query\_fn and free\_fn, the MPI call will return the error code returned by the last callback, namely free\_fn. If one or more of the requests in a call to MPI\_{WAIT|TEST}{SOME|ALL} failed, then the MPI call will return MPI\_ERR\_IN\_STATUS. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its free\_fn callback function. However, if the MPI function was passed MPI\_STATUSES\_IGNORE, then the individual error codes returned by each callback functions will be lost.

Advice to users. query\_fn must not set the error field of status since query\_fn may be called by MPI\_WAIT or MPI\_TEST, in which case the error field of status should not change. The MPI library knows the "context" in which query\_fn is invoked and can decide correctly when to put the returned error code in the error field of status. (End of advice to users.)

MPI\_GREQUEST\_COMPLETE(request)

```
INOUT request generalized request (handle)
int MPI_Grequest_complete(MPI_Request request)
MPI_Grequest_complete(request, ierror)
   TYPE(MPI_Request), INTENT(IN) :: request
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
```

INTEGER REQUEST, IERROR

The call informs MPI that the operations represented by the generalized request request are complete (see definitions in Section 2.4). A call to MPI\_WAIT(request, status) will return and a call to MPI\_TEST(request, flag, status) will return flag=true only after a call to MPI\_GREQUEST\_COMPLETE has declared that these operations are complete.

MPI imposes no restrictions on the code executed by the callback functions. However, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as MPI\_TEST, MPI\_REQUEST\_FREE, or MPI\_CANCEL still hold. For example, these calls are supposed to be local and nonblocking. Therefore, the callback functions query\_fn, free\_fn, or cancel\_fn should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once MPI\_CANCEL is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired "local" semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

Advice to implementors. A call to MPI\_GREQUEST\_COMPLETE may unblock a blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (*End of advice to implementors.*)

 $^{24}$ 

```
12.2.1 Examples
```

**Example 12.1** This example shows the code for a user-defined reduce operation on an **int** using a binary tree: each non-root node receives two messages, sums them, and sends them up. We assume that no status is returned and that the operation cannot be cancelled.

```
7
     typedef struct {
8
        MPI_Comm comm;
9
        int tag;
10
        int root;
11
        int valin;
12
        int *valout;
13
        MPI_Request request;
14
        } ARGS;
15
16
17
     int myreduce(MPI_Comm comm, int tag, int root,
18
                    int valin, int *valout, MPI_Request *request)
19
     {
20
        ARGS *args;
21
        pthread_t thread;
22
23
        /* start request */
^{24}
        MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);
25
26
        args = (ARGS*)malloc(sizeof(ARGS));
27
        args->comm = comm;
28
        args->tag = tag;
29
        args->root = root;
30
        args->valin = valin;
^{31}
        args->valout = valout;
32
        args->request = *request;
33
34
        /* spawn thread to handle request */
35
        /* The availability of the pthread_create call is system dependent */
36
        pthread_create(&thread, NULL, reduce_thread, args);
37
38
        return MPI_SUCCESS;
39
     }
40
41
     /* thread code */
42
     void* reduce_thread(void *ptr)
43
     {
44
        int lchild, rchild, parent, lval, rval, val;
45
        MPI_Request req[2];
46
        ARGS *args;
47
48
        args = (ARGS*)ptr;
```

1 2 3

4

5

```
2
   /* compute left and right child and parent in tree; set
                                                                                    3
      to MPI_PROC_NULL if does not exist */
   /* code not shown */
                                                                                    4
                                                                                    5
   . . .
                                                                                    6
                                                                                    7
   MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
                                                                                    8
   MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
   MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
                                                                                    9
                                                                                    10
   val = lval + args->valin + rval;
                                                                                   11
   MPI_Send( &val, 1, MPI_INT, parent, args->tag, args->comm );
   if (parent == MPI_PROC_NULL) *(args->valout) = val;
                                                                                   12
   MPI_Grequest_complete((args->request));
                                                                                   13
                                                                                   14
   free(ptr);
                                                                                    15
   return(NULL);
                                                                                    16
}
                                                                                    17
                                                                                   18
int query_fn(void *extra_state, MPI_Status *status)
                                                                                   19
{
   /* always send just one int */
                                                                                   20
                                                                                   21
   MPI_Status_set_elements(status, MPI_INT, 1);
   /* can never cancel so always true */
                                                                                   22
                                                                                   23
   MPI_Status_set_cancelled(status, 0);
                                                                                   24
   /* choose not to return a value for this */
                                                                                   25
   status->MPI_SOURCE = MPI_UNDEFINED;
                                                                                   26
   /* tag has no meaning for this generalized request */
   status->MPI_TAG = MPI_UNDEFINED;
                                                                                   27
                                                                                   28
   /* this generalized request never fails */
                                                                                   29
   return MPI_SUCCESS;
                                                                                   30
}
                                                                                   31
                                                                                   32
                                                                                   33
int free_fn(void *extra_state)
                                                                                   34
{
   /* this generalized request does not need to do any freeing */
                                                                                   35
   /* as a result it never fails here */
                                                                                   36
                                                                                   37
   return MPI_SUCCESS;
}
                                                                                   38
                                                                                   39
                                                                                    40
                                                                                   41
int cancel_fn(void *extra_state, int complete)
                                                                                   42
{
   /* This generalized request does not support cancelling.
                                                                                   43
                                                                                   44
      Abort if not already done. If done then treat as if cancel failed.*/
                                                                                   45
   if (!complete) {
                                                                                   46
     fprintf(stderr,
                                                                                   47
             "Cannot cancel generalized request - aborting program\n");
                                                                                   48
     MPI_Abort(MPI_COMM_WORLD, 99);
```

```
}
return MPI_SUCCESS;
}
```

# 12.3 Associating Information with Status

MPI supports several different types of requests besides those for point-to-point operations. These range from MPI calls for I/O to generalized requests. It is desirable to allow these calls to use the same request mechanism, which allows one to wait or test on different types of requests. However, MPI\_{TEST|WAIT}{ANY|SOME|ALL} returns a status with information about the request. With the generalization of requests, one needs to define what information will be returned in the status object.

Each MPI call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI\_{TEST|WAIT}{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful values for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in the status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, these calls are provided:

2526

```
MPI_STATUS_SET_ELEMENTS(status, datatype, count)
```

```
27
       INOUT
                 status
                                             status with which to associate count (Status)
28
       IN
                 datatype
                                             datatype associated with count (handle)
29
30
       IN
                 count
                                             number of elements to associate with status (integer)
^{31}
32
     int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
33
                     int count)
34
     MPI_Status_set_elements(status, datatype, count, ierror)
35
          TYPE(MPI_Status), INTENT(INOUT) :: status
36
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                 datatype
37
          INTEGER, INTENT(IN) :: count
38
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
41
          INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
42
43
44
45
46
47
48
```

1

 $\mathbf{2}$ 

3

4 5

> 6 7

8

9

10

11

MPI\_STATUS\_SET\_ELEMENTS\_X(status, datatype, count) 1 2 INOUT status with which to associate count (Status) status 3 IN datatype datatype associated with count (handle) 4 IN count number of elements to associate with status (integer) 56 7 int MPI\_Status\_set\_elements\_x(MPI\_Status \*status, MPI\_Datatype datatype, 8 MPI\_Count count) 9 MPI\_Status\_set\_elements\_x(status, datatype, count, ierror) 10 TYPE(MPI\_Status), INTENT(INOUT) :: status 11 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 12INTEGER(KIND = MPI\_COUNT\_KIND), INTENT(IN) :: count 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI\_STATUS\_SET\_ELEMENTS\_X(STATUS, DATATYPE, COUNT, IERROR) 16INTEGER STATUS(MPI\_STATUS\_SIZE), DATATYPE, IERROR 17 INTEGER (KIND=MPI\_COUNT\_KIND) COUNT 18 These functions modify the opaque part of status so that a call to 19 MPI\_GET\_ELEMENTS or MPI\_GET\_ELEMENTS\_X will return count. MPI\_GET\_COUNT 20will return a compatible value. 2122 The number of elements is set instead of the count because the former Rationale. 23 can deal with a nonintegral number of datatypes. (End of rationale.) 2425A subsequent call to MPI\_GET\_COUNT(status, datatype, count), 26MPI\_GET\_ELEMENTS(status, datatype, count), or 27MPI\_GET\_ELEMENTS\_X(status, datatype, count) must use a datatype argument that has 28the same type signature as the datatype argument that was used in the call to 29 MPI\_STATUS\_SET\_ELEMENTS or MPI\_STATUS\_SET\_ELEMENTS\_X. 30 31Rationale. The requirement of matching type signatures for these calls is similar 32 to the restriction that holds when **count** is set by a receive operation: in that case, 33 the calls to MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and MPI\_GET\_ELEMENTS\_X 34 must use a datatype with the same signature as the datatype used in the receive call. 35(End of rationale.) 36 37 38 MPI\_STATUS\_SET\_CANCELLED(status, flag) 39 INOUT status status with which to associate cancel flag (Status) 40 41 IN flag if true indicates request was cancelled (logical) 4243 int MPI\_Status\_set\_cancelled(MPI\_Status \*status, int flag) 44MPI\_Status\_set\_cancelled(status, flag, ierror) 45TYPE(MPI\_Status), INTENT(INOUT) :: status 4647LOGICAL, INTENT(OUT) :: flag 48 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Unofficial Draft for Comment Only

### MPI\_STATUS\_SET\_CANCELLED(STATUS, FLAG, IERROR) INTEGER STATUS(MPI\_STATUS\_SIZE), IERROR LOGICAL FLAG

If flag is set to true then a subsequent call to  $MPI\_TEST\_CANCELLED(status, flag)$  will also return flag = true, otherwise it will return false.

Advice to users. Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI\_GET\_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra\_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI\_RECV, may lead to unpredictable results and is strongly discouraged. (*End of advice to users.*)

15 16 17

18

32

33

39

40

41

42

43

44

# 12.4 MPI and Threads

This section specifies the interaction between MPI calls and threads. The section lists 19minimal requirements for thread compliant MPI implementations and defines functions 20that can be used for initializing the thread environment. MPI may be implemented in 21environments where threads are not supported or perform poorly. Therefore, MPI imple-22 mentations are not required to be thread compliant as defined in this section. Regard-23less of whether or not the MPI implementation is thread compliant, MPI\_INITIALIZED,  $^{24}$ MPI\_FINALIZED, MPI\_QUERY\_THREAD, MPI\_IS\_THREAD\_MAIN, MPI\_GET\_VERSION 25and MPI\_GET\_LIBRARY\_VERSION must always be thread-safe. When a thread is exe-26cuting one of these routines, if another concurrently running thread also makes an MPI call, 27the outcome will be as if the calls executed in some order. 28

This section generally assumes a thread package similar to POSIX threads [39], but the syntax and semantics of thread calls are not specified here — these are beyond the scope of this document.

12.4.1 General

 In a thread-compliant implementation, an MPI process is a process that may be multithreaded. Each thread can issue MPI calls; however, threads are not separately addressable:
 a rank in a send or receive call identifies a process, not a thread. A message sent to a process can be received by any thread in this process.

- *Rationale.* This model corresponds to the POSIX model of interprocess communication: the fact that a process is multi-threaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations 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.*)
- <sup>45</sup> Advice to users. It is the user's responsibility to prevent races when threads within <sup>46</sup> the same application post conflicting communication calls. The user can make sure <sup>47</sup> that two threads in the same process will not issue conflicting communication calls by <sup>48</sup> using distinct communicators at each thread. (*End of advice to users.*)

# Unofficial Draft for Comment Only

1

2

3

4

5

6 7

8

9

10

11

12

13

The two main requirements for a thread-compliant implementation are listed below.

- 1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
- 2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

**Example 12.2** Process 0 consists of two threads. The first thread executes a blocking send call MPI\_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes a blocking receive call MPI\_Recv(buff2, count, type, 0, 0, comm, &status), i.e., the first thread sends a message that is received by the second thread. This communication should always succeed. According to the first requirement, the execution will correspond to some interleaving of the two calls. According to the second requirement, a call can only block the calling thread and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receive call will occur. Once both calls occur, the communication is enabled and both calls will complete. On the other hand, a single-threaded process that posts a send, followed by a matching receive, may deadlock. The progress requirement for multithreaded implementations is stronger, as a blocked call cannot prevent progress in other threads.

Advice to implementors. MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process-global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is released in between. Finer locks can provide more concurrency, at the expense of higher locking overheads. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads. (*End of advice to implementors.*)

## 12.4.2 Clarifications

Initialization and Completion The call to MPI\_FINALIZE should occur on the same thread that initialized MPI. We call this thread the **main thread**. The call should occur only after all process threads have completed their MPI calls, and have no pending communications or I/O operations.

Rationale. This constraint simplifies implementation. (End of rationale.)

Multiple threads completing the same request. A program in which two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent MPI\_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a request can only be completed once. Any combination of wait or test that violates this rule is erroneous. 

### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

Rationale. This restriction is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is possible to post a second wait on the same handle. With threads, an MPI\_WAIT{ANY|SOME|ALL} may be blocked without having nullified its request(s) so it becomes the user's responsibility to avoid using the same request in an MPI\_WAIT on another thread. This constraint also simplifies implementation, as only one thread

will be blocked on any communication or I/O event. (*End of rationale.*)

8 9

1

2

3

4

5

6

7

<sup>10</sup> Probe A receive call that uses source and tag values returned by a preceding call to <sup>11</sup> MPI\_PROBE or MPI\_IPROBE will receive the message matched by the probe call only <sup>12</sup> if there was no other matching receive after the probe and before that receive. In a multi-<sup>13</sup> threaded environment, it is up to the user to enforce this condition using suitable mutual <sup>14</sup> exclusion logic. This can be enforced by making sure that each communicator is used by <sup>15</sup> only one thread on each process. Alternatively, MPI\_MPROBE or MPI\_IMPROBE can be <sup>16</sup> used.

<sup>18</sup> Collective calls Matching of collective calls on a communicator, window, or file handle is
 done according to the order in which the calls are issued at each process. If concurrent
 threads issue such calls on the same communicator, window or file handle, it is up to the
 user to make sure the calls are correctly ordered, using interthread synchronization.

- Advice to users. With three concurrent threads in each MPI process of a communicator comm, it is allowed that thread A in each MPI process calls a collective operation on comm, thread B calls a file operation on an existing filehandle that was formerly opened on comm, and thread C invokes one-sided operations on an existing window handle that was also formerly created on comm. (*End of advice to users.*)
- Rationale. As specified in MPI\_FILE\_OPEN and MPI\_WIN\_CREATE, a file handle
   and a window handle inherit only the group of processes of the underlying communi cator, but not the communicator itself. Accesses to communicators, window handles
   and file handles cannot affect one another. (End of rationale.)
  - Advice to implementors. If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (End of advice to implementors.)
- Exception handlers An exception handler does not necessarily execute in the context of the
   thread that made the exception-raising MPI call; the exception handler may be executed
   by a thread that is distinct from the thread that will return the error code.
  - *Rationale.* The MPI implementation may be multithreaded, so that part of the communication protocol may execute on a thread that is distinct from the thread that made the MPI call. The design allows the exception handler to be executed on the thread where the exception occurred. (*End of rationale.*)
- $45 \\ 46$

28

33

34

35

36 37

41

42

43

- 47
- 48

Interaction with signals and cancellations The outcome is undefined if a thread that executes an MPI call is cancelled (by another thread), or if a thread catches a signal while executing an MPI call. However, a thread of an MPI process may terminate, and may catch signals or be cancelled by another thread when not executing MPI calls.

*Rationale.* Few C library functions are signal safe, and many have cancellation points — points at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancel-safe" or "async-signal-safe"). (*End of rationale.*)

Advice to users. Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to sigwait for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

Advice to implementors. The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (End of advice to implementors.)

### 12.4.3 Initialization

The following function may be used to initialize MPI, and to initialize the MPI thread environment, instead of MPI\_INIT.

### MPI\_INIT\_THREAD(required, provided)

IN	required	desired level of thread support (integer)	28
IIN	required	desired level of thread support (integer)	29
OUT	provided	provided level of thread support (integer)	30
			31
int MPI_I	nit_thread(int *argc, cha	r ***argv, int required, int *provided)	32
MDT To it		· · · · · · · · · · · · · · · · · · ·	33
	thread(required, provided	-	34
	ER, INTENT(IN) :: requir		35
	ER, INTENT(OUT) :: provi		36
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	37
			38
	THREAD(REQUIRED, PROVIDED		39
INTEG	ER REQUIRED, PROVIDED, IE	IRRUR	
			40
			41

Advice to users. In C, the passing of argc and argv is optional, as with MPI\_INIT as discussed in Section 8.7. In C, null pointers may be passed in their place. (End of advice to users.)

This call initializes MPI in the same way that a call to MPI\_INIT would. In addition, it initializes the thread environment. The argument required is used to specify the desired level of thread support. The possible values are listed in increasing order of thread support.

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

 $44 \\ 45$ 

1 **MPI\_THREAD\_SINGLE** Only one thread will execute. 2 **MPI\_THREAD\_FUNNELED** The process may be multi-threaded, but the application must 3 ensure that only the main thread makes MPI calls (for the definition of main thread, 4 see MPI\_IS\_THREAD\_MAIN on page 492). 56 **MPI\_THREAD\_SERIALIZED** The process may be multi-threaded, and multiple threads may  $\overline{7}$ make MPI calls, but only one at a time: MPI calls are not made concurrently from 8 two distinct threads (all MPI calls are "serialized"). 9 10 **MPI\_THREAD\_MULTIPLE** Multiple threads may call MPI, with no restrictions. 11These values are monotonic; i.e., MPI\_THREAD\_SINGLE < MPI\_THREAD\_FUNNELED < 12MPI\_THREAD\_SERIALIZED < MPI\_THREAD\_MULTIPLE. 13 Different processes in MPI\_COMM\_WORLD may require different levels of thread sup-14port. 15The call returns in **provided** information about the actual level of thread support that 16will be provided by MPI. It can be one of the four values listed above. 17The level(s) of thread support that can be provided by MPI\_INIT\_THREAD will depend 18 on the implementation, and may depend on information provided by the user before the 19program started to execute (e.g., with arguments to mpiexec). If possible, the call will 20return provided = required. Failing this, the call will return the least supported level such 21that provided > required (thus providing a stronger level of support than required by the 22user). Finally, if the user requirement cannot be satisfied, then the call will return in 23provided the highest supported level. 24A thread compliant MPI implementation will be able to return provided 25= MPI\_THREAD\_MULTIPLE. Such an implementation may always return provided 26= MPI\_THREAD\_MULTIPLE, irrespective of the value of required. 27An MPI library that is not thread compliant must always return 28provided=MPI\_THREAD\_SINGLE, even if MPI\_INIT\_THREAD is called on a multithreaded 29 process. The library should also return correct values for the MPI calls that can be executed 30 before initialization, even if multiple threads have been spawned.  $^{31}$ 32 Such code is erroneous, but if the MPI initialization is performed by a Rationale. 33 library, the error cannot be detected until MPI\_INIT\_THREAD is called. The require-34 ments in the previous paragraph ensure that the error can be properly detected. (End 35of rationale.) 36 37 A call to MPI\_INIT has the same effect as a call to MPI\_INIT\_THREAD with a required 38= MPI\_THREAD\_SINGLE. 39 Vendors may provide (implementation dependent) means to specify the level(s) of 40thread support available when the MPI program is started, e.g., with arguments to mpiexec.  $^{41}$ This will affect the outcome of calls to MPI\_INIT and MPI\_INIT\_THREAD. Suppose, for 42example, that an MPI program has been started so that only MPI\_THREAD\_MULTIPLE is 43available. Then  $MPI_INIT_THREAD$  will return provided =  $MPI_THREAD_MULTIPLE$ , irre-44spective of the value of required; a call to MPI\_INIT will also initialize the MPI thread support 45level to MPI\_THREAD\_MULTIPLE. Suppose, instead, that an MPI program has been started 46so that all four levels of thread support are available. Then, a call to MPI\_INIT\_THREAD 47will return provided = required; alternatively, a call to MPI\_INIT will initialize the MPI 48thread support level to MPI\_THREAD\_SINGLE.

*Rationale.* Various optimizations are possible when MPI code is executed singlethreaded, or is executed on multiple threads, but not concurrently: mutual exclusion code may be omitted. Furthermore, if only one thread executes, then the MPI library can use library functions that are not thread safe, without risking conflicts with user threads. Also, the model of one communication thread, multiple computation threads fits many applications well, e.g., if the process code is a sequential Fortran/C program with MPI calls that has been parallelized by a compiler for execution on an SMP node, in a cluster of SMPs, then the process computation is multi-threaded, but MPI calls will likely execute on a single thread.

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multi-threaded MPI codes. (*End of rationale.*)

Advice to implementors. If provided is not MPI\_THREAD\_SINGLE then the MPI library should not invoke C or Fortran library calls that are not thread safe, e.g., in an environment where malloc is not thread safe, then malloc should not be used by the MPI library.

Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI\_INIT\_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time.

Note that required need not be the same value on all processes of MPI\_COMM\_WORLD. (*End of advice to implementors.*)

The following function can be used to query the current level of thread support.

MPI\_QUERY\_THREAD(provided)

OUT	provided	provided level of thread support (integer)
int MPI_Q	uery_thread(int *provided	)
INTEG	_thread(provided, ierror) ER, INTENT(OUT) :: provi ER, OPTIONAL, INTENT(OUT)	ded
- •	_THREAD(PROVIDED, IERROR) ER PROVIDED, IERROR	

The call returns in provided the current level of thread support, which will be the value returned in provided by MPI\_INIT\_THREAD, if MPI was initialized by a call to MPI\_INIT\_THREAD().

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

```
1
     MPI_IS_THREAD_MAIN(flag)
2
        OUT
                 flag
                                               true if calling thread is main thread, false otherwise
3
                                               (logical)
4
5
     int MPI_Is_thread_main(int *flag)
6
\overline{7}
     MPI_Is_thread_main(flag, ierror)
8
          LOGICAL, INTENT(OUT) :: flag
9
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                    ierror
10
     MPI_IS_THREAD_MAIN(FLAG, IERROR)
11
          LOGICAL FLAG
12
          INTEGER IERROR
13
14
          This function can be called by a thread to determine if it is the main thread (the thread
15
     that called MPI_INIT or MPI_INIT_THREAD).
16
          All routines listed in this section must be supported by all MPI implementations.
17
18
           Rationale.
                         MPI libraries are required to provide these calls even if they do not
19
           support threads, so that portable code that contains invocations to these functions
20
           can link correctly. MPI_INIT continues to be supported so as to provide compatibility
21
           with current MPI codes. (End of rationale.)
22
23
           Advice to users. It is possible to spawn threads before MPI is initialized, but no MPI
24
           call other than MPI_GET_VERSION, MPI_INITIALIZED, or MPI_FINALIZED should
25
           be executed by these threads, until MPI_INIT_THREAD is invoked by one thread
26
           (which, thereby, becomes the main thread). In particular, it is possible to enter the
27
           MPI execution with a multi-threaded process.
28
           The level of thread support provided is a global property of the MPI process that can
29
           be specified only once, when MPI is initialized on that process (or before). Portable
30
           third party libraries have to be written so as to accommodate any provided level of
31
           thread support. Otherwise, their usage will be restricted to specific level(s) of thread
32
           support. If such a library can run only with specific level(s) of thread support, e.g.,
33
           only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be used to check
34
           whether the user initialized MPI to the correct level of thread support and, if not,
35
           raise an exception. (End of advice to users.)
36
37
38
39
40
41
42
43
44
45
46
47
48
```

# Chapter 13

# I/O

### 13.1 Introduction

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.

The significant optimizations required for efficiency (e.g., grouping [47], collective buffering [7, 15, 48, 52, 58], and disk-directed I/O [43]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

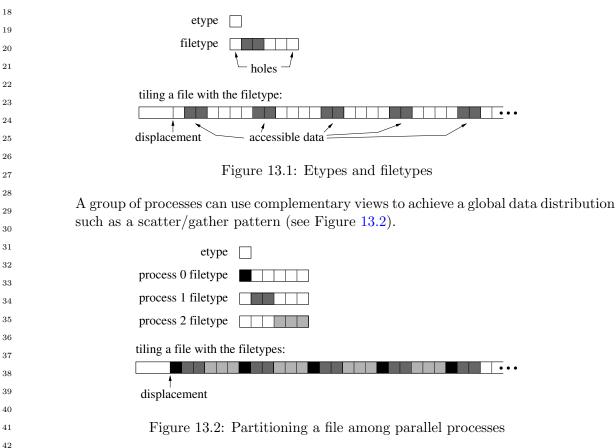
### 13.1.1 Definitions

- file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.
- **displacement** A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement."
- etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

 $^{24}$ 

filetype A *filetype* is the basis for partitioning a file among processes and defines a template
 for accessing the file. A filetype is either a single etype or a derived MPI datatype
 constructed from multiple instances of the same etype. In addition, the extent of any
 hole in the filetype must be a multiple of the etype's extent. The displacements in the
 typemap of the filetype are not required to be distinct, but they must be non-negative
 and monotonically nondecreasing.

view A view defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI\_TYPE\_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 13.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI\_BYTE).



offset An offset is a position in the file relative to the current view, expressed as a count of
 etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0
 is the location of the first etype visible in the view (after skipping the displacement and
 any initial holes in the view). For example, an offset of 2 for process 1 in Figure 13.2 is
 the position of the eighth etype in the file after the displacement. An "explicit offset"
 is an offset that is used as an argument in explicit data access routines.

# Unofficial Draft for Comment Only

- file size and end of file The size of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the end of file is the offset of the first etype accessible in the current view starting after the last byte in the file.
- file pointer A file pointer is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.
- file handle A file handle is an opaque object created by MPI\_FILE\_OPEN and freed by MPI\_FILE\_CLOSE. All operations on an open file reference the file through the file handle.

#### 13.2 File Manipulation

13.2.1 Opening a File

MPI\_FILE\_OPEN(comm, filename, amode, info, fh)

IN	comm	communicator (handle)	21
	film and a		22
IN	filename	name of file to open (string)	23
IN	amode	file access mode (integer)	24
IN	info	info object (handle)	25
	£ŀ-		26
OUT	fh	new file handle (handle)	27

- int MPI\_File\_open(MPI\_Comm comm, const char \*filename, int amode, MPI\_Info info, MPI\_File \*fh)
- MPI\_File\_open(comm, filename, amode, info, fh, ierror) TYPE(MPI\_Comm), INTENT(IN) :: comm CHARACTER(LEN=\*), INTENT(IN) :: filename INTEGER, INTENT(IN) :: amode TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_File), INTENT(OUT) :: fh INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_FILE\_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) CHARACTER\*(\*) FILENAME

INTEGER COMM, AMODE, INFO, FH, IERROR

42MPI\_FILE\_OPEN opens the file identified by the file name filename on all processes in 43 the comm communicator group. MPI\_FILE\_OPEN is a collective routine: all processes must 44provide the same value for **amode**, and all processes must provide filenames that reference the same file. (Values for info may vary.) comm must be an intracommunicator; it is erroneous to pass an intercommunicator to MPI\_FILE\_OPEN. Errors in MPI\_FILE\_OPEN are raised using the default file error handler (see Section 13.7). A process can open a file independently of

### **Unofficial Draft for Comment Only**

1  $\mathbf{2}$ 

3

4

5

6

7

8

9 10

11

12

13 14

1516

17 18 19

20

28

29

30 31

32

33

34

35

36

37

38

39

40

41

45

46

47

other processes by using the MPI\_COMM\_SELF communicator. The file handle returned, fh,
 can be subsequently used to access the file until the file is closed using MPI\_FILE\_CLOSE.
 Before calling MPI\_FINALIZE, the user is required to close (via MPI\_FILE\_CLOSE) all files
 that were opened with MPI\_FILE\_OPEN. Note that the communicator comm is unaffected
 by MPI\_FILE\_OPEN and continues to be usable in all MPI routines (e.g., MPI\_SEND).
 Furthermore, the use of comm will not interfere with I/O behavior.

The format for specifying the file name in the filename argument is implementation
 dependent and must be documented by the implementation.

Advice to implementors. An implementation may require that filename include a string or strings specifying additional information about the file. Examples include the type of filesystem (e.g., a prefix of ufs:), a remote hostname (e.g., a prefix of machine.univ.edu:), or a file password (e.g., a suffix of /PASSWORD=SECRET). (End of advice to implementors.)

Advice to users. On some implementations of MPI, the file namespace may not be identical from all processes of all applications. For example, "/tmp/foo" may denote different files on different processes, or a single file may have many names, dependent on process location. The user is responsible for ensuring that a single file is referenced by the filename argument, as it may be impossible for an implementation to detect this type of namespace error. (*End of advice to users.*)

Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation (no data representation conversion is performed). (POSIX files are linear byte streams in the native representation.) The file view can be changed via the MPI\_FILE\_SET\_VIEW routine.

The following access modes are supported (specified in **amode**, a bit vector OR of the following integer constants):

- MPI\_MODE\_RDONLY read only,
- MPI\_MODE\_RDWR reading and writing,
- MPI\_MODE\_WRONLY write only,
- MPI\_MODE\_CREATE create the file if it does not exist,
- MPI\_MODE\_EXCL error if creating file that already exists,
- MPI\_MODE\_DELETE\_ON\_CLOSE delete file on close,
- MPI\_MODE\_UNIQUE\_OPEN file will not be concurrently opened elsewhere,
  - MPI\_MODE\_SEQUENTIAL file will only be accessed sequentially,
  - MPI\_MODE\_APPEND set initial position of all file pointers to end of file.

Advice to users. C users can use bit vector OR (|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Fortran 77 users can use (nonportably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition.). (End of advice to users.)

# Unofficial Draft for Comment Only

Advice to implementors. The values of these constants must be defined such that the bitwise OR and the sum of any distinct set of these constants is equivalent. (End of advice to implementors.)

The modes MPI\_MODE\_RDONLY, MPI\_MODE\_RDWR, MPI\_MODE\_WRONLY, MPI\_MODE\_CREATE, and MPI\_MODE\_EXCL have identical semantics to their POSIX counterparts [39]. Exactly one of MPI\_MODE\_RDONLY, MPI\_MODE\_RDWR, or MPI\_MODE\_WRONLY, must be specified. It is erroneous to specify MPI\_MODE\_CREATE or MPI\_MODE\_EXCL in conjunction with MPI\_MODE\_RDONLY; it is erroneous to specify MPI\_MODE\_SEQUENTIAL together with MPI\_MODE\_RDWR.

The MPI\_MODE\_DELETE\_ON\_CLOSE mode causes the file to be deleted (equivalent to performing an MPI\_FILE\_DELETE) when the file is closed.

The MPI\_MODE\_UNIQUE\_OPEN mode allows an implementation to optimize access by eliminating the overhead of file locking. It is erroneous to open a file in this mode unless the file will not be concurrently opened elsewhere.

Advice to users. For MPI\_MODE\_UNIQUE\_OPEN, not opened elsewhere includes both inside and outside the MPI environment. In particular, one needs to be aware of potential external events which may open files (e.g., automated backup facilities). When MPI\_MODE\_UNIQUE\_OPEN is specified, the user is responsible for ensuring that no such external events take place. (End of advice to users.)

The MPI\_MODE\_SEQUENTIAL mode allows an implementation to optimize access to some sequential devices (tapes and network streams). It is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Specifying MPI\_MODE\_APPEND only guarantees that all shared and individual file pointers are positioned at the initial end of file when MPI\_FILE\_OPEN returns. Subsequent positioning of file pointers is application dependent. In particular, the implementation does not ensure that all writes are appended.

Errors related to the access mode are raised in the class MPI\_ERR\_AMODE.

The info argument is used to provide information regarding file access patterns and file system specifics (see Section 13.2.8). The constant MPI\_INFO\_NULL can be used when no info needs to be specified.

Advice to users. Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the info argument or facilities outside the scope of MPI. (End of advice to users.)

Files are opened by default using nonatomic mode file consistency semantics (see Section 13.6.1). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI\_FILE\_SET\_ATOMICITY.

13.2.2 Closing a File

MPI_FILE	_CLOSE(fh)
INOUT	fh

file handle (handle)

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{31}$ 

```
498
```

```
1
     int MPI_File_close(MPI_File *fh)
\mathbf{2}
     MPI_File_close(fh, ierror)
3
          TYPE(MPI_File), INTENT(INOUT) :: fh
4
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
     MPI_FILE_CLOSE(FH, IERROR)
7
          INTEGER FH, IERROR
8
          MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
9
      MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
10
      opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
11
      MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.
12
13
           Advice to users. If the file is deleted on close, and there are other processes currently
14
           accessing the file, the status of the file and the behavior of future accesses by these
15
           processes are implementation dependent. (End of advice to users.)
16
17
          The user is responsible for ensuring that all outstanding nonblocking requests and
18
      split collective operations associated with fh made by a process have completed before that
19
      process calls MPI_FILE_CLOSE.
20
          The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
21
      MPI_FILE_NULL.
22
23
     13.2.3 Deleting a File
24
25
26
      MPI_FILE_DELETE(filename, info)
27
       IN
                                              name of file to delete (string)
                  filename
28
29
       IN
                 info
                                              info object (handle)
30
^{31}
      int MPI_File_delete(const char *filename, MPI_Info info)
32
33
     MPI_File_delete(filename, info, ierror)
34
          CHARACTER(LEN=*), INTENT(IN) :: filename
35
          TYPE(MPI_Info), INTENT(IN) :: info
36
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
37
     MPI_FILE_DELETE(FILENAME, INFO, IERROR)
38
          CHARACTER*(*) FILENAME
39
          INTEGER INFO, IERROR
40
41
          MPI_FILE_DELETE deletes the file identified by the file name filename. If the file does
42
      not exist, MPI_FILE_DELETE raises an error in the class MPI_ERR_NO_SUCH_FILE.
43
          The info argument can be used to provide information regarding file system specifics
44
      (see Section 13.2.8). The constant MPI_INFO_NULL refers to the null info, and can be used
45
      when no info needs to be specified.
46
          If a process currently has the file open, the behavior of any access to the file (as well
47
      as the behavior of any outstanding accesses) is implementation dependent. In addition,
48
      whether an open file is deleted or not is also implementation dependent. If the file is not
```

deleted, an error in the class MPI\_ERR\_FILE\_IN\_USE or MPI\_ERR\_ACCESS will be raised. Errors are raised using the default error handler (see Section 13.7).

Resizing a File 13.2.4 MPI\_FILE\_SET\_SIZE(fh, size) INOUT fh file handle (handle) IN size size to truncate or expand file (integer) int MPI\_File\_set\_size(MPI\_File fh, MPI\_Offset size) MPI\_File\_set\_size(fh, size, ierror) TYPE(MPI\_File), INTENT(IN) :: fh INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: size INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_FILE\_SET\_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI\_OFFSET\_KIND) SIZE

MPI\_FILE\_SET\_SIZE resizes the file associated with the file handle fh. size is measured in bytes from the beginning of the file. MPI\_FILE\_SET\_SIZE is collective; all processes in the group must pass identical values for size.

If size is smaller than the current file size, the file is truncated at the position defined by size. The implementation is free to deallocate file blocks located beyond this position.

If size is larger than the current file size, the file size becomes size. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (those locations with displacements between old file size and size) are undefined. It is implementation dependent whether the MPI\_FILE\_SET\_SIZE routine allocates file space — use MPI\_FILE\_PREALLOCATE to force file space to be reserved.

MPI\_FILE\_SET\_SIZE does not affect the individual file pointers or the shared file pointer. If MPI\_MODE\_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call this routine.

Advice to users. It is possible for the file pointers to point beyond the end of file after a MPI\_FILE\_SET\_SIZE operation truncates a file. This is valid, and equivalent to seeking beyond the current end of file. (*End of advice to users.*)

All nonblocking requests and split collective operations on fh must be completed before calling MPI\_FILE\_SET\_SIZE. Otherwise, calling MPI\_FILE\_SET\_SIZE is erroneous. As far as consistency semantics are concerned, MPI\_FILE\_SET\_SIZE is a write operation that conflicts with operations that access bytes at displacements between the old and new file sizes (see Section 13.6.1).

44 45 46

48

1

2

3 4

> 5 6 7

9 10

11 12

13

14

15

16

17 18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34 35

36

37

38 39

40

41

42

```
1
              Preallocating Space for a File
     13.2.5
\mathbf{2}
3
4
     MPI_FILE_PREALLOCATE(fh, size)
5
       INOUT
                 fh
                                               file handle (handle)
6
7
       IN
                 size
                                              size to preallocate file (integer)
8
9
      int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
10
     MPI_File_preallocate(fh, size, ierror)
11
          TYPE(MPI_File), INTENT(IN) ::
                                               fh
12
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::
                                                                size
13
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
14
15
     MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
16
          INTEGER FH, IERROR
17
          INTEGER(KIND=MPI_OFFSET_KIND) SIZE
18
          MPI_FILE_PREALLOCATE ensures that storage space is allocated for the first size bytes
19
      of the file associated with fh. MPI_FILE_PREALLOCATE is collective; all processes in the
20
      group must pass identical values for size. Regions of the file that have previously been
21
      written are unaffected. For newly allocated regions of the file, MPI_FILE_PREALLOCATE
22
      has the same effect as writing undefined data. If size is larger than the current file size, the
23
      file size increases to size. If size is less than or equal to the current file size, the file size is
24
      unchanged.
25
          The treatment of file pointers, pending nonblocking accesses, and file consistency is the
26
      same as with MPI_FILE_SET_SIZE. If MPI_MODE_SEQUENTIAL mode was specified when
27
      the file was opened, it is erroneous to call this routine.
28
29
           Advice to users. In some implementations, file preallocation may be expensive. (End
30
           of advice to users.)
^{31}
32
      13.2.6 Querying the Size of a File
33
34
35
      MPI_FILE_GET_SIZE(fh, size)
36
37
       IN
                  fh
                                               file handle (handle)
38
       OUT
                 size
                                               size of the file in bytes (integer)
39
40
      int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
41
42
     MPI_File_get_size(fh, size, ierror)
43
          TYPE(MPI_File), INTENT(IN) :: fh
44
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) ::
                                                                  size
45
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
46
47
     MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
          INTEGER FH, IERROR
48
```

Unofficial Draft for Comment Only

1 INTEGER(KIND=MPI\_OFFSET\_KIND) SIZE  $\mathbf{2}$ MPI\_FILE\_GET\_SIZE returns, in size, the current size in bytes of the file associated with 3 the file handle fh. As far as consistency semantics are concerned, MPI\_FILE\_GET\_SIZE is a 4 data access operation (see Section 13.6.1). 56 13.2.7 Querying File Parameters 7 8 9 MPI\_FILE\_GET\_GROUP(fh, group) 10 11 IN fh file handle (handle) 12OUT group group which opened the file (handle) 13 14int MPI\_File\_get\_group(MPI\_File fh, MPI\_Group \*group) 1516MPI\_File\_get\_group(fh, group, ierror) 17 TYPE(MPI\_File), INTENT(IN) :: fh 18 TYPE(MPI\_Group), INTENT(OUT) :: group 19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20MPI\_FILE\_GET\_GROUP(FH, GROUP, IERROR) 21INTEGER FH, GROUP, IERROR 22 23MPI\_FILE\_GET\_GROUP returns a duplicate of the group of the communicator used to  $^{24}$ open the file associated with fh. The group is returned in group. The user is responsible for 25freeing group. 2627MPI\_FILE\_GET\_AMODE(fh, amode) 2829 IN fh file handle (handle) 30 OUT amode file access mode used to open the file (integer) 3132 33 int MPI\_File\_get\_amode(MPI\_File fh, int \*amode) 34 MPI\_File\_get\_amode(fh, amode, ierror) 35 TYPE(MPI\_File), INTENT(IN) :: fh 36 INTEGER, INTENT(OUT) :: amode 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI\_FILE\_GET\_AMODE(FH, AMODE, IERROR) INTEGER FH, AMODE, IERROR 40 41 MPI\_FILE\_GET\_AMODE returns, in amode, the access mode of the file associated with 42fh. 43 44**Example 13.1** In Fortran 77, decoding an **amode** bit vector will require a routine such as 45the following: 46

```
1
            SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
\mathbf{2}
     !
3
     !
          TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
4
     !
          IF SET, RETURN 1 IN BIT_FOUND, O OTHERWISE
5
     Ţ
6
            INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
7
            BIT_FOUND = 0
8
            CP_AMODE = AMODE
9
       100 CONTINUE
10
            LBIT = 0
11
            HIFOUND = 0
12
            DO 20 L = MAX_BIT, 0, -1
13
               MATCHER = 2**L
14
               IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
15
                   HIFOUND = 1
16
                   LBIT = MATCHER
17
                   CP_AMODE = CP_AMODE - MATCHER
18
               END IF
19
       20
           CONTINUE
20
            IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
21
            IF (BIT_FOUND .EQ. O .AND. HIFOUND .EQ. 1 .AND. &
22
                 CP_AMODE .GT. 0) GO TO 100
23
            END
24
          This routine could be called successively to decode amode, one bit at a time. For
25
     example, the following code fragment would check for MPI_MODE_RDONLY.
26
27
            CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
28
            IF (BIT_FOUND .EQ. 1) THEN
29
               PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
30
            ELSE
31
               PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
32
            END IF
33
34
     13.2.8 File Info
35
36
     Hints specified via info (see Chapter 9) allow a user to provide information such as file
37
     access patterns and file system specifics to direct optimization. Providing hints may enable
38
     an implementation to deliver increased I/O performance or minimize the use of system
39
     resources. However, hints do not change the semantics of any of the I/O interfaces. In other
40
     words, an implementation is free to ignore all hints. Hints are specified on a per file basis, in
41
     MPI_FILE_OPEN, MPI_FILE_DELETE, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO,
42
     via the opaque info object. When an info object that specifies a subset of valid hints is passed
43
     to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will be no effect on previously set
44
     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

### Unofficial Draft for Comment Only

mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

```
MPI_FILE_SET_INFO(fh, info)
          fh
 INOUT
                                     file handle (handle)
 IN
           info
                                     info object (handle)
int MPI_File_set_info(MPI_File fh, MPI_Info info)
MPI_File_set_info(fh, info, ierror)
    TYPE(MPI_File), INTENT(IN) ::
                                     fh
    TYPE(MPI_Info), INTENT(IN) ::
                                     info
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                         ierror
MPI_FILE_SET_INFO(FH, INFO, IERROR)
    INTEGER FH, INFO, IERROR
```

MPI\_FILE\_SET\_INFO sets new values for the hints of the file associated with fh. MPI\_FILE\_SET\_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Advice to users. Many info items that an implementation can use when it creates or opens a file cannot easily be changed once the file has been created or opened. Thus, an implementation may ignore hints issued in this call that it would have accepted in an open call. (End of advice to users.)

MPI\_FILE\_GET\_INFO(fh, info\_used)

IN	fh	file handle (handle)	32
OUT	info_used	new info object (handle)	33
			34
int MDT F	ile_get_info(MPI_File fh,	MDI Info tinfo used)	35
IIIC MFI_P	iie_get_iiio(mri_riie iii,	MF1_1110 #1110_dsed)	36
MPI_File_	get_info(fh, info_used, i	error)	37
TYPE(	<pre>MPI_File), INTENT(IN) ::</pre>	fh	38
TYPE(	<pre>MPI_Info), INTENT(OUT) ::</pre>	info_used	39
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	40
MDT ETTE	CET INFO/FUL INFO LIGED I		41
	GET_INFO(FH, INFO_USED, I	ERRUR)	42
INTEG	ER FH, INFO_USED, IERROR		43
MPI F	ILE GET INFO returns a new	info object containing the hints of the file associ-	44

MPI\_FILE\_GET\_INFO returns a new info object containing the hints of the file associated with fh. The current setting of all hints actually used by the system related to this open file is returned in info\_used. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pairs. The user is responsible for freeing info\_used via MPI\_INFO\_FREE. 48

### 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

Advice to users. The info object returned in info\_used will contain all hints currently active for this file. This set of hints may be greater or smaller than the set of hints passed in to MPI\_FILE\_OPEN, MPI\_FILE\_SET\_VIEW, or MPI\_FILE\_SET\_INFO, as the system may not recognize some hints set by the user, and may recognize other hints that the user has not set. (*End of advice to users.*)

# Reserved File Hints

Some potentially useful hints (info key values) are outlined below. The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described. (For more details on "info," see Chapter 9.)

These hints mainly affect access patterns and the layout of data on parallel I/O devices. For each hint name introduced, we describe the purpose of the hint, and the type of the hint value. The "[**SAME**]" annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., file\_perm is only useful during file creation).

access\_style (comma separated list of strings): This hint specifies the manner in which
 the file will be accessed until the file is closed or until the access\_style key value is
 altered. The hint value is a comma separated list of the following: read\_once, write\_once,
 read\_mostly, write\_mostly, sequential, reverse\_sequential, and random.

<sup>24</sup> collective\_buffering (boolean) [SAME]: This hint specifies whether the application may
 <sup>25</sup> benefit from collective buffering. Collective buffering is an optimization performed
 <sup>26</sup> on collective accesses. Accesses to the file are performed on behalf of all processes in
 <sup>27</sup> the group by a number of target nodes. These target nodes coalesce small requests
 <sup>28</sup> into large disk accesses. Valid values for this key are true and false. Collective buffering
 <sup>29</sup> parameters are further directed via additional hints: cb\_block\_size, cb\_buffer\_size, and
 <sup>30</sup> cb\_nodes.

- cb\_block\_size (integer) [SAME]: This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (cyclic) pattern.
- <sup>35</sup> cb\_buffer\_size (integer) [SAME]: This hint specifies the total buffer space that can be used
   <sup>36</sup> for collective buffering on each target node, usually a multiple of cb\_block\_size.
  - cb\_nodes (integer) [SAME]: This hint specifies the number of target nodes to be used for collective buffering.
- chunked (comma separated list of integers) [SAME]: This hint specifies that the file
   consists of a multidimentional array that is often accessed by subarrays. The value
   for this hint is a comma separated list of array dimensions, starting from the most
   significant one (for an array stored in row-major order, as in C, the most significant
   dimension is the first one; for an array stored in column-major order, as in Fortran, the
   most significant dimension is the last one, and array dimensions should be reversed).
- <sup>47</sup> chunked\_item (comma separated list of integers) [SAME]: This hint specifies the size
   <sup>48</sup> of each array entry, in bytes.

1

 $\mathbf{2}$ 

3

4

5

6 7

 $^{31}$ 

32

33

34

38

- chunked\_size (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.
- filename (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI\_FILE\_GET\_INFO. This key is ignored when passed to MPI\_FILE\_OPEN, MPI\_FILE\_SET\_VIEW, MPI\_FILE\_SET\_INFO, and MPI\_FILE\_DELETE.
- file\_perm (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI\_FILE\_OPEN with an amode that includes MPI\_MODE\_CREATE. The set of valid values for this key is implementation dependent.
- io\_node\_list (comma separated list of strings) [SAME]: This hint specifies the list of I/O devices that should be used to store the file. This hint is most relevant when the file is created.
- nb\_proc (integer) [SAME]: This hint specifies the number of parallel processes that will typically be assigned to run programs that access this file. This hint is most relevant when the file is created.
- num\_io\_nodes (integer) [SAME]: This hint specifies the number of I/O devices in the system. This hint is most relevant when the file is created.
- striping\_factor (integer) [SAME]: This hint specifies the number of I/O devices that the file should be striped across, and is relevant only when the file is created.
- striping\_unit (integer) [SAME]: This hint specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

# 13.3 File Views

			36
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)		37	
INOUT	fh	file handle (handle)	38
IN	disp	displacement (integer)	39
	•		40
IN	etype	elementary datatype (handle)	41
IN	filetype	filetype (handle)	42
IN	datarep	data representation (string)	43
			44
IN	info	info object (handle)	45
			46

### 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

47

1	MDI File act view(fh dian styre filetyre deteron info isrner)
2	<pre>MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)         TYPE(MPI_File), INTENT(IN) :: fh</pre>
3	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
4	TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
5	CHARACTER(LEN=*), INTENT(IN) :: datarep
6	TYPE(MPI_Info), INTENT(IN) :: info
7	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8	INTEGER, OFITONAL, INTENT(OUT) TEITOT
9	MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
10	INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
11	CHARACTER*(*) DATAREP
12	INTEGER(KIND=MPI_OFFSET_KIND) DISP
13	The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file.
14	The start of the view is set to disp; the type of data is set to etype; the distribution of data
15	to processes is set to filetype; and the representation of data in the file is set to datarep.
16	In addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file
17	pointer to zero. MPI_FILE_SET_VIEW is collective; the values for datarep and the extents
18	of etype in the file data representation must be identical on all processes in the group; values
19	for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be
20	committed.
21	The etype always specifies the data layout in the file. If etype is a portable datatype (see
22	Section 2.4), the extent of etype is computed by scaling any displacements in the datatype
23	to match the file data representation. If <b>etype</b> is not a portable datatype, no scaling is done
24	when computing the extent of etype. The user must be careful when using nonportable
25	etypes in heterogeneous environments; see Section 13.5.1 for further details.
26	If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special
27	displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displace-
28	ment to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is
29	invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.
30	
31	Rationale. For some sequential files, such as those corresponding to magnetic tapes
32	or streaming network connections, the <i>displacement</i> may not be meaningful.
33	MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files.
34	(End of rationale.)
35	
36	Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will
37	immediately follow $MPI_FILE_OPEN$ in numerous instances. A high-quality imple-
38	mentation will ensure that this behavior is efficient. (End of advice to implementors.)
39	
40	The disp displacement argument specifies the position (absolute offset in bytes from
41	the beginning of the file) where the view begins.
42	Advise to second diamonal here and to align here down on where the file in the day of a second
43	Advice to users. disp can be used to skip headers or when the file includes a sequence
44	of data segments that are to be accessed in different patterns (see Figure 13.3). Sep-
45	arate views, each using a different displacement and filetype, can be used to access
46	each segment.
47	(End of advice to users.)
48	

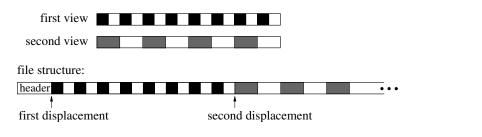


Figure 13.3: Displacements

An etype (elementary datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed by using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes.

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the etype (see Section 13.5). (End of advice to users.)

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the etype nor the filetype is permitted to contain overlapping regions. This restriction is equivalent to the "datatype used in a receive cannot specify overlapping regions" restriction for communication. Note that filetypes from different processes may still overlap each other.

If a filetype has holes in it, then the data in the holes is inaccessible to the calling process. However, the disp, etype, and filetype arguments can be changed via future calls to MPI\_FILE\_SET\_VIEW to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the etype and filetype.

The info argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 13.2.8). The constant MPI\_INFO\_NULL refers to the null info and can be used when no info needs to be specified.

The datarep argument is a string that specifies the representation of data in the file. See the file interoperability section (Section 13.5) for details and a discussion of valid values.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI\_FILE\_SET\_VIEW — otherwise, the call to MPI\_FILE\_SET\_VIEW is erroneous.

Z

 $^{24}$ 

 $^{31}$ 

```
1
     MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)
2
       IN
                fh
                                            file handle (handle)
3
       OUT
                disp
                                            displacement (integer)
4
5
       OUT
                                            elementary datatype (handle)
                etype
6
       OUT
                filetype
                                            filetype (handle)
7
       OUT
                datarep
                                            data representation (string)
8
9
     int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
10
                    MPI_Datatype *filetype, char *datarep)
11
12
     MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) ::
                                                             disp
15
         TYPE(MPI_Datatype), INTENT(OUT) ::
                                                 etype, filetype
16
         CHARACTER(LEN=*), INTENT(OUT) :: datarep
17
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
18
19
     MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
          INTEGER FH, ETYPE, FILETYPE, IERROR
20
21
         CHARACTER*(*) DATAREP
22
         INTEGER(KIND=MPI_OFFSET_KIND) DISP
23
```

<sup>24</sup> MPI\_FILE\_GET\_VIEW returns the process's view of the data in the file. The current <sup>25</sup> value of the displacement is returned in disp. The etype and filetype are new datatypes with <sup>26</sup> typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI\_MAX\_DATAREP\_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI\_FILE\_GET\_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

33 34 35

36

37

27

28

29

30

 $^{31}$ 

32

# 13.4 Data Access

## 13.4.1 Data Access Routines

<sup>38</sup> Data is moved between files and processes by issuing read and write calls. There are <sup>40</sup> three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), <sup>41</sup> synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective <sup>42</sup> *vs.* collective). The following combinations of these data access routines, including two <sup>43</sup> types of file pointers (individual and shared) are provided in Table 13.1.

POSIX read()/fread() and write()/fwrite() are blocking, noncollective operations and use individual file pointers. The MPI equivalents are MPI\_FILE\_READ and MPI\_FILE\_WRITE.

Implementations of data access routines may buffer data to improve performance. This does not affect reads, as the data is always available in the user's buffer after a read operation

positioning	synchronism	cod	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
		MPI_FILE_IWRITE_SHARED	
	split collective	N/A	MPI_FILE_READ_ORDERED_BEGIN
			MPI_FILE_READ_ORDERED_END
			MPI_FILE_WRITE_ORDERED_BEGIN
			MPI_FILE_WRITE_ORDERED_END

Table 13.1: Data access routines

completes. For writes, however, the MPI\_FILE\_SYNC routine provides the only guarantee that data has been transferred to the storage device.

#### Positioning

MPI provides three types of positioning for data access routines: **explicit offsets**, **individual file pointers**, and **shared file pointers**. The different positioning methods may be mixed within the same program and do not affect each other.

The data access routines that accept explicit offsets contain \_AT in their name (e.g., MPI\_FILE\_WRITE\_AT). Explicit offset operations perform data access at the file position given directly as an argument — no file pointer is used nor updated. Note that this is not equivalent to an atomic seek-and-read or seek-and-write operation, as no "seek" is issued. Operations with explicit offsets are described in Section 13.4.2.

The names of the individual file pointer routines contain no positional qualifier (e.g., MPI\_FILE\_WRITE). Operations with individual file pointers are described in Section 13.4.3. The data access routines that use shared file pointers contain \_SHARED or \_ORDERED in their name (e.g., MPI\_FILE\_WRITE\_SHARED). Operations with shared file pointers are described in Section 13.4.4.

The main semantic issues with MPI-maintained file pointers are how and when they are updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to the next data item after the last one that is accessed by the operation. In a nonblocking or split collective operation, the pointer is updated by the call that initiates the I/O, possibly before the access completes.

More formally,

$$new_file_offset = old_file_offset + rac{elements(datatype)}{elements(etype)} imes count$$

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.

#### **Unofficial Draft for Comment Only**

 $^{31}$ 

1	Synchronism
2 3 4	<ul><li>MPI supports blocking and nonblocking I/O routines.</li><li>A blocking I/O call will not return until the I/O request is completed.</li><li>A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete.</li></ul>
5 6 7 8 9	Given suitable hardware, this allows the transfer of data out of and into the user's buffer to proceed concurrently with computation. A separate <i>request complete</i> 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
11 12 13	for immediate. It is erroneous to access the local buffer of a nonblocking data access operation, or to use that buffer as the source or target of other communications, between the initiation and
14 15 16	completion of the operation. The split collective routines support a restricted form of "nonblocking" operations for collective data access (see Section 13.4.5).
17 18	Coordination
19 20 21 22 23 24 25 26 27 28 29	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 pro- cess. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 13.6.4 for rules on semantics of collective calls. Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.
30 31 32 33 34 35 36 37 38 39 40	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.
41 42 43 44 45	A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 and Section 4.1.11. The data is accessed from those parts of the file specified by the current

view (Section 13.3). The type signature of datatype must match the type signature of some
 number of contiguous copies of the etype of the current view. As in a receive, it is erroneous

to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI\_TEST, MPI\_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

To prevent problems with the argument copying and register Advice to users. optimization done by Fortran compilers, please note the hints in Sections 18.1.10-18.1.20. (End of advice to users.)

For blocking routines, status is returned directly. For nonblocking routines and split 12collective 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 14from status by using MPI\_GET\_COUNT and MPI\_GET\_ELEMENTS (or MPI\_GET\_ELEMENTS\_X), respectively. The interpretation of the MPI\_ERROR field is the same as for other operations — normally undefined, but meaningful if an MPI routine returns MPI\_ERR\_IN\_STATUS. The user can pass (in C and Fortran) MPI\_STATUS\_IGNORE in the status argument if the return value of this argument is not needed. The status can be 19passed to MPI\_TEST\_CANCELLED to determine if the operation was cancelled. All other 20fields of status are undefined. 21

When reading, a program can detect the end of file by noting that the amount of data read is less than the amount requested. Writing past the end of file increases the file size. The amount of data accessed will be the amount requested, unless an error is raised (or a read reaches the end of file).

#### Data Access with Explicit Offsets 13.4.2

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)

			33
IN	fh	file handle (handle)	34
IN	offset	file offset (integer)	35
OUT	buf	initial address of buffer (choice)	36
IN	count	number of elements in buffer (integer)	37 38
IN	datatype	datatype of each buffer element (handle)	39
OUT	status	status object (Status)	40
		• • • •	41

int MPI\_File\_read\_at(MPI\_File fh, MPI\_Offset offset, void \*buf, int count, MPI\_Datatype datatype, MPI\_Status \*status)

MPI\_File\_read\_at(fh, offset, buf, count, datatype, status, ierror) 45TYPE(MPI\_File), INTENT(IN) :: fh 46INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: offset 47TYPE(\*), DIMENSION(..) :: buf 48

#### **Unofficial Draft for Comment Only**

1

2

3

4

56

 $\overline{7}$ 

8

9

10 11

13

15

16

17

18

22

23

 $^{24}$ 

2526

2728

29

30 3132

42

43

```
1
         INTEGER, INTENT(IN) :: count
\mathbf{2}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Status) :: status
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
6
         <type> BUF(*)
7
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
8
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
9
10
         MPI_FILE_READ_AT reads a file beginning at the position specified by offset.
11
12
     MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status)
13
14
                 fh
       IN
                                            file handle (handle)
15
       IN
                offset
                                            file offset (integer)
16
       OUT
                 buf
                                            initial address of buffer (choice)
17
18
       IN
                count
                                            number of elements in buffer (integer)
19
       IN
                datatype
                                            datatype of each buffer element (handle)
20
       OUT
                status
                                            status object (Status)
21
22
23
     int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,
^{24}
                    int count, MPI_Datatype datatype, MPI_Status *status)
25
     MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
26
         TYPE(MPI_File), INTENT(IN) :: fh
27
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
28
         TYPE(*), DIMENSION(..) :: buf
29
         INTEGER, INTENT(IN) :: count
30
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
^{31}
         TYPE(MPI_Status) :: status
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
35
          <type> BUF(*)
36
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
37
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
38
         MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT
39
     interface.
40
41
42
43
44
45
46
47
48
```

MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)				
INOUT	fh	file handle (handle)	2	
IN	offset	file offset (integer)	3 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	status	status object (Status)	9	
001	status	Startas (Startas)	10	
int MPI_F	Sile_write_at(MPI_File fh	, MPI_Offset offset, const void *buf,	11	
	int count, MPI_Datat	ype datatype, MPI_Status *status)	12	
MPI_File_	write_at(fh, offset, buf	, count, datatype, status, ierror)	13 14	
	MPI_File), INTENT(IN) ::		15	
	ER(KIND=MPI_OFFSET_KIND)		16	
	(*), DIMENSION(), INTEN	T(IN) :: buf	17	
	ER, INTENT(IN) :: count [MPI_Datatype), INTENT(IN]	) ·· datatype	18	
	(MPI_Status) :: status	, dababype	19 20	
	ER, OPTIONAL, INTENT(OUT)	) :: ierror	20	
MPT FTLE	WRITE AT(FH. OFFSET. BUF	, COUNT, DATATYPE, STATUS, IERROR)	22	
	<pre>BUF(*)</pre>	,,,	23	
INTEG	ER FH, COUNT, DATATYPE, S	STATUS(MPI_STATUS_SIZE), IERROR	24	
INTEG	ER(KIND=MPI_OFFSET_KIND)	OFFSET	25 26	
MPI_F	FILE_WRITE_AT writes a file	beginning at the position specified by offset.	27	
MPI_FILE	_WRITE_AT_ALL(fh, offset, bu	uf, count, datatype, status)	29 30	
INOUT	fh	file handle (handle)	31	
IN	offset	file offset (integer)	32	
IN	buf	initial address of buffer (choice)	33	
IN	count	number of elements in buffer (integer)	34	
IN	datatype	datatype of each buffer element (handle)	35 36	
OUT		·-	37	
001	status	status object (Status)	38	
int MPI F	Tile write at all(MPI File	e fh, MPI_Offset offset, const void *buf,	39	
		ype datatype, MPI_Status *status)	40 41	
MPI_File_	write_at_all(fh, offset,	buf, count, datatype, status, ierror)	42	
	<pre>MPI_File), INTENT(IN) ::</pre>		43	
	ER(KIND=MPI_OFFSET_KIND)		44	
	(*), DIMENSION(), INTEN ER, INTENT(IN) :: count	T(IN) :: buf	45 46	
	[MPI_Datatype), INTENT(IN)	) :: datatvpe	47	
	[MPI_Status) :: status		48	

1	INTE	GER, OPTIONAL, INTENT(O	UT) :: ierror		
2 3	MPI_FILE	MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)			
4	<type> BUF(*)</type>				
5			, STATUS(MPI_STATUS_SIZE), IERROR		
6	INTE	GER(KIND=MPI_OFFSET_KIN	D) UFFSET		
7			collective version of the blocking		
8 9	MPI_FILE	_WRITE_AT interface.			
10					
11	MPI_FILE	_IREAD_AT(fh, offset, buf,	count, datatype, request)		
12	IN	fh	file handle (handle)		
13 14	IN	offset	file offset (integer)		
15	OUT	buf	initial address of buffer (choice)		
16	IN	count	number of elements in buffer (integer)		
17	IN	datatype	datatype of each buffer element (handle)		
18 19	OUT	request	request object (handle)		
20					
21	int MPI_		<pre>fh, MPI_Offset offset, void *buf, int count,</pre>		
22		MPI_Datatype datat	type, MPI_Request *request)		
23 24			uf, count, datatype, request, ierror)		
25	TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset				
26	TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf				
27	INTEGER, INTENT(IN) :: count				
28 29		(MPI_Datatype), INTENT(	v.		
30		(MPI_Request), INTENT(O GER, OPTIONAL, INTENT(O	-		
31					
32			UF, COUNT, DATATYPE, REQUEST, IERROR)		
33 34	01	e> BUF(*) GER FH, COUNT, DATATYPE	. REQUEST. TERROR		
35		GER(KIND=MPI_OFFSET_KIN			
36	MPI	FILE IREAD AT is a nonble	ocking version of the MPI_FILE_READ_AT interface.		
37					
$\frac{38}{39}$		IDEAD AT ALL (the offect	buf, count, datatype, request)		
40		Υ.			
41	IN	fh 	file handle (handle)		
42	IN	offset	file offset (integer)		
43 44	OUT	buf	initial address of buffer (choice)		
44 45	IN	count	number of elements in buffer (integer)		
46	IN	datatype	datatype of each buffer element (handle)		
47	OUT	request	request object (handle)		
48					

int MPI_F		e fh, MPI_Offset offset, void *buf, ype datatype, MPI_Request *request)	1 2	
<pre>MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)    TYPE(MPI_File), INTENT(IN) :: fh    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset    TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf    INTEGER, INTENT(IN) :: count    TYPE(MPI_Datatype), INTENT(IN) :: datatype    TYPE(MPI_Request), INTENT(OUT) :: request    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
<type INTEG INTEG</type 	<pre>MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)     <type> BUF(*)     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET     MPI_FILE_IREAD_AT_ALL is a nonblocking version of MPI_FILE_READ_AT_ALL. See</type></pre>			
Section 13.	.6.5 for semantics of nonblock: .IWRITE_AT(fh, offset, buf, co	ing collective file operations.	17 18 19 20	
INOUT	fh	,	21	
		file handle (handle)	22	
IN	offset	file offset (integer)	23 24	
IN	buf	initial address of buffer (choice)	24 25	
IN	count	number of elements in buffer (integer)	26	
IN	datatype	datatype of each buffer element (handle)	27	
OUT	request	request object (handle)	28	
001	lequeet	request object (narrier)	29	
int MPI_F		n, MPI_Offset offset, const void *buf, ype datatype, MPI_Request *request)	30 31 32	
MPI_File_	iwrite_at(fh, offset, but	f, count, datatype, request, ierror)	33	
TYPE(	<pre>MPI_File), INTENT(IN) ::</pre>	fh	34	
	ER(KIND=MPI_OFFSET_KIND)		35	
		C(IN), ASYNCHRONOUS :: buf	$\frac{36}{37}$	
	ER, INTENT(IN) :: count	) datatuma	38	
	<pre>MPI_Datatype), INTENT(IN) MPI_Request), INTENT(OUT)</pre>	v1	39	
	ER, OPTIONAL, INTENT(OUT)	-	40	
			41	
	> BUF(*)	F, COUNT, DATATYPE, REQUEST, IERROR)	42	
• 1	ER FH, COUNT, DATATYPE, H	REQUEST, IERROR	43 44	
	ER(KIND=MPI_OFFSET_KIND)		44 45	
MPI F	ILE IWRITE AT is a nonbloci	king version of the MPI_FILE_WRITE_AT interface.	46	
			47	
			48	

```
1
      MPI_FILE_IWRITE_AT_ALL(fh, offset, buf, count, datatype, request)
2
       INOUT
                 fh
                                              file handle (handle)
3
       IN
                 offset
                                              file offset (integer)
4
5
       IN
                  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
      int MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
12
                     int count, MPI_Datatype datatype, MPI_Request *request)
13
     MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror)
14
          TYPE(MPI_File), INTENT(IN) :: fh
15
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
16
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
17
          INTEGER, INTENT(IN) :: count
18
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                   datatype
19
          TYPE(MPI_Request), INTENT(OUT) ::
                                                   request
20
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
21
22
      MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
23
          <type> BUF(*)
^{24}
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
25
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
26
          MPI_FILE_IWRITE_AT_ALL is a nonblocking version of MPI_FILE_WRITE_AT_ALL.
27
28
             Data Access with Individual File Pointers
29
      13.4.3
30
      MPI maintains one individual file pointer per process per file handle. The current value
^{31}
      of this pointer implicitly specifies the offset in the data access routines described in this
32
      section. These routines only use and update the individual file pointers maintained by MPI.
33
      The shared file pointer is not used nor updated.
34
          The individual file pointer routines have the same semantics as the data access with
35
      explicit offset routines described in Section 13.4.2, with the following modification:
36
37
         • the offset is defined to be the current value of the MPI-maintained individual file
38
           pointer.
39
      After an individual file pointer operation is initiated, the individual file pointer is updated
40
      to point to the next etype after the last one that will be accessed. The file pointer is updated
41
      relative to the current view of the file.
42
          If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
43
      to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.
44
45
46
47
48
```

MPI\_FILE\_READ(fh, buf, count, datatype, status)

	- READ(III, bui, count, datatype	e, status <i>j</i>	-
INOUT	fh	file handle (handle)	2 3
OUT	buf	initial address of buffer (choice)	4
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6
OUT	status	status object (Status)	7
001	Status		8 9
int MPI_	File_read(MPI_File fh, vo:	id *buf, int count, MPI_Datatype datatype,	10
	MPI_Status *status)		11
MPI_File	_read(fh, buf, count, data	atype, status, ierror)	12
	(MPI_File), INTENT(IN) ::		13 14
	(*), DIMENSION() :: bu	ıf	14 15
	GER, INTENT(IN) :: count		16
	(MPI_Datatype), INTENT(IN) (MPI_Status) :: status	) :: datatype	17
	GER, OPTIONAL, INTENT(OUT)	) :: ierror	18
MDT ETTE	_READ(FH, BUF, COUNT, DATA		19 20
	e> BUF(*)	AIIFE, STATUS, TERROR)	20
• -		STATUS(MPI_STATUS_SIZE), IERROR	22
MPI	FILE_READ reads a file using	the individual file pointer	23
		ine marviatar me pomoer.	24
-	0	code fragment is an example of reading a file until	25 26
the end o	f file is reached:		27
! Read	a preexisting input file	until all data has been read.	28
		if all requested data is read.	29
! The	Fortran 90 "exit" statemen	nt exits the loop.	30 31
in	togon bufgigo numrood	totprocessed, status(MPI_STATUS_SIZE)	31
	rameter (bufsize=100)	totprocessed, status(Mr1_S1A105_S1ZE)	33
-	al localbuffer(bufsiz	ze)	34
in	teger (kind=MPI_OFFSET_KIN	ND) zero	35
	<u>^</u>		36 37
ze	ro = 0		38
ca	11 MPI_FILE_OPEN( MPI_COMM	4_WORLD, 'myoldfile', &	39
		E_RDONLY, MPI_INFO_NULL, myfh, ierr )	40
ca		n, zero, MPI_REAL, MPI_REAL, 'native', &	41
- <b>ـ</b>		)_NULL, ierr )	42 43
to do	tprocessed = 0		43
40		, localbuffer, bufsize, MPI_REAL, &	45
	-	is, ierr )	46
		is, MPI_REAL, numread, ierr )	47
		harfford managed	48

Unofficial Draft for Comment Only

call process\_input( localbuffer, numread )

1

```
1
               totprocessed = totprocessed + numread
\mathbf{2}
                if ( numread < bufsize ) exit
3
            enddo
4
5
            write(6,1001) numread, bufsize, totprocessed
6
     1001 format( "No more data: read", I3, "and expected", I3, &
7
                     "Processed total of", I6, "before terminating job." )
8
9
            call MPI_FILE_CLOSE( myfh, ierr )
10
11
12
     MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
13
14
       INOUT
                 fh
                                             file handle (handle)
15
       OUT
                 buf
                                             initial address of buffer (choice)
16
       IN
                 count
                                             number of elements in buffer (integer)
17
18
       IN
                 datatype
                                             datatype of each buffer element (handle)
19
       OUT
                                             status object (Status)
                 status
20
21
     int MPI_File_read_all(MPI_File fh, void *buf, int count,
22
                    MPI_Datatype datatype, MPI_Status *status)
23
^{24}
     MPI_File_read_all(fh, buf, count, datatype, status, ierror)
25
          TYPE(MPI_File), INTENT(IN) ::
                                             fh
26
          TYPE(*), DIMENSION(...)
                                     :: buf
27
          INTEGER, INTENT(IN) :: count
28
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                  datatype
          TYPE(MPI_Status) :: status
29
30
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
^{31}
     MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
32
          <type> BUF(*)
33
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
34
35
          MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
36
37
     MPI_FILE_WRITE(fh, buf, count, datatype, status)
38
39
       INOUT
                 fh
                                             file handle (handle)
40
       IN
                 buf
                                             initial address of buffer (choice)
41
       IN
                 count
                                             number of elements in buffer (integer)
42
43
       IN
                 datatype
                                             datatype of each buffer element (handle)
44
       OUT
                 status
                                             status object (Status)
45
46
     int MPI_File_write(MPI_File fh, const void *buf, int count,
47
                    MPI_Datatype datatype, MPI_Status *status)
48
```

	write(fh, buf, count, dat	• -	1
	<pre>MPI_File), INTENT(IN) ::</pre>		2 3
	<pre> (), DIMENSION(), INTENT CR, INTENT(IN) :: count </pre>	(IN) :: buf	4
	<pre>IPI_Datatype), INTENT(IN)</pre>	:: datatvpe	5
	PI_Status) :: status	51	6
INTEGE	ER, OPTIONAL, INTENT(OUT)	:: ierror	7
MPI_FILE_W	RITE(FH, BUF, COUNT, DAT	ATYPE, STATUS, IERROR)	8 9
	• BUF(*)		9 10
INTEGE	ER FH, COUNT, DATATYPE, S	TATUS(MPI_STATUS_SIZE), IERROR	11
MPI_FI	LE_WRITE writes a file using	g the individual file pointer.	12
			13
MPI_FILE_\	NRITE_ALL(fh, buf, count, da	atatype, status)	14 15
INOUT	fh	file handle (handle)	16
IN	buf	initial address of buffer (choice)	17
			18
	count	number of elements in buffer (integer)	19 20
	datatype	datatype of each buffer element (handle)	20
OUT	status	status object (Status)	22
int MDT Ei	le mite all (MDT File fi	and which int and	23
IIIC MPI_FI		, const void *buf, int count, e, MPI_Status *status)	24
MDT E'I			25 26
	<pre>/rite_all(in, bui, count, IPI_File), INTENT(IN) ::</pre>	datatype, status, ierror) fh	27
	<pre>(), DIMENSION(), INTENT</pre>		28
	ER, INTENT(IN) :: count		29
	<pre>IPI_Datatype), INTENT(IN)</pre>	:: datatype	$30 \\ 31$
	PI_Status) :: status		32
INTEGE	ER, OPTIONAL, INTENT(OUT)	:: lerror	33
		DATATYPE, STATUS, IERROR)	34
<i>v</i> 1	BUF(*) B FH COUNT DATATYPE S	TATUS(MPI_STATUS_SIZE), IERROR	35
			$\frac{36}{37}$
	LE_WRITE_ALL is a collective	ve version of the blocking MPI_FILE_WRITE inter-	38
face.			39
			40
			41 42
			42 43
			44
			45
			46
			47

```
1
     MPI_FILE_IREAD(fh, buf, count, datatype, request)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       OUT
                buf
                                           initial address of buffer (choice)
4
5
       IN
                count
                                           number of elements in buffer (integer)
6
       IN
                                           datatype of each buffer element (handle)
                datatype
7
       OUT
                request
                                           request object (handle)
8
9
     int MPI_File_iread(MPI_File fh, void *buf, int count,
10
                    MPI_Datatype datatype, MPI_Request *request)
11
12
     MPI_File_iread(fh, buf, count, datatype, request, ierror)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
15
         INTEGER, INTENT(IN) :: count
16
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
17
         TYPE(MPI_Request), INTENT(OUT) ::
                                                request
18
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
19
     MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
20
          <type> BUF(*)
21
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
22
23
         MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.
24
25
     Example 13.3 The following Fortran code fragment illustrates file pointer update seman-
26
     tics:
27
28
         Read the first twenty real words in a file into two local
     !
29
     Т
         buffers. Note that when the first MPI_FILE_IREAD returns,
30
     !
         the file pointer has been updated to point to the
31
         eleventh real word in the file.
     I.
32
33
            integer
                       bufsize, req1, req2
34
            integer, dimension(MPI_STATUS_SIZE) :: status1, status2
35
            parameter (bufsize=10)
36
                       buf1(bufsize), buf2(bufsize)
            real
37
            integer (kind=MPI_OFFSET_KIND) zero
38
39
            zero = 0
40
            call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
41
                                  MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
42
            call MPI_FILE_SET_VIEW( myfh, zero, MPI_REAL, MPI_REAL, 'native', &
43
                                 MPI_INFO_NULL, ierr )
44
            call MPI_FILE_IREAD( myfh, buf1, bufsize, MPI_REAL, &
45
                                   req1, ierr )
46
            call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, &
47
                                   req2, ierr )
48
```

call	L MPI_WAIT( req1, status1	, ierr )	1		
	L MPI_WAIT( req2, status2		2		
			3 4		
call	call MPI_FILE_CLOSE( myfh, ierr )				
			5		
			7		
MPI_FILE_	IREAD_ALL(fh, buf, count, da	tatype, request)	8		
INOUT	fh	file handle (handle)	9		
OUT	buf	initial address of buffer (choice)	10		
IN	count	number of elements in buffer (integer)	11 12		
IN	datatype	datatype of each buffer element (handle)	13		
OUT	request	request object (handle)	14		
			15		
int MPI_F	ile_iread_all(MPI_File fh	, void *buf, int count,	16 17		
	MPI_Datatype datatype	e, MPI_Request *request)	18		
MPI_File_i	iread_all(fh, buf, count,	datatype, request, ierror)	19		
TYPE(N	MPI_File), INTENT(IN) ::	fh	20		
	*), DIMENSION(), ASYNCH	RONOUS :: buf	21		
	ER, INTENT(IN) :: count		22		
	<pre>MPI_Datatype), INTENT(IN) MDI_Baguagt) INTENT(OUT)</pre>	• -	23 24		
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror			24 25		
<pre>MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)</pre>					
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR					
	MPI_FILE_IREAD_ALL is a nonblocking version of MPI_FILE_READ_ALL.				
	ILL_INLAD_ALL IS a HOHDIOCH	ting version of WFT_TTEL_NEAD_AEL.	30 31		
			32		
MPI_FILE_	IWRITE(fh, buf, count, dataty	pe, request)	33		
INOUT	fh	file handle (handle)	34		
IN	buf	initial address of buffer (choice)	35 36		
IN	count	number of elements in buffer (integer)	37		
IN	datatype	datatype of each buffer element (handle)	38		
OUT	request	request object (handle)	39		
			40 41		
int MPI_F	ile_iwrite(MPI_File fh, c	onst void *buf, int count,	41 42		
	MPI_Datatype datatype	e, MPI_Request *request)	43		
MPI_File_f	iwrite(fh, buf, count, da	tatype, request, ierror)	44		
	MPI_File), INTENT(IN) ::	fh	45		
	<pre>*), DIMENSION(), INTENT</pre>	(IN), ASYNCHRONOUS :: buf	46		
	ER, INTENT(IN) :: count		47 48		
TYPE(N	TYPE(MPI_Datatype), INTENT(IN) :: datatype 4				

```
1
         TYPE(MPI_Request), INTENT(OUT) :: request
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
3
     MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
4
          <type> BUF(*)
5
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
6
7
         MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.
8
9
     MPI_FILE_IWRITE_ALL(fh, buf, count, datatype, request)
10
11
       INOUT
                fh
                                            file handle (handle)
12
       IN
                buf
                                            initial address of buffer (choice)
13
                                            number of elements in buffer (integer)
       IN
                count
14
15
       IN
                datatype
                                            datatype of each buffer element (handle)
16
       OUT
                                            request object (handle)
                request
17
18
     int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count,
19
                    MPI_Datatype datatype, MPI_Request *request)
20
21
     MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
22
         TYPE(MPI_File), INTENT(IN) :: fh
23
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
^{24}
         INTEGER, INTENT(IN) :: count
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         TYPE(MPI_Request), INTENT(OUT) ::
                                                request
27
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
28
     MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
29
         <type> BUF(*)
30
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
^{31}
32
         MPI_FILE_IWRITE_ALL is a nonblocking version of MPI_FILE_WRITE_ALL.
33
34
     MPI_FILE_SEEK(fh, offset, whence)
35
36
       INOUT
                fh
                                            file handle (handle)
37
       IN
                offset
                                            file offset (integer)
38
       IN
                whence
                                            update mode (state)
39
40
41
     int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
42
     MPI_File_seek(fh, offset, whence, ierror)
43
         TYPE(MPI_File), INTENT(IN) :: fh
44
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
45
         INTEGER, INTENT(IN) :: whence
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
     MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
```

ТМТІ	EGER FH, WHENCE, IERROR		1	
	EGER(KIND=MPI_OFFSET_KIND	) OFFSET	2	
MDI	Ell E SEEK undated the indiv	vidual file pointer according to whence, which has the	3	
	possible values:	vidual life pointer according to whence, which has the	4	
U	-		5 6	
• MP	I_SEEK_SET: the pointer is set	t to offset	7	
• MP	I_SEEK_CUR: the pointer is se	t to the current pointer position plus $offset$	8	
• MP	I_SEEK_END: the pointer is se	t to the end of file plus offset	9 10	
The	offset can be negative, which	allows seeking backwards. It is erroneous to seek to	11	
	re position in the view.		12	
			13	
	E_GET_POSITION(fh, offset)		14	
			15	
IN	fh	file handle (handle)	16 17	
OUT	offset	offset of individual pointer (integer)	18	
			19	
int MPI	_File_get_position(MPI_Fi	le fh, MPI_Offset *offset)	20	
	e_get_position(fh, offset		21	
	E(MPI_File), INTENT(IN) :		22	
	EGER(KIND=MPI_OFFSET_KIND		23 24	
	EGER, OPTIONAL, INTENT(OU	T) :: ierror	24	
	MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)			
	INTEGER FH, IERROR			
INTE	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET			
MPI.	MPI_FILE_GET_POSITION returns, in offset, the current position of the individual file			
pointer in	n etype units relative to the c	urrent view.	30	
Ada	vice to users. The offset car	h be used in a future call to MPI_FILE_SEEK using	31 32	
		rn to the current position. To set the displacement to	33	
		irst convert offset into an absolute byte position using	34	
MP	I_FILE_GET_BYTE_OFFSET	, then call MPI_FILE_SET_VIEW with the resulting	35	
disj	placement. (End of advice to	users.)	36	
			37	
			38	
MPI_FILE	E_GET_BYTE_OFFSET(fh, of	fset, disp)	39 40	
IN	fh	file handle (handle)	40	
IN	offset	offset (integer)	42	
OUT	disp	absolute byte position of offset (integer)	43	
001	uisp	absolute byte position of onset (integer)	44	
int MPT	File get byte offset(MPT	_File fh, MPI_Offset offset,	45	
	MPI_Offset *disp)		46	
ינים דסא	-	act dian iconce	47 48	
MF1_F110	MPI_File_get_byte_offset(fh, offset, disp, ierror) 48			

1 2 3 4	INTEG INTEG	MPI_File), INTENT(IN) ER(KIND=MPI_OFFSET_KIN ER(KIND=MPI_OFFSET_KIN ER, OPTIONAL, INTENT(C	D), INTENT(IN) :: offset D), INTENT(OUT) :: disp
5 6 7 8	INTEG	GET_BYTE_OFFSET(FH, OF ER FH, IERROR ER(KIND=MPI_OFFSET_KIN	
9 10 11 12	position. T		converts a view-relative offset into an absolute byte (from the beginning of the file) of offset relative to the
13	13.4.4 Da	ata Access with Shared Fi	le Pointers
14 15 16 17 18 19 20	processes in the offset in update the nor update The sh	n the communicator group n the data access routines e shared file pointer mainta ed. nared file pointer routines h	pointer per collective MPI_FILE_OPEN (shared among ). The current value of this pointer implicitly specifies described in this section. These routines only use and ined by MPI. The individual file pointers are not used ave the same semantics as the data access with explicit
21 22	offset routi	nes described in Section 1	3.4.2, with the following modifications:
23	• the o	ffset is defined to be the cu	$% \mathcal{M}^{(1)}$ is a state of the $MPI\text{-maintained}$ shared file pointer,
24 25 26		ffect of multiple calls to sh were serialized, and	ared file pointer routines is defined to behave as if the
20 27 28	• the u file v	_	outines is erroneous unless all processes use the same
29 30 31 32 33 34	istic. The After point to th	user needs to use other syn a shared file pointer opera	ter routines, the serialization ordering is not determin- nchronization means to enforce a specific order. tion is initiated, the shared file pointer is updated to one that will be accessed. The file pointer is updated e.
35 36 37	Noncollecti	ve Operations	
38 39	MPI_FILE_	.READ_SHARED(fh, buf, c	ount, datatype, status)
40	INOUT	fh	file handle (handle)
41 42	OUT	buf	initial address of buffer (choice)
43	IN	count	number of elements in buffer (integer)
44	IN	datatype	datatype of each buffer element (handle)
45 46 47 48	OUT	status	status object (Status)

int MPI_File	_read_shared(MPI_File f MPI_Datatype datatype	fh, void *buf, int count, . MPI Status *status)	$\frac{1}{2}$	
			3	
MPI_File_read_shared(fh, buf, count, datatype, status, ierror)				
	_File), INTENT(IN) ::		5	
	DIMENSION() :: buf	Ê	6	
=	INTENT(IN) :: count		7	
	_Datatype), INTENT(IN)	:: datatype	8	
TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror			9	
INTEGER,	DITIONAL, INTENI(001)	101101	10	
		I, DATATYPE, STATUS, IERROR)	11	
<type> BI</type>			12	
INTEGER I	FH, COUNT, DATATYPE, SI	TATUS(MPI_STATUS_SIZE), IERROR	13	
MPI_FILE	_READ_SHARED reads a fi	ile using the shared file pointer.	14 15	
		0 1	16	
			17	
MPI_FILE_WR	RITE_SHARED(fh, buf, coun	t, datatype, status)	18	
INOUT fh		file handle (handle)	19	
IN bu	f	initial address of buffer (choice)	20	
		number of elements in buffer (integer)	21	
			22	
IN dat	tatype	datatype of each buffer element (handle)	23	
OUT sta	atus	status object (Status)	24	
			25	
int MPI_File	_write_shared(MPI_File	fh, const void *buf, int count,	26	
	MPI_Datatype datatype	, MPI_Status *status)	27	
MPT File writ	te shared (fh buf cour	nt, datatype, status, ierror)	28	
	_File), INTENT(IN) ::	fh	29 30	
	DIMENSION(), INTENT(		31	
	INTENT(IN) :: count		32	
-	_Datatype), INTENT(IN)	:: datatype	33	
TYPE(MPI	_Status) :: status		34	
INTEGER,	OPTIONAL, INTENT(OUT)	:: ierror	35	
MDT ETTE WRT		NT, DATATYPE, STATUS, IERROR)	36	
<pre>type&gt; Bl</pre>		VI, DAIATTE, STATOS, TEMON	37	
• -		TATUS(MPI_STATUS_SIZE), IERROR	38	
			39	
MPI_FILE	_WRITE_SHARED writes a	file using the shared file pointer.	40	
			41	
			42	
			43	
			44	

47 48

 $45 \\ 46$ 

```
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
                count
                                            number of elements in buffer (integer)
6
       IN
                                            datatype of each buffer element (handle)
                datatype
7
       OUT
                request
                                            request object (handle)
8
9
     int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
10
                    MPI_Datatype datatype, MPI_Request *request)
11
12
     MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
15
         INTEGER, INTENT(IN) :: count
16
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
17
         TYPE(MPI_Request), INTENT(OUT) :: request
18
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
19
     MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
20
          <type> BUF(*)
21
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
22
23
         MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
^{24}
     interface.
25
26
27
     MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
28
       INOUT
                                            file handle (handle)
                fh
29
       IN
                buf
                                            initial address of buffer (choice)
30
31
       IN
                count
                                            number of elements in buffer (integer)
32
       IN
                datatype
                                            datatype of each buffer element (handle)
33
       OUT
                request
                                            request object (handle)
34
35
36
     int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
37
                    MPI_Datatype datatype, MPI_Request *request)
38
     MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
39
         TYPE(MPI_File), INTENT(IN) :: fh
40
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
41
         INTEGER, INTENT(IN) :: count
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         TYPE(MPI_Request), INTENT(OUT) ::
                                                request
44
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
45
46
     MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
47
          <type> BUF(*)
48
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

# MPI\_FILE\_IWRITE\_SHARED is a nonblocking version of the MPI\_FILE\_WRITE\_SHARED interface.

#### **Collective Operations**

The semantics of a collective access using a shared file pointer is that the accesses to the file will be in the order determined by the ranks of the processes within the group. For each process, the location in the file at which data is accessed is the position at which the shared file pointer would be after all processes whose ranks within the group less than that of this process had accessed their data. In addition, in order to prevent subsequent shared offset accesses by the same processes from interfering with this collective access, the call might return only after all the processes within the group have initiated their accesses. When the call returns, the shared file pointer points to the next etype accessible, according to the file view used by all processes, after the last etype requested.

Advice to users. There may be some programs in which all processes in the group need to access the file using the shared file pointer, but the program may not *require* that data be accessed in order of process rank. In such programs, using the shared ordered routines (e.g., MPI\_FILE\_WRITE\_ORDERED rather than MPI\_FILE\_WRITE\_SHARED) may enable an implementation to optimize access, improving performance. (*End of advice to users.*)

Advice to implementors. Accesses to the data requested by all processes do not have to be serialized. Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel. (*End of advice to implementors.*)

## MPI\_FILE\_READ\_ORDERED(fh, buf, count, datatype, status)

INOUT	fh	file handle (handle)	30
OUT	buf	initial address of buffer (choice)	31 32
	501	initial dealess of sunor (choice)	
IN	count	number of elements in buffer (integer)	33
IN	datatype	datatype of each buffer element (handle)	34
	addatype		35
OUT	status	status object (Status)	36
			37
int MPI_F	ile_read_ordered(MPI_File	e fh, void *buf, int count,	38
	MPI_Datatype datatype	e, MPI_Status *status)	39
MDT Eile	and and and (fb buf and		40
		nt, datatype, status, ierror)	41
	<pre>MPI_File), INTENT(IN) ::</pre>		42
	*), DIMENSION() :: bu	lf	43
	ER, INTENT(IN) :: count		44
TYPE(	MPI_Datatype), INTENT(IN)	:: datatype	45
TYPE(	MPI_Status) :: status		46
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	
			47
MPI_FILE_	READ_ORDERED(FH, BUF, COU	INT, DATATYPE, STATUS, IERROR)	48

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $^{24}$ 

1 2	<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR</type>			
3 4 5	MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED interface.			
6 7	MPI FILE	_WRITE_ORDERED(fh, buf,	count. datatype. status)	
8	INOUT	fh	file handle (handle)	
9 10	IN	buf	initial address of buffer (choice)	
11	IN	count	number of elements in buffer (integer)	
12	IN	datatype	datatype of each buffer element (handle)	
13 14	OUT	status	status object (Status)	
15	001	Status	status object (Status)	
16 17 18	int MPI_F		ile fh, const void *buf, int count, pe, MPI_Status *status)	
19 20 21 22 23 24 25 26 27	TYPE( TYPE( INTEC TYPE( TYPE( INTEC MPI_FILE_	(MPI_File), INTENT(IN) : (*), DIMENSION(), INTE GER, INTENT(IN) :: coum (MPI_Datatype), INTENT(I (MPI_Status) :: status GER, OPTIONAL, INTENT(OU	NT(IN) :: buf t N) :: datatype	
28 29	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR			
30 31 32	interface.	FILE_WRITE_ORDERED is a	collective version of the $MPI\_FILE\_WRITE\_SHARED$	
33 34	Seek			
35 36 37 38	to call the		specified when the file was opened, it is erroneous <code>IPI_FILE_SEEK_SHARED</code> and	
39	MPI_FILE	_SEEK_SHARED(fh, offset, w	vhence)	
40 41	INOUT	fh	file handle (handle)	
41	IN	offset	file offset (integer)	
43	IN	whence	update mode (state)	
44 45	int MPI_F	File_seek_shared(MPI_File	e fh, MPI_Offset offset, int whence)	
46 47 48	MPI_File_	_seek_shared(fh, offset, (MPI_File), INTENT(IN) :	whence, ierror)	

INTEGER, INTENT(IN) ::	
INTEGER, OPTIONAL, INT	
MPI_FILE_SEEK_SHARED(FH, O	FFSET, WHENCE, IERROR)
INTEGER FH, WHENCE, IEI	5
INTEGER(KIND=MPI_OFFSE	0
	updates the shared me pointer according to whence, which
has the following possible value	5. 10
• MPI_SEEK_SET: the pointe	er is set to offset 11
• MPI_SEEK_CUR: the point	er is set to the current pointer position plus offset $12$
• MPI_SEEK_END: the point	er is set to the end of file plus offset 14
MPI FILE SEEK SHARED	is collective; all the processes in the communicator group
	must call MPL FILE SEEK SHARED with the same values
for offset and whence.	17
	which allows seeking backwards. It is erroneous to seek to
a negative position in the view.	19
	20
MPI_FILE_GET_POSITION_SH	ARED(fh, offset) 22 23
IN fh	file handle (handle) 24
OUT offset	offset of shared pointer (integer) 25
	26
int MPI_File_get_position_:	shared(MPI_File fh, MPI_Offset *offset) 27
MPI_File_get_position_share	ed(fh. offset. ierror)
TYPE(MPI_File), INTENT	(TN) ·· fh
	r KIND) INTENT(OUT) ·· offset
INTEGER, OPTIONAL, INT	- 31
NET ETTE GET DOGTTON GUAD	
MPI_FILE_GET_POSITION_SHAR	ED(FH, UFFSET, IERRUR) 34
INTEGER FH, IERROR	
INTEGER(KIND=MPI_OFFSE	I_KIND) UFFSEI 36
MPI_FILE_GET_POSITION	_SHARED returns, in offset, the current position of the 37
shared file pointer in etype unit	s relative to the current view. 38
	39
	et can be used in a future call to MPI_FILE_SEEK_SHARED
8	SET to return to the current position. To set the displace- $41$
-	voluter position first convert <b>offset</b> into an absolute byte
position using MPI_FILE_	pointer position, first convert offset into an absolute byte
	GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with
	GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with 42
the resulting displacement	GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with 43 44 45
	GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with 43 44 45
the resulting displacement 13.4.5 Split Collective Data A MPI provides a restricted form	GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with 42 43 44 45 44 45 44 45 45 45 45 45 45 45 45

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.
- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an MPI\_FILE\_READ\_ALL on one process does not match an MPI\_FILE\_READ\_ALL\_BEGIN/ MPI\_FILE\_READ\_ALL\_END pair on another process.
  - Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid the problems described in "A Problem with Code Movements and Register Optimization," Section 18.1.17, but not all of the problems, such as those described in Sections 18.1.12, 18.1.13, and 18.1.16.
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

```
MPI_File_read_all_begin(fh, ...);
...
MPI_File_read_all(fh, ...);
...
MPI_File_read_all_end(fh, ...);
```

is erroneous.

 $^{31}$ 

• In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

6 The arguments for these routines have the same meaning as for the equivalent collective  $\overline{7}$ versions (e.g., the argument definitions for MPI\_FILE\_READ\_ALL\_BEGIN and 8 MPI\_FILE\_READ\_ALL\_END are equivalent to the arguments for MPI\_FILE\_READ\_ALL). 9 The begin routine (e.g., MPI\_FILE\_READ\_ALL\_BEGIN) begins a split collective operation 10 that, when completed with the matching end routine (i.e., MPI\_FILE\_READ\_ALL\_END) 11 produces the result as defined for the equivalent collective routine (i.e., 12MPI\_FILE\_READ\_ALL). 13 For the purpose of consistency semantics (Section 13.6.1), a matched pair of split 14collective data access operations (e.g., MPI\_FILE\_READ\_ALL\_BEGIN and 15MPI\_FILE\_READ\_ALL\_END) compose a single data access. 1617 18 MPI\_FILE\_READ\_AT\_ALL\_BEGIN(fh, offset, buf, count, datatype) 19 IN fh file handle (handle) 2021offset file offset (integer) IN 22 OUT buf initial address of buffer (choice) 23IN count number of elements in buffer (integer)  $^{24}$ 25IN datatype datatype of each buffer element (handle) 2627int MPI\_File\_read\_at\_all\_begin(MPI\_File fh, MPI\_Offset offset, void \*buf, 28int count, MPI\_Datatype datatype) 2930 MPI\_File\_read\_at\_all\_begin(fh, offset, buf, count, datatype, ierror) TYPE(MPI\_File), INTENT(IN) :: fh 31INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: 32 offset 33 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 34 INTEGER, INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: 35datatype INTEGER, OPTIONAL, INTENT(OUT) :: 36 ierror

531

1

 $\mathbf{2}$ 

3

4

5

37

38

39

40

```
1
     MPI_FILE_READ_AT_ALL_END(fh, buf, status)
\mathbf{2}
       IN
                fh
                                            file handle (handle)
3
       OUT
                 buf
                                            initial address of buffer (choice)
4
5
       OUT
                                            status object (Status)
                status
6
7
     int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
8
     MPI_File_read_at_all_end(fh, buf, status, ierror)
9
         TYPE(MPI_File), INTENT(IN) :: fh
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
11
         TYPE(MPI_Status) :: status
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
15
          <type> BUF(*)
16
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
17
18
19
     MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
20
21
       INOUT
                fh
                                            file handle (handle)
22
       IN
                offset
                                            file offset (integer)
23
       IN
                 buf
                                            initial address of buffer (choice)
24
25
       IN
                count
                                            number of elements in buffer (integer)
26
                                            datatype of each buffer element (handle)
       IN
                 datatype
27
28
     int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
29
                    const void *buf, int count, MPI_Datatype datatype)
30
^{31}
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
         TYPE(MPI_File), INTENT(IN) :: fh
32
33
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
34
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
35
         INTEGER, INTENT(IN) :: count
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
37
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
38
     MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
39
         <type> BUF(*)
40
         INTEGER FH, COUNT, DATATYPE, IERROR
41
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
42
43
44
45
46
47
48
```

1 MPI\_FILE\_WRITE\_AT\_ALL\_END(fh, buf, status)  $\mathbf{2}$ INOUT fh file handle (handle) 3 IN buf initial address of buffer (choice) 4 OUT status object (Status) 5status 6 7 int MPI\_File\_write\_at\_all\_end(MPI\_File fh, const void \*buf, 8 MPI\_Status \*status) 9 MPI\_File\_write\_at\_all\_end(fh, buf, status, ierror) 10 TYPE(MPI\_File), INTENT(IN) :: fh 11 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 12TYPE(MPI\_Status) :: status 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415MPI\_FILE\_WRITE\_AT\_ALL\_END(FH, BUF, STATUS, IERROR) 16<type> BUF(\*) 17INTEGER FH, STATUS(MPI\_STATUS\_SIZE), IERROR 18 19 20MPI\_FILE\_READ\_ALL\_BEGIN(fh, buf, count, datatype) 21INOUT fh file handle (handle) 22 23OUT buf initial address of buffer (choice)  $^{24}$ IN count number of elements in buffer (integer) 25IN datatype datatype of each buffer element (handle) 2627int MPI\_File\_read\_all\_begin(MPI\_File fh, void \*buf, int count, 28 29 MPI\_Datatype datatype) 30 MPI\_File\_read\_all\_begin(fh, buf, count, datatype, ierror) 31TYPE(MPI\_File), INTENT(IN) :: fh 32 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf 33 INTEGER, INTENT(IN) :: count 34 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 37 MPI\_FILE\_READ\_ALL\_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) 38 <type> BUF(\*) 39 INTEGER FH, COUNT, DATATYPE, IERROR 40 41 42MPI\_FILE\_READ\_ALL\_END(fh, buf, status) 43 INOUT fh file handle (handle) 44OUT 45buf initial address of buffer (choice) 46OUT status object (Status) status 47

```
1
     int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
\mathbf{2}
     MPI_File_read_all_end(fh, buf, status, ierror)
3
         TYPE(MPI_File), INTENT(IN) :: fh
4
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
5
         TYPE(MPI_Status) :: status
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
9
         <type> BUF(*)
10
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
11
12
13
     MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
14
       INOUT
                fh
                                           file handle (handle)
15
16
       IN
                buf
                                           initial address of buffer (choice)
17
       IN
                count
                                           number of elements in buffer (integer)
18
       IN
                datatype
                                           datatype of each buffer element (handle)
19
20
21
     int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
                    MPI_Datatype datatype)
22
23
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
24
         TYPE(MPI_File), INTENT(IN) :: fh
25
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
26
         INTEGER, INTENT(IN) :: count
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
29
     MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
30
^{31}
         <type> BUF(*)
         INTEGER FH, COUNT, DATATYPE, IERROR
32
33
34
35
     MPI_FILE_WRITE_ALL_END(fh, buf, status)
36
       INOUT
                fh
                                           file handle (handle)
37
       IN
                buf
                                           initial address of buffer (choice)
38
39
       OUT
                status
                                           status object (Status)
40
41
     int MPI_File_write_all_end(MPI_File fh, const void *buf,
42
                    MPI_Status *status)
43
     MPI_File_write_all_end(fh, buf, status, ierror)
44
         TYPE(MPI_File), INTENT(IN) :: fh
45
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
46
47
         TYPE(MPI_Status) :: status
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

	WRITE_ALL_END(FH, BUF, ST > BUF(*)	ATUS, IERROR)	1 2
• 1	ER FH, STATUS(MPI_STATUS_	SIZE), IERROR	3
			4
			5
MPI_FILE_	READ_ORDERED_BEGIN(fh,	buf, count, datatype)	6 7
INOUT	fh	file handle (handle)	8
OUT	buf	initial address of buffer (choice)	9
IN	count	number of elements in buffer (integer)	10
IN	datatype	datatype of each buffer element (handle)	11
	datatype	datatype of each build clement (nandie)	12
int MPI_F:	ile_read_ordered_begin(MP	I_File fh, void *buf, int count,	13 14
	MPI_Datatype datatype		15
MPT File '	read ordered begin(fh bu	f, count, datatype, ierror)	16
	MPI_File), INTENT(IN) ::	• •	17
TYPE(	*), DIMENSION(), ASYNCH	RONOUS :: buf	18
	ER, INTENT(IN) :: count		19 20
	MPI_Datatype), INTENT(IN)	01	21
INTEG	ER, OPTIONAL, INTENT(OUT)	:: lerror	22
		F, COUNT, DATATYPE, IERROR)	23
• 1	> BUF(*)		24
INTEG	ER FH, COUNT, DATATYPE, I	ERRUR	25 26
			20 27
MPL FILF	READ_ORDERED_END(fh, bi	if status)	28
INOUT	fh	file handle (handle)	29
			30
OUT	buf	initial address of buffer (choice)	31
OUT	status	status object (Status)	32 33
			34
int MPI_F	ile_read_ordered_end(MPI_	File fh, void *buf, MPI_Status *status)	35
MPI_File_:	read_ordered_end(fh, buf,	status, ierror)	36
	MPI_File), INTENT(IN) ::		37
	*), DIMENSION(), ASYNCH	RONOUS :: buf	38
	MPI_Status) :: status ER, OPTIONAL, INTENT(OUT)	:: ierror	39 40
			41
	READ_ORDERED_END(FH, BUF,	STATUS, IERROR)	42
01	> BUF(*) ER FH, STATUS(MPI_STATUS_	STZE) TERROR	43
1111201	LIVIN, SINIOS(IIII_SINIOS_		44
			45
			46 47
			48

```
1
     MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
2
       INOUT
                fh
                                            file handle (handle)
3
                buf
       IN
                                            initial address of buffer (choice)
4
5
       IN
                                            number of elements in buffer (integer)
                count
6
       IN
                datatype
                                            datatype of each buffer element (handle)
7
8
     int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
9
                    MPI_Datatype datatype)
10
11
     MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
14
         INTEGER, INTENT(IN) :: count
15
         TYPE(MPI_Datatype), INTENT(IN) ::
                                                datatype
16
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
17
     MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
18
          <type> BUF(*)
19
         INTEGER FH, COUNT, DATATYPE, IERROR
20
21
22
     MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
23
24
       INOUT
                fh
                                            file handle (handle)
25
       IN
                buf
                                            initial address of buffer (choice)
26
       OUT
                status
                                            status object (Status)
27
28
29
     int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
30
                    MPI_Status *status)
^{31}
     MPI_File_write_ordered_end(fh, buf, status, ierror)
32
         TYPE(MPI_File), INTENT(IN) :: fh
33
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS ::
                                                                   buf
34
         TYPE(MPI_Status) :: status
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
38
          <type> BUF(*)
39
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
40
41
42
     13.5
             File Interoperability
43
```

At the most basic level, file interoperability is the ability to read the information previously
 written to a file — not just the bits of data, but the actual information the bits represent.
 MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation
 (Section 13.5.2) as well as the data conversion functions (Section 13.5.3).

Interoperability within a single MPI environment (which could be considered "operability") ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 13.6.1), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI\_COMM\_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar operations similar to POSIX cp, rm, and mv can be performed on the file. Furthermore, it is expected that the facility provided maintains the correspondence between absolute byte offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the MPI environment are at byte offset 102 outside the MPI environment). As an example, a simple off-line conversion utility that transfers and converts files between the native file system and the MPI environment would suffice, provided it maintained the offset coherence mentioned above. In a high-quality implementation of MPI, users will be able to manipulate MPI files using the same or similar tools that the native file system offers for manipulating its files.

The remaining aspect of file interoperability, converting between different machine representations, is supported by the typing information specified in the etype and filetype. This facility allows the information in files to be shared between any two applications, regardless of whether they use MPI, and regardless of the machine architectures on which they run.

MPI supports multiple data representations: "native," "internal," and "external32." An implementation may support additional data representations. MPI also supports userdefined data representations (see Section 13.5.3). The "native" and "internal" data representations are implementation dependent, while the "external32" representation is common to all MPI implementations and facilitates file interoperability. The data representation is specified in the datarep argument to MPI\_FILE\_SET\_VIEW.

Advice to users. MPI is not guaranteed to retain knowledge of what data representation was used when a file is written. Therefore, to correctly retrieve file data, an MPI 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
 46
 lost in type conversions with a purely homogeneous environment. The disadvantage
 47
 is the loss of transparent interoperability within a heterogeneous MPI environment.

#### Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$ 

6 7

8 9

10

11 12

13

14 15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39 40

41

42

43

1	
2	Advice to users. This data representation should only be used in a homogeneous
3	MPI environment, or when the MPI application is capable of performing the data turne conversions itself. (End of advice to weere)
4	type conversions itself. (End of advice to users.)
5	Advice to implementors. When implementing read and write operations on
6	top of MPI message-passing, the message data should be typed as MPI_BYTE
7	to ensure that the message routines do not perform any type conversions on the
8	data. (End of advice to implementors.)
9	
10	"internal" This data representation can be used for I/O operations in a homogeneous
11	or heterogeneous environment; the implementation will perform type conversions if
12	necessary. The implementation is free to store data in any format of its choice, with
13	the restriction that it will maintain constant extents for all predefined datatypes in any
14	one file. The environment in which the resulting file can be reused is implementation-
15	defined and must be documented by the implementation.
16	
17	Rationale. This data representation allows the implementation to perform $I/O$
18	efficiently in a heterogeneous environment, though with implementation-defined
19	restrictions on how the file can be reused. (End of rationale.)
20	
21	Advice to implementors. Since "external32" is a superset of the functionality
22	provided by "internal," an implementation may choose to implement "internal"
23	as "external32." (End of advice to implementors.)
24	"external32" This data representation states that read and write operations convert all
25	data from and to the "external32" representation defined in Section 13.5.2. The data
26	conversion rules for communication also apply to these conversions (see Section 3.3.2).
27	The data on the storage medium is always in this canonical representation, and the
28	data in memory is always in the local process's native representation.
29	This data representation has several advantages. First, all processes reading the file
30 31	in a heterogeneous MPI environment will automatically have the data converted to
32	their respective native representations. Second, the file can be exported from one MPI
33	environment and imported into any other MPI environment with the guarantee that
34	the second environment will be able to read all the data in the file.
35	
36	The disadvantage of this data representation is that data precision and I/O perfor- mance may be lost in data type conversions.
37	mance may be lost in data type conversions.
38	Advice to implementors. When implementing read and write operations on top
39	of MPI message-passing, the message data should be converted to and from the
40	"external32" representation in the client, and sent as type MPI_BYTE. This will
41	avoid possible double data type conversions and the associated further loss of
42	precision and performance. (End of advice to implementors.)
43	
44	13.5.1 Datatypes for File Interoperability
45	
46	If the file data representation is other than "native," care must be taken in constructing
47	etypes and filetypes. Any of the datatype constructor functions may be used; however,
48	for those functions that accept displacements in bytes, the displacements must be specified

in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI\_FILE\_GET\_TYPE\_EXTENT can be used to calculate the extents of datatypes in the file. For etypes and filetypes that are portable datatypes (see Section 2.4), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

Advice to users. One can logically think of the file as if it were stored in the memory 9 of a file server. The etype and filetype are interpreted as if they were defined at this 10 file server, by the same sequence of calls used to define them at the calling process. 11 If the data representation is "native", then this logical file server runs on the same 12architecture as the calling process, so that these types define the same data layout 13 on the file as they would define in the memory of the calling process. If the etype 14 and filetype are portable datatypes, then the data layout defined in the file is the 15same as would be defined in the calling process memory, up to a scaling factor. The 16 routine MPI\_FILE\_GET\_TYPE\_EXTENT can be used to calculate this scaling factor. 17 Thus, two equivalent, portable datatypes will define the same data layout in the file, 18 even in a heterogeneous environment with "internal", "external32", or user defined 19 data representations. Otherwise, the etype and filetype must be constructed so that 20their typemap and extent are the same on any architecture. This can be achieved 21if they have an explicit upper bound and lower bound (defined using 22

MPI\_TYPE\_CREATE\_RESIZED). This condition must also be fulfilled by any datatype 23that is used in the construction of the etype and filetype, if this datatype is replicated contiguously, either explicitly, by a call to MPI\_TYPE\_CONTIGUOUS, or implicitly, by a blocklength argument that is greater than one. If an etype or filetype is not portable, and has a typemap or extent that is architecture dependent, then the data 27layout specified by it on a file is implementation dependent.

File data representations other than "native" may be different from corresponding data representations in memory. Therefore, for these file data representations, it is important not to use hardwired byte offsets for file positioning, including the initial displacement that specifies the view. When a portable datatype (see Section 2.4) is used in a data access operation, any holes in the datatype are scaled to match the data representation. However, note that this technique only works when all the processes that created the file view build their etypes from the same predefined datatypes. For example, if one process uses an etype built from MPI\_INT and another uses an etype built from MPI\_FLOAT, the resulting views may be nonportable because the relative sizes of these types may differ from one data representation to another. (End of advice to users.)

MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)		
IN	fh	file handle (handle)
IN	datatype	datatype (handle)
OUT	extent	datatype extent (integer)

#### **Unofficial Draft for Comment Only**

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8

 $^{24}$ 

25

26

2829

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40 41 42

MPI\_File\_get\_type\_extent(fh, datatype, extent, ierror) TYPE(MPI\_File), INTENT(IN) :: fh TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: extent INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_FILE\_GET\_TYPE\_EXTENT(FH, DATATYPE, EXTENT, IERROR) INTEGER FH, DATATYPE, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTENT Returns the extent of datatype in the file fh. This extent will be the same for all processes accessing the file fh. If the current view uses a user-defined data representation (see Section 13.5.3), MPI uses the dtype\_file\_extent\_fn callback to calculate the extent. 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 13.5.3). (End of advice to implementors.) 13.5.2 External Data Representation: "external32" All MPI implementations are required to support the data representation defined in this section. Support of optional datatypes (e.g., MPI\_INTEGER2) is not required. All floating point values are in big-endian IEEE format [37] of the appropriate size. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +16383, 112 fraction bits, and an encoding analogous to the "Double" format. All integral values are in two's complement big-endian format. Bigendian means most significant byte at lowest address byte. For C \_Bool, Fortran LOGICAL, and C++ bool, 0 implies false and nonzero implies true. C float \_Complex, double \_Complex, and long double \_Complex, Fortran COMPLEX and DOUBLE COMPLEX, and other complex types are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [38]. Wide characters (of type MPI\_WCHAR) are in Unicode format [59]. All signed numerals (e.g., MPI\_INT, MPI\_REAL) have the sign bit at the most significant bit. MPI\_COMPLEX and MPI\_DOUBLE\_COMPLEX have the sign bit of the real and imaginary parts at the most significant bit of each part. According to IEEE specifications [37], the "NaN" (not a number) is system dependent. It should not be interpreted within MPI as anything other than "NaN." Advice to implementors. The MPI treatment of "NaN" is similar to the approach used in XDR (see ftp://ds.internic.net/rfc/rfc1832.txt). (End of advice to implementors.) All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

1

 $\mathbf{2}$ 

3

4

5

6

7

8 9

10

11

12

13

14

1516

17

18

19

2021

22 23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

 $^{41}$ 

4243

44

4546

47

Advice to implementors. All bytes of LOGICAL and bool must be checked to determine the value. (End of advice to implementors.)

Advice to users. The type MPI\_PACKED is treated as bytes and is not converted. The user should be aware that MPI\_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

The sizes of the predefined datatypes returned from MPI\_TYPE\_CREATE\_F90\_REAL, MPI\_TYPE\_CREATE\_F90\_COMPLEX, and MPI\_TYPE\_CREATE\_F90\_INTEGER are defined in Section 18.1.9, page 647.

Advice to implementors. When converting a larger size integer to a smaller size integer, only the least significant bytes are moved. Care must be taken to preserve the sign bit value. This allows no conversion errors if the data range is within the range of the smaller size integer. (End of advice to implementors.)

Table 13.2 specifies the sizes of predefined datatypes in "external32" format.

#### 13.5.3 User-Defined Data Representations

There are two situations that cannot be handled by the required representations:

1. a user wants to write a file in a representation unknown to the implementation, and

2. a user wants to read a file written in a representation unknown to the implementation.

User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.

# MPI\_REGISTER\_DATAREP(datarep, read\_conversion\_fn, write\_conversion\_fn, dtype\_file\_extent\_fn, extra\_state)

IN	datarep	data representation identifier (string)	31
IN	read_conversion_fn	function invoked to convert from file representation to	32
		native representation (function)	33
	·. · · ·		34
IN	write_conversion_fn	function invoked to convert from native representation	35
		to file representation (function)	36
IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as	37
		represented in the file (function)	38
IN	extra_state	extra state	39
IIN	extra_state		40
NDT			41
int MPI.	_Register_datarep(const c	-	42
	-	sion_function *read_conversion_fn,	43
	-	sion_function *write_conversion_fn,	44
	-	function *dtype_file_extent_fn,	45
	void *extra_state)		46
MPI_Regi	ister_datarep(datarep, re	ad_conversion_fn, write_conversion_fn,	47

Unofficial Draft for Comment Only

dtype\_file\_extent\_fn, extra\_state, ierror)

 $^{24}$ 

Туре	Length	Optional Type
MPI_PACKED	1	MPI_INTEGER1
MPI_BYTE	1	MPI_INTEGER2
MPI_CHAR	1	MPI_INTEGER4
MPI_UNSIGNED_CHAR	1	MPI_INTEGER8
MPI_SIGNED_CHAR	1	 MPI_INTEGER16
MPI_WCHAR	2	
MPI_SHORT	2	MPI_REAL2
MPI_UNSIGNED_SHORT	2	MPI_REAL4
MPI_INT	4	MPI_REAL8
MPI_UNSIGNED	4	MPI_REAL16
MPI_LONG	4	
MPI_UNSIGNED_LONG	4	MPI_COMPLEX4
MPI_LONG_LONG_INT	8	MPI_COMPLEX8
MPI_UNSIGNED_LONG_LONG	8	MPI_COMPLEX16
MPI_FLOAT	4	MPI_COMPLEX32
MPI_DOUBLE	8	
MPI_LONG_DOUBLE	16	
MPI_C_BOOL	1	
MPI_INT8_T	1	C++ Types
		Cri Types
MPI_INT16_T	2	
MPI_INT32_T	4	MPI_CXX_BOOL
MPI_INT64_T	8	MPI_CXX_FLOAT_COMPLE
MPI_UINT8_T	1	MPI_CXX_DOUBLE_COMPL
MPI_UINT16_T	2	MPI_CXX_LONG_DOUBLE_
MPI_UINT32_T	4	
MPI_UINT64_T	8	
MPI_AINT	8	
MPI_COUNT	8	
MPI_OFFSET	8	
MPI_C_COMPLEX	2*4	
MPI_C_FLOAT_COMPLEX	2*4	
	∠*4 2*8	
MPI_C_DOUBLE_COMPLEX		
MPI_C_LONG_DOUBLE_COMPLEX	2*16	
MPI_CHARACTER	1	
MPI_LOGICAL	4	
MPI_INTEGER	4	
MPI_REAL	4	
MPI_DOUBLE_PRECISION	8	
MPI_COMPLEX	2*4	
MPI_DOUBLE_COMPLEX	2*8	
	1.0	
Table 13.2:	"external32	2" sizes of predefined datatypes

Optional Type	Length
MPI_INTEGER1	1
MPI_INTEGER2	2
MPI_INTEGER4	4
MPI_INTEGER8	8
MPI_INTEGER16	16
MPI_REAL2	2
MPI_REAL4	4
MPI_REAL8	8
MPI_REAL16	16
MPI_COMPLEX4	2*2
MPI_COMPLEX8	2*4
MPI_COMPLEX16	2*8
MPI_COMPLEX32	2*16

C++ Types Le	ength
MPI_CXX_BOOL	1
MPI_CXX_FLOAT_COMPLEX	2*4
MPI_CXX_DOUBLE_COMPLEX	2*8
MPI_CXX_LONG_DOUBLE_COMPLEX	2*16

```
CHARACTER(LEN=*), INTENT(IN) :: datarep
    PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
    PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
    PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
             DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
    CHARACTER*(*) DATAREP
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER IERROR
```

14The call associates read\_conversion\_fn, write\_conversion\_fn, and dtype\_file\_extent\_fn with the data representation identifier datarep. datarep can then be used as an argument to MPI\_FILE\_SET\_VIEW, causing subsequent data access operations to call the conversion functions to convert all data items accessed between file data representation and 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 20in the error class MPI\_ERR\_DUP\_DATAREP is raised using the default file error handler 21(see Section 13.7). The length of a data representation string is limited to the value of 22MPI\_MAX\_DATAREP\_STRING. MPI\_MAX\_DATAREP\_STRING must have a value of at least 64. 23No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

```
Extent Callback
```

```
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
             MPI_Aint *file_extent, void *extra_state);
```

```
ABSTRACT INTERFACE
```

```
SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
ierror)
    TYPE(MPI_Datatype) :: datatype
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
    INTEGER :: ierror
```

SUBROUTINE DATAREP\_EXTENT\_FUNCTION(DATATYPE, EXTENT, EXTRA\_STATE, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTENT, EXTRA\_STATE

The function dtype\_file\_extent\_fn must return, in file\_extent, the number of bytes required to store datatype in the file representation. The function is passed, in extra\_state, the argument that was passed to the MPI\_REGISTER\_DATAREP call. MPI will only call this routine with predefined datatypes employed by the user.

**Datarep Conversion Functions** 

```
typedef int MPI_Datarep_conversion_function(void *userbuf,
             MPI_Datatype datatype, int count, void *filebuf,
```

1

 $\mathbf{2}$ 

3

4

5 6

9

10

11

12

13

15

16

17

24

2526

27

28

29 30

31

32

33

34

35

36

37

38

39 40

41

42

43

44 45

46

1	MDI Offact position word toward state).
2	<pre>MPI_Offset position, void *extra_state);</pre>
3	ABSTRACT INTERFACE
4	SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
5	filebuf, position, extra_state, ierror)
6	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
7	TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype
8	INTEGER :: count, ierror
9	INTEGER(KIND=MPI_OFFSET_KIND) :: position
10 11	INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
12	
13	SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION, EXTRA_STATE, IERROR)
14	<pre><type> USERBUF(*), FILEBUF(*)</type></pre>
15	INTEGER COUNT, DATATYPE, IERROR
16	INTEGER(KIND=MPI_OFFSET_KIND) POSITION
17	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
18	The function read_conversion_fn must convert from file data representation to na-
19	tive representation. Before calling this routine, MPI allocates and fills filebuf with count
20 21	contiguous data items. The type of each data item matches the corresponding entry for the
21	predefined datatype in the type signature of datatype. The function is passed, in extra_state,
23	the argument that was passed to the $MPI\_REGISTER\_DATAREP$ call. The function must
24	copy all count data items from filebuf to userbuf in the distribution described by datatype,
25	converting each data item from file representation to native representation. datatype will be
26	equivalent to the datatype that the user passed to the read function. If the size of datatype is less than the size of the court data items, the court function must treat datatype
27	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
28	converted data at the location in userbuf specified by position into the (tiled) datatype.
29 30	
31	Advice to users. Although the conversion functions have similarities to MPI_PACK
32	and MPI_UNPACK, one should note the differences in the use of the arguments count
33	and position. In the conversion functions, count is a count of data items (i.e., count
34	of typemap entries of datatype), and position is an index into this typemap. In
35	MPI_PACK, incount refers to the number of whole datatypes, and position is a number of bytes. ( <i>End of advice to users.</i> )
36	or bytes. (1911 of unite to users.)
37	Advice to implementors. A converted read operation could be implemented as follows:
38 39	
40	1. Get file extent of all data items
41	2. Allocate a filebuf large enough to hold all count data items
42	3. Read data from file into filebuf
43	4. Call read_conversion_fn to convert data and place it into userbuf
44	5. Deallocate filebuf
45	
46 47	(End of advice to implementors.)
47	
-	

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call, and the userbuf pointer will be unchanged.

*Rationale.* Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion function can use the **position** to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

The function write\_conversion\_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the write function. The function is passed, in extra\_state, the argument that was passed to the MPI\_REGISTER\_DATAREP call.

The predefined constant MPI\_CONVERSION\_FN\_NULL may be used as either write\_conversion\_fn or read\_conversion\_fn. In that case, MPI will not attempt to invoke write\_conversion\_fn or read\_conversion\_fn, respectively, but will perform the requested data access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by using a filebuf large enough to hold all the requested data items or else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section (read\_conversion\_fn, write\_conversion\_fn, and dtype\_file\_extent\_fn) when one of the read or write routines in Section 13.4, or MPI\_FILE\_GET\_TYPE\_EXTENT is called by the user. dtype\_file\_extent\_fn will only be passed predefined datatypes employed by the user. The conversion functions will only be passed datatypes equivalent to those that the user has passed to one of the routines noted above.

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

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.

7 8 9

10

11

19

20

21

22 23

 $^{24}$ 

25

26

27

28 29

30

31 32

33

34

35 36

37 38

39

1

 $\mathbf{2}$ 

3

## 13.5.4 Matching Data Representations

It is the user's responsibility to ensure that the data representation used to read data from a file is *compatible* with the data representation that was used to write that data to the file.

In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible. Similarly, using different representation names on two different implementations may yield compatible representations.

Compatibility can be obtained when "external32" representation is used, although precision may be lost and the performance may be less than when "native" representation is used. Compatibility is guaranteed using "external32" provided at least one of the following conditions is met.

- The data access routines directly use types enumerated in Section 13.5.2, that are supported by all implementations participating in the I/O. The predefined type used to write a data item must also be used to read a data item.
- In the case of Fortran 90 programs, the programs participating in the data accesses obtain compatible datatypes using MPI routines that specify precision and/or range (Section 18.1.9).
- For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatibility with another implementation's "native" or "internal" representation.

Advice to users. Section 18.1.9 defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

## 13.6 Consistency and Semantics

13.6.1 File Consistency

40Consistency semantics define the outcome of multiple accesses to a single file. All file 41 accesses in MPI are relative to a specific file handle created from a collective open. MPI 42provides three levels of consistency: sequential consistency among all accesses using a single 43file handle, sequential consistency among all accesses using file handles created from a single 44collective open with atomic mode enabled, and user-imposed consistency among accesses 45other than the above. Sequential consistency means the behavior of a set of operations will 46be as if the operations were performed in some serial order consistent with program order; 47each access appears atomic, although the exact ordering of accesses is unspecified. User-48imposed consistency may be obtained using program order and calls to MPI\_FILE\_SYNC.

Let  $FH_1$  be the set of file handles created from one particular collective open of the file FOO, and  $FH_2$  be the set of file handles created from a different collective open of FOO. Note that nothing restrictive is said about  $FH_1$  and  $FH_2$ : the sizes of  $FH_1$  and  $FH_2$  may be different, the groups of processes used for each open may or may not intersect, the file handles in  $FH_1$  may be destroyed before those in  $FH_2$  are created, etc. Consider the following three cases: a single file handle (e.g.,  $fh_1 \in FH_1$ ), two file handles created from a single collective open (e.g.,  $fh_{1a} \in FH_1$  and  $fh_{1b} \in FH_1$ ), and two file handles from different collective opens (e.g.,  $fh_1 \in FH_1$  and  $fh_2 \in FH_2$ ).

For the purpose of consistency semantics, a matched pair (Section 13.4.5) of split collective data access operations (e.g., MPI\_FILE\_READ\_ALL\_BEGIN and MPI\_FILE\_READ\_ALL\_END) compose a single data access operation. Similarly, a nonblocking data access routine (e.g., MPI\_FILE\_IREAD) and the routine which completes the request (e.g., MPI\_WAIT) also compose a single data access operation. For all cases below, these data access operations are subject to the same constraints as blocking data access operations.

Advice to users. For an MPI\_FILE\_IREAD and MPI\_WAIT pair, the operation begins when MPI\_FILE\_IREAD is called and ends when MPI\_WAIT returns. (*End of advice to users.*)

Assume that  $A_1$  and  $A_2$  are two data access operations. Let  $D_1$  ( $D_2$ ) be the set of absolute byte displacements of every byte accessed in  $A_1$  ( $A_2$ ). The two data accesses *overlap* if  $D_1 \cap D_2 \neq \emptyset$ . The two data accesses *conflict* if they overlap and at least one is a write access.

Let  $SEQ_{fh}$  be a sequence of file operations on a single file handle, bracketed by MPI\_FILE\_SYNCs on that file handle. (Both opening and closing a file implicitly perform an MPI\_FILE\_SYNC.)  $SEQ_{fh}$  is a "write sequence" if any of the data access operations in the sequence are writes or if any of the file manipulation operations in the sequence change the state of the file (e.g., MPI\_FILE\_SET\_SIZE or MPI\_FILE\_PREALLOCATE). Given two sequences,  $SEQ_1$  and  $SEQ_2$ , we say they are not *concurrent* if one sequence is guaranteed to completely precede the other (temporally).

The requirements for guaranteeing sequential consistency among all accesses to a particular file are divided into the three cases given below. If any of these requirements are not met, then the value of all data in that file is implementation dependent.

Case 1:  $fh_1 \in FH_1$  All operations on  $fh_1$  are sequentially consistent if atomic mode is set. If nonatomic mode is set, then all operations on  $fh_1$  are sequentially consistent if they are either nonconcurrent, nonconflicting, or both.

Case 2:  $fh_{1a} \in FH_1$  and  $fh_{1b} \in FH_1$  Assume  $A_1$  is a data access operation using  $fh_{1a}$ , and  $A_2$  is a data access operation using  $fh_{1b}$ . If for any access  $A_1$ , there is no access  $A_2$ that conflicts with  $A_1$ , then MPI guarantees sequential consistency.

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.

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

 $\mathbf{6}$ 

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

 $45 \\ 46$ 

 $47 \\ 48$ 

1 2 3 4 5 6 7 8 9 10 11	semantics.			
12 13	MPI_FILE_	_SET_ATOMICITY(fh, flag)		
14	INOUT	fh	file handle (handle)	
15 16 17	IN	flag	true to set atomic mode, $false$ to set nonatomic mode (logical)	
18 19	int MPI_F	'ile_set_atomicity(MPI_Fi	le fh, int flag)	
MPI_File_set_atomicity(fh, flag, ierror) TYPE(MPI_File), INTENT(IN) :: fh LOGICAL, INTENT(IN) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG				
				28 29 30 31 32 33 34 35 36 37

Advice to implementors. Since the semantics guaranteed by atomic mode are stronger than those guaranteed by nonatomic mode, an implementation is free to adhere to the more stringent atomic mode semantics for outstanding requests. (End of advice to implementors.)

MPI_FILE_GET_ATOMICITY(fh, flag) <sup>1</sup>				
IN	fh	file handle (handle)	2	
OUT	flag	true if atomic mode, false if nonatomic mode (logical)	3	
001	Пад	true il atomic mode, laise il nonatomic mode (logical)	4	
int MDT T	Cilo mot otomioity (MDI Fil	a fh int uflam)	5 6	
IIIC MPI_F	Sile_get_atomicity(MPI_Fil	le in, inc *ilag)	7	
MPI_File_	get_atomicity(fh, flag, i	lerror)	8	
	<pre>MPI_File), INTENT(IN) ::</pre>	fh	9	
	CAL, INTENT(OUT) :: flag		10	
INTEC	ER, OPTIONAL, INTENT(OUT)	:: ierror	11	
MPI_FILE_	GET_ATOMICITY(FH, FLAG, 1	IERROR)	12	
	ER FH, IERROR		13	
LOGIC	CAL FLAG		14	
	THE CET ATOMICITY roturn	s the current consistency semantics for data access	15	
		ated by one collective open. If flag is true, atomic	16	
-	abled; if flag is false, nonatom		17	
mode is er			18	
			19	
MPI_FILE	_SYNC(fh)		20 21	
INOUT	fh	file handle (handle)	21	
			23	
int MPI_F	Sile_sync(MPI_File fh)		24	
NDT D:1.			25	
	<pre>sync(fh, ierror) (MPI_File), INTENT(IN) ::</pre>	fh	26	
	ER, OPTIONAL, INTENT(OUT)		27	
	LER, OF FIGNAL, INTENT (001)		28	
	SYNC(FH, IERROR)		29	
INTEC	ER FH, IERROR		30	
Callin	g MPI_FILE_SYNC with fh car	uses all previous writes to fh by the calling process	31	
	-	f other processes have made updates to the storage	32	
device, the	en all such updates become visi	ble to subsequent reads of <b>fh</b> by the calling process.	33	
MPI_FILE	SYNC may be necessary to e	ensure sequential consistency in certain cases (see	34 35	
above).			36	
	FILE_SYNC is a collective oper		37	
		that all nonblocking requests and split collective	38	
-	_	for calling $MPI_FILE_SYNC$ — otherwise, the call	39	
to MPI_FI	LE_SYNC is erroneous.		40	
1260 D		Files	41	
13.6.2 R	andom Access vs. Sequential	Files	42	
MPI distin	guishes ordinary random acce	ess files from sequential stream files, such as pipes	43	
-	-	nust be opened with the $MPI_MODE_SEQUENTIAL$	44	
		e only permitted data access operations are shared	45	
-		nd etypes with holes are erroneous. In addition, the	46	
		therefore, calls to MPI_FILE_SEEK_SHARED and	47 48	
MPI_FILE_GET_POSITION_SHARED are erroneous, and the pointer update rules specified <sup>48</sup>				

for the data access routines do not apply. The amount of data accessed by a data access operation will be the amount requested unless the end of file is reached or an error is raised.

*Rationale.* This implies that reading on a pipe will always wait until the requested amount of data is available or until the process writing to the pipe has issued an end of file. (*End of rationale.*)

Finally, for some sequential files, such as those corresponding to magnetic tapes or streaming network connections, writes to the file may be destructive. In other words, a write may act as a truncate (a MPI\_FILE\_SET\_SIZE with size set to the current position) followed by the write.

11 12 13

1

 $\mathbf{2}$ 

3 4

5 6

7

8

9

10

## 13.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

Nonblocking data access routines inherit the following progress rule from nonblocking point to point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

28 29

## 13.6.4 Collective File Operations

<sup>30</sup> Collective file operations are subject to the same restrictions as collective communication <sup>31</sup> operations. For a complete discussion, please refer to the semantics set forth in Section 5.13.

<sup>32</sup> Collective file operations are collective over a duplicate of the communicator used to <sup>33</sup> open the file — this duplicate communicator is implicitly specified via the file handle ar-<sup>34</sup> gument. Different processes can pass different values for other arguments of a collective <sup>35</sup> routine unless specified otherwise.

36 37

38

## 13.6.5 Nonblocking Collective File Operations

Nonblocking collective file operations are defined only for data access routines with explicit
 offsets and individual file pointers but not with shared file pointers.

<sup>41</sup> Nonblocking collective file operations are subject to the same restrictions as blocking <sup>42</sup> collective I/O operations. All processes belonging to the group of the communicator that <sup>43</sup> was used to open the file must call collective I/O operations (blocking and nonblocking) <sup>44</sup> in the same order. This is consistent with the ordering rules for collective operations in <sup>45</sup> threaded environments. For a complete discussion, please refer to the semantics set forth <sup>46</sup> in Section 5.13.

<sup>47</sup> Nonblocking collective I/O operations do not match with blocking collective I/O oper <sup>48</sup> ations. Multiple nonblocking collective I/O operations can be outstanding on a single file

handle. High quality MPI implementations should be able to support a large number of pending nonblocking I/O operations.

All nonblocking collective I/O calls are local and return immediately, irrespective of the status of other processes. The call initiates the operation which may progress independently of any communication, computation, or I/O. The call returns a request handle, which must be passed to a completion call. Input buffers should not be modified and output buffers should not be accessed before the completion call returns. The same progress rules described for nonblocking collective operations apply for nonblocking collective I/O operations. For a complete discussion, please refer to the semantics set forth in Section 5.12.

#### 13.6.6 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if etype is MPI\_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer.

Advice to users. In most cases, use of MPI\_BYTE as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

#### 13.6.7 Miscellaneous Clarifications

Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the comm and info used in an MPI\_FILE\_OPEN, or the etype and filetype used in an MPI\_FILE\_SET\_VIEW, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the etype and filetype must be committed before calling MPI\_FILE\_SET\_VIEW, and the datatype must be committed before calling MPI\_FILE\_READ or MPI\_FILE\_WRITE.

#### 13.6.8 MPI\_Offset Type

MPI\_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type MPI\_Offset.

In Fortran, the corresponding integer is an integer with kind parameter MPI\_OFFSET\_KIND, which is defined in the mpi\_f08 module, the mpi module and the mpif.h include file.

In Fortran 77 environments that do not support KIND parameters, MPI\_Offset arguments should be declared as an INTEGER of suitable size. The language interoperability implications for MPI\_Offset are similar to those for addresses (see Section 18.2).

2 3 4 5 6 7 8 9 10	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 <i>hints</i> specified via info when a file is created (see Section 13.2.8).
11	13.6.10 File Size
12 13 14 15 16	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 <i>size changing</i> 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.
17 18 19 20	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 <i>high byte</i> be the byte in that set with the largest displacement. The file size is the larger of
21	• One plus the displacement of the high byte.
22 23	$\bullet$ The size immediately after the size changing routine, or $MPI\_FILE\_OPEN,$ returned.
24 25 26 27 28 29	When applying consistency semantics, calls to MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and MPI_FILE_GET_SIZE is considered a read of the file (which overlaps with all accesses to the file).
30 31 32 33 34	Advice to users. Any sequence of operations containing the collective routines MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 13.6.1 are satisfied. (End of advice to users.)
35 36 37	File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.
38 39 40 41 42 43 44	Advice to users. Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an MPI_FILE_READ of 10 bytes and an MPI_FILE_SET_SIZE to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. (End of advice to users.)
45	13.6.11 Examples
46	The examples in this section illustrate the application of the MDI consistency and semantics

The examples in this section illustrate the application of the MPI consistency and semantics
 guarantees. These address

13.6.9

1

Logical vs. Physical File Layout

• conflicting accesses on file handles obtained from a single collective open, and	1
• all accesses on file handles obtained from two separate collective opens.	2 3
	4
The simplest way to achieve consistency for conflicting accesses is to obtain sequential	5
consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10	6
integers. If the latter, every element of b will be 5. If nonatomic mode is set, the results of	7
the read are undefined.	8
/* Process 0 */	9
int i, a[10];	10
int TRUE = 1;	11
	12
for ( i=0;i<10;i++)	13
a[i] = 5;	14
	15
MPI_File_open( MPI_COMM_WORLD, "workfile",	16
MPI_MODE_RDWR   MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );	17
<pre>MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );</pre>	18
MPI_File_set_atomicity( fh0, TRUE );	19
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);	20
/* MPI_Barrier( MPI_COMM_WORLD ); */	21
	22
/* Process 1 */	23
int b[10];	24
int TRUE = 1;	25
MPI_File_open( MPI_COMM_WORLD, "workfile",	26
MPI_MODE_RDWR   MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );	27
<pre>MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );</pre>	28
<pre>MPI_File_set_atomicity( fh1, TRUE );</pre>	29
/* MPI_Barrier( MPI_COMM_WORLD ); */	30
<pre>MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &amp;status);</pre>	31
A user may guarantee that the write on process 0 precedes the read on process 1 by imposing	32
temporal order with, for example, calls to MPI_BARRIER.	33
temporal order with; for example; early to fin 1_D/www.ew.	34
Advice to users. Routines other than MPI_BARRIER may be used to impose temporal	35 36
order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,	30
received by process 1 using MPI_RECV. (End of advice to users.)	38
	39
Alternatively, a user can impose consistency with nonatomic mode set:	40
/* Process 0 */	41
<pre>/* Process 0 */ int i, a[10];</pre>	42
for ( i=0;i<10;i++)	43
a[i] = 5;	44
~L-J ~,	45
MPI_File_open( MPI_COMM_WORLD, "workfile",	46

MPI\_File\_set\_view( fh0, 0, MPI\_INT, MPI\_INT, "native", MPI\_INFO\_NULL );

MPI\_MODE\_RDWR | MPI\_MODE\_CREATE, MPI\_INFO\_NULL, &fh0 );

```
1
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
\mathbf{2}
     MPI_File_sync( fh0 );
3
     MPI_Barrier( MPI_COMM_WORLD );
4
     MPI_File_sync( fh0 );
5
     /* Process 1 */
6
     int b[10];
7
     MPI_File_open( MPI_COMM_WORLD, "workfile",
8
                     MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
9
     MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
10
     MPI_File_sync( fh1 );
11
     MPI_Barrier( MPI_COMM_WORLD );
12
     MPI_File_sync( fh1 );
13
     MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status );
14
15
     The "sync-barrier-sync" construct is required because:
16
17
        • The barrier ensures that the write on process 0 occurs before the read on process 1.
18
        • The first sync guarantees that the data written by all processes is transferred to the
19
          storage device.
20
21
        • The second sync guarantees that all data which has been transferred to the storage
22
          device is visible to all processes. (This does not affect process 0 in this example.)
23
         The following program represents an erroneous attempt to achieve consistency by elim-
^{24}
     inating the apparently superfluous second "sync" call for each process.
25
26
     /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
27
     /* Process 0 */
28
     int i, a[10];
29
     for ( i=0;i<10;i++)</pre>
30
        a[i] = 5;
^{31}
32
     MPI_File_open( MPI_COMM_WORLD, "workfile",
33
                     MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
34
     MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
35
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
36
     MPI_File_sync( fh0 );
37
     MPI_Barrier( MPI_COMM_WORLD );
38
39
     /* Process 1 */
40
     int b[10];
^{41}
     MPI_File_open( MPI_COMM_WORLD, "workfile",
42
                     MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
43
     MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
44
     MPI_Barrier( MPI_COMM_WORLD );
45
     MPI_File_sync( fh1 );
46
     MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status );
47
48
     /* ----- THIS EXAMPLE IS ERRONEOUS ----- */
```

The above program also violates the MPI rule against out-of-order collective operations and will deadlock for implementations in which MPI\_FILE\_SYNC blocks.

Advice to users. Some implementations may choose to implement MPI\_FILE\_SYNC as a temporally synchronizing function. When using such an implementation, the "sync-barrier-sync" construct above can be replaced by a single "sync." The results of using such code with an implementation for which MPI\_FILE\_SYNC is not temporally synchronizing is undefined. (*End of advice to users.*)

#### Asynchronous I/O

The behavior of asynchronous I/O operations is determined by applying the rules specified above for synchronous I/O operations.

The following examples all access a preexisting file "myfile." Word 10 in myfile initially contains the integer 2. Each example writes and reads word 10.

First consider the following code fragment:

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```
int a = 4, b;
                                                                                  35
MPI_File_open( MPI_COMM_WORLD, "myfile",
                                                                                  36
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
                                                                                  37
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INF0_NULL );
                                                                                   38
/* MPI_File_set_atomicity( fh, TRUE ); Use this to set atomic mode. */
                                                                                   39
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
                                                                                   40
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
                                                                                   41
MPI_Wait(&reqs[0], &status);
                                                                                  42
MPI_Wait(&reqs[1], &status);
                                                                                   43
                                                                                  44
```

If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee sequential consistency in nonatomic mode.

On the other hand, the following code fragment:

 $\overline{7}$ 

 $^{24}$ 

```
1
     int a = 4, b;
\mathbf{2}
     MPI_File_open( MPI_COMM_WORLD, "myfile",
3
                      MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
4
     MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
\mathbf{5}
     MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
6
     MPI_Wait(&reqs[0], &status);
7
     MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
8
     MPI_Wait(&reqs[1], &status);
9
     defines the same ordering as:
10
11
     int a = 4, b;
12
     MPI_File_open( MPI_COMM_WORLD, "myfile",
13
                       MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
14
     MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
15
     MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status );
16
     MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status );
17
18
     Since
19
         • nonconcurrent operations on a single file handle are sequentially consistent, and
20
21
         • the program fragments specify an order for the operations,
22
23
     MPI guarantees that both program fragments will read the value 4 into b. There is no need
24
     to set atomic mode for this example.
25
          Similar considerations apply to conflicting accesses of the form:
26
     MPI_File_iwrite_all(fh,...);
27
     MPI_File_iread_all(fh,...);
28
     MPI_Waitall(...);
29
30
          In addition, as mentioned in Section 13.6.5, nonblocking collective I/O operations have
^{31}
     to be called in the same order on the file handle by all processes.
32
          Similar considerations apply to conflicting accesses of the form:
33
34
     MPI_File_write_all_begin(fh,...);
35
     MPI_File_iread(fh,...);
36
     MPI_Wait(fh,...);
37
     MPI_File_write_all_end(fh,...);
38
          Recall that constraints governing consistency and semantics are not relevant to the
39
     following:
40
41
     MPI_File_write_all_begin(fh,...);
42
     MPI_File_read_all_begin(fh,...);
43
     MPI_File_read_all_end(fh,...);
44
     MPI_File_write_all_end(fh,...);
45
46
     since split collective operations on the same file handle may not overlap (see Section 13.4.5).
47
48
```

## 13.7 I/O Error Handling

By default, communication errors are fatal — MPI\_ERRORS\_ARE\_FATAL is the default error handler associated with MPI\_COMM\_WORLD. I/O errors are usually less catastrophic (e.g., "file not found") than communication errors, and common practice is to catch these errors and continue executing. For this reason, MPI provides additional error facilities for I/O.

Advice to users. MPI does not specify the state of a computation after an erroneous MPI call has occurred. A high-quality implementation will support the I/O error handling facilities, allowing users to write programs using common practice for I/O. (*End of advice to users.*)

Like communicators, each file handle has an error handler associated with it. The MPI I/O error handling routines are defined in Section 8.3.

When MPI calls a user-defined error handler resulting from an error on a particular file handle, the first two arguments passed to the file error handler are the file handle and the error code. For I/O errors that are not associated with a valid file handle (e.g., in MPI\_FILE\_OPEN or MPI\_FILE\_DELETE), the first argument passed to the error handler is MPI\_FILE\_NULL.

I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI\_ERRORS\_RETURN. The default file error handler has two purposes: when a new file handle is created (by MPI\_FILE\_OPEN), the error handler for the new file handle is initially set to the default error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., MPI\_FILE\_OPEN or MPI\_FILE\_DELETE) use the default file error handler. The default file error handler can be changed by specifying MPI\_FILE\_NULL as the fh argument to MPI\_FILE\_SET\_ERRHANDLER. The current value of the default file error handler can be determined by passing MPI\_FILE\_NULL as the fh argument to MPI\_FILE\_GET\_ERRHANDLER.

*Rationale.* For communication, the default error handler is inherited from MPI\_COMM\_WORLD. In I/O, there is no analogous "root" file handle from which default properties can be inherited. Rather than invent a new global file handle, the default file error handler is manipulated as if it were attached to MPI\_FILE\_NULL. (*End of rationale.*)

## 13.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the error classes defined in Table 13.3.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI\_ERR\_TYPE.

## 13.9 Examples

13.9.1 Double Buffering with Split Collective I/O

This example shows how to overlap computation and output. The computation is performed by the function compute\_buffer().

#### Unofficial Draft for Comment Only

 $^{24}$ 

 $^{31}$ 

2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12	MPI_ERR_FILE	Invalid file handle
12	MPI_ERR_NOT_SAME	Collective argument not identical on all
14		processes, or collective routines called in
		a different order by different processes
15	MPI_ERR_AMODE	Error related to the amode passed to
16		MPI_FILE_OPEN
17	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
18		MPI_FILE_SET_VIEW
19	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
20		a file which supports sequential access only
21	MPI_ERR_NO_SUCH_FILE	File does not exist
22	MPI_ERR_FILE_EXISTS	File exists
23	MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
24	MPI_ERR_ACCESS	Permission denied
25	MPI_ERR_NO_SPACE	Not enough space
26	MPI_ERR_QUOTA	Quota exceeded
27	MPI_ERR_READ_ONLY	Read-only file or file system
28	MPI_ERR_FILE_IN_USE	File operation could not be completed, as
29		the file is currently open by some process
30	MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-
31		tered because a data representation identi-
32		fier that was already defined was passed to
33		MPI_REGISTER_DATAREP
34	MPI_ERR_CONVERSION	An error occurred in a user supplied data
35		conversion function.
36	MPI_ERR_IO	Other I/O error
37	T 11 10 6	
38	Table 13.3	3: I/O Error Classes
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		

```
1
2
*
* Function:
                                                                          3
                    double_buffer
                                                                          4
* Synopsis:
                                                                          5
                                                                         6
      void double_buffer(
*
                                                                         7
              MPI_File fh,
                                                    ** IN
*
                                                                         8
*
              MPI_Datatype buftype,
                                                    ** IN
                                                                         9
              int bufcount
                                                    ** IN
*
                                                                         10
*
       )
                                                                         11
*
* Description:
                                                                         12
       Performs the steps to overlap computation with a collective write
                                                                         13
*
       by using a double-buffering technique.
                                                                         14
*
                                                                         15
*
                                                                         16
* Parameters:
                                                                         17
      fh
                       previously opened MPI file handle
*
                                                                         18
       buftype
*
                       MPI datatype for memory layout
                                                                         19
                        (Assumes a compatible view has been set on fh)
*
*
       bufcount
                        # buftype elements to transfer
                                                                         20
                                                                        21
*-----*/
                                                                         22
                                                                         23
/* this macro switches which buffer "x" is pointing to */
                                                                         ^{24}
#define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
                                                                         25
                                                                         26
void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount)
{
                                                                         27
                                                                         28
                                                                         29
  MPI_Status status;
                          /* status for MPI calls */
  float *buffer1, *buffer2; /* buffers to hold results */
                                                                         30
                                                                         31
  float *compute_buf_ptr; /* destination buffer */
                           /* for computing */
                                                                         32
                                                                         33
  float *write_buf_ptr; /* source for writing */
                                                                         34
                          /* determines when to quit */
  int done;
                                                                         35
  /* buffer initialization */
                                                                         36
                                                                         37
  buffer1 = (float *)
                                                                         38
                    malloc(bufcount*sizeof(float));
                                                                         39
  buffer2 = (float *)
                    malloc(bufcount*sizeof(float));
                                                                         40
                                                                         41
  compute_buf_ptr = buffer1; /* initially point to buffer1 */
                                                                         42
  write_buf_ptr = buffer1; /* initially point to buffer1 */
                                                                         43
                                                                         44
                                                                         45
  /* DOUBLE-BUFFER prolog:
                                                                         46
   *
       compute buffer1; then initiate writing buffer1 to disk
                                                                         47
   */
                                                                         48
  compute_buffer(compute_buf_ptr, bufcount, &done);
```

560

```
1
         MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
\mathbf{2}
3
         /* DOUBLE-BUFFER steady state:
4
          *
             Overlap writing old results from buffer pointed to by write_buf_ptr
5
             with computing new results into buffer pointed to by compute_buf_ptr.
          *
6
          *
7
          *
             There is always one write-buffer and one compute-buffer in use
8
          *
             during steady state.
9
          */
10
         while (!done) {
11
            TOGGLE_PTR(compute_buf_ptr);
12
            compute_buffer(compute_buf_ptr, bufcount, &done);
13
            MPI_File_write_all_end(fh, write_buf_ptr, &status);
14
            TOGGLE_PTR(write_buf_ptr);
15
            MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
16
         }
17
18
         /* DOUBLE-BUFFER epilog:
19
              wait for final write to complete.
          *
20
          */
21
         MPI_File_write_all_end(fh, write_buf_ptr, &status);
22
23
^{24}
         /* buffer cleanup */
25
         free(buffer1);
26
         free(buffer2);
27
     }
28
29
     13.9.2 Subarray Filetype Constructor
30
^{31}
32
33
34
35
36
37
38
39
40
41
                                        Process 0
                                                         Process 2
42
                                        Process 1
                                                         Process 3
43
44
                              Figure 13.4: Example array file layout
45
46
          Assume we are writing out a 100x100 2D array of double precision floating point num-
47
     bers that is distributed among 4 processes such that each process has a block of 25 columns
48
```

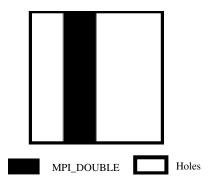


Figure 13.5: Example local array filetype for process 1

(e.g., process 0 has columns 0-24, process 1 has columns 25-49, etc.; see Figure 13.4). To create the filetypes for each process one could use the following C program (see Section 4.1.3):

```
17
double subarray[100][25];
                                                                                   18
MPI_Datatype filetype;
                                                                                   19
int sizes[2], subsizes[2], starts[2];
                                                                                   20
int rank;
                                                                                   21
                                                                                   22
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                                                                                   23
sizes[0]=100; sizes[1]=100;
                                                                                   ^{24}
subsizes[0]=100; subsizes[1]=25;
                                                                                   25
starts[0]=0; starts[1]=rank*subsizes[1];
                                                                                   26
                                                                                   27
MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                                                                                   28
                           MPI_DOUBLE, &filetype);
                                                                                   29
                                                                                   30
 Or, equivalently in Fortran:
                                                                                   ^{31}
                                                                                   32
    double precision subarray(100,25)
                                                                                   33
    integer filetype, rank, ierror
                                                                                   34
    integer sizes(2), subsizes(2), starts(2)
                                                                                   35
                                                                                   36
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
                                                                                   37
    sizes(1)=100
    sizes(2)=100
                                                                                   38
                                                                                   39
    subsizes(1)=100
    subsizes(2)=25
                                                                                   40
                                                                                   41
    starts(1)=0
                                                                                   42
    starts(2)=rank*subsizes(2)
                                                                                   43
                                                                                   44
    call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
                MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
                                                                                   45
                                                                    &
                                                                                   46
                filetype, ierror)
                                                                                   47
```

The generated filetype will then describe the portion of the file contained within the

## Unofficial Draft for Comment Only

9 10

11 12 13

14

15

16

1 2	process's subarray with holes for the space taken by the other processes. the filetype created for process 1.	Figure 13.5 shows
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		
31		
32		
33 34		
35		
36		
37		
38		
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		

## Chapter 14

# **Tool Support**

## 14.1 Introduction

This chapter discusses interfaces that allow debuggers, performance analyzers, and other tools to extract information about the operation of MPI processes. Specifically, this chapter defines both the MPI profiling interface (Section 14.2), which supports the transparent interception and inspection of MPI calls, and the MPI tool information interface (Section 14.3), which supports the inspection and manipulation of MPI control and performance variables. The interfaces described in this chapter are all defined in the context of an MPI process, i.e., are callable from the same code that invokes other MPI functions.

## 14.2 Profiling Interface

#### 14.2.1 Requirements

To meet the requirements for the  $\mathsf{MPI}$  profiling interface, an implementation of the  $\mathsf{MPI}$  functions must

1. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.4), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix PMPI\_ for each MPI function in each provided language binding and language support method. For routines implemented as macros, it is still required that the PMPI\_ version be supplied and work as expected, but it is not possible to replace at link time the MPI\_ version with a user-defined version.

For Fortran, the different support methods cause several specific procedure names. Therefore, several profiling routines (with these specific procedure names) are needed for each Fortran MPI routine, as described in Section 18.1.5.

- 2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.
- 3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economize by implementing it only for the lowest level routines.

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

14

15

1

2

3

4

5

6

7

8

9

10

13

14.2.2 Discussion

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

5. provide a no-op routine MPI\_PCONTROL in the MPI library.

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code that implements MPI on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

We believe that having such an interface is important if MPI is to be attractive to end users, since the availability of many different tools will be a significant factor in attracting users to the MPI standard.

The profiling interface is just that, an interface. It says *nothing* about the way in which it is used. There is therefore no attempt to lay down what information is collected through the interface, or how the collected information is saved, filtered, or displayed.

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as "internetworking" multiple MPI implementations. Since all that is defined is an interface, there is no objection to its being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which executable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the MPI profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this section is only present as justification and discussion of the logic for those requirements.

The examples below show one way in which an implementation could be constructed to meet the requirements on a Unix system (there are doubtless others that would be equally valid).

44

<sup>45</sup> 14.2.3 Logic of the Design

<sup>46</sup> Provided that an MPI implementation meets the requirements above, it is possible for <sup>47</sup> the implementor of the profiling system to intercept the MPI calls that are made by the user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

#### 14.2.4 Miscellaneous Control of Profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This capability is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at non-critical points in the calculation.
- Adding user events to a trace file.

These requirements are met by use of MPI\_PCONTROL.

```
MPI_PCONTROL(level, ...)
```

IN	level			Profiling level (integer)
int	MPI_Pcontrol(const	int	level,	)

MPI\_Pcontrol(level)
 INTEGER, INTENT(IN) :: level

```
MPI_PCONTROL(LEVEL)
INTEGER LEVEL
```

MPI libraries themselves make no use of this routine, and simply return immediately to the user code. However the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, we are unable to specify precisely the semantics that will be provided by calls to MPI\_PCONTROL. This vagueness extends to the number of arguments to the function, and their datatypes.

However to provide some level of portability of user codes to different profiling libraries, we request the following meanings for certain values of level.

level==0 Profiling is disabled.
 level==1 Profiling is enabled at a normal default level of detail.
 level==2 Profile buffers are flushed, which may be a no-op in some profilers.
 All other values of level have profile library defined effects and additional arguments.

We also request that the default state after MPI\_INIT has been called is for profiling to be enabled at the normal default level. (i.e., as if MPI\_PCONTROL had just been called with the argument 1). This allows users to link with a profiling library and to obtain profile output without having to modify their source code at all.

The provision of MPI\_PCONTROL as a no-op in the standard MPI library supports the collection of more detailed profiling information with source code that can still link against the standard MPI library. 48

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

## 14.2.5 Profiler Implementation Example

A profiler can accumulate the total amount of data sent by the MPI\_SEND function, along with the total elapsed time spent in the function as the following example shows:

```
\mathbf{5}
     Example 14.1
6
\overline{7}
     static int totalBytes = 0;
     static double totalTime = 0.0;
8
9
10
     int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
11
                    int dest, int tag, MPI_Comm comm)
12
     {
                                                 /* Pass on all arguments */
         double tstart = MPI_Wtime();
13
14
         int size;
15
                         = PMPI_Send(buffer,count,datatype,dest,tag,comm);
         int result
16
17
         totalTime += MPI_Wtime() - tstart;
                                                            /* and time
                                                                                    */
18
19
         MPI_Type_size(datatype, &size); /* Compute size */
20
         totalBytes += count*size;
21
22
         return result;
23
     }
^{24}
25
             MPI Library Implementation Example
     14.2.6
26
     If the MPI library is implemented in C on a Unix system, then there are various options,
27
     including the two presented here, for supporting the name-shift requirement. The choice
28
     between these two options depends partly on whether the linker and compiler support weak
29
     symbols.
30
^{31}
     Systems with Weak Symbols
32
33
     If the compiler and linker support weak external symbols (e.g., Solaris 2.x, other System
34
     V.4 machines), then only a single library is required as the following example shows:
35
36
     Example 14.2
37
     #pragma weak MPI_Example = PMPI_Example
38
39
     int PMPI_Example(/* appropriate args */)
40
     {
41
          /* Useful content */
42
     }
43
```

The effect of this **#pragma** is to define the external symbol MPI\_Example as a weak definition. This means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library); however if no other definition exists, then the linker will use the weak definition.

## Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

#### Systems Without Weak Symbols

In the absence of weak symbols then one possible solution would be to use the C macro preprocessor as the following example shows:

#### Example 14.3

```
#ifdef PROFILELIB
# ifdef __STDC__
# define FUNCTION(name) P##name
# else
# define FUNCTION(name) P/**/name
# endif
#else
# define FUNCTION(name) name
#endif
```

Each of the user visible functions in the library would then be declared thus

```
int FUNCTION(MPI_Example)(/* appropriate args */)
{
```

/\* Useful content \*/
}

The same source file can then be compiled to produce both versions of the library, depending on the state of the **PROFILELIB** macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions that she wishes to intercept, references to any others being fulfilled by the normal MPI library. Therefore the link step can look something like this

#### % cc ... -lmyprof -lpmpi -lmpi

Here libmyprof.a contains the profiler functions that intercept some of the MPI functions, libpmpi.a contains the "name shifted" MPI functions, and libmpi.a contains the normal definitions of the MPI functions.

#### 14.2.7 Complications

#### Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI func-tions (e.g., a portable implementation of the collective operations implemented using point to point communications), there is potential for profiling functions to be called from within an MPI function that was called from a profiling function. This could lead to "double counting" of the time spent in the inner routine. Since this effect could actually be useful under some circumstances (e.g., it might allow one to answer the question "How much time is spent in the point to point routines when they are called from collective functions?"), we have decided not to enforce any restrictions on the author of the MPI library that would

## Unofficial Draft for Comment Only

 $\frac{4}{5}$ 

 $^{24}$ 

overcome this. Therefore the author of the profiling library should be aware of this problem,
 and guard against it. In a single-threaded world this is easily achieved through use of a
 static variable in the profiling code that remembers if you are already inside a profiling
 routine. It becomes more complex in a multi-threaded environment (as does the meaning
 of the times recorded).

## <sup>7</sup> Linker Oddities

6

The Unix linker traditionally operates in one pass: the effect of this is that functions from libraries are only included in the image if they are needed at the time the library is scanned. When combined with weak symbols, or multiple definitions of the same function, this can cause odd (and unexpected) effects.

Consider, for instance, an implementation of MPI in which the Fortran binding is 13 achieved by using wrapper functions on top of the C implementation. The author of the 14profile library then assumes that it is reasonable only to provide profile functions for the C 15binding, since Fortran will eventually call these, and the cost of the wrappers is assumed 16to be small. However, if the wrapper functions are not in the profiling library, then none 17of the profiled entry points will be undefined when the profiling library is called. Therefore 18 none of the profiling code will be included in the image. When the standard MPI library 19is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of 20the MPI functions. The overall effect is that the code will link successfully, but will not be 21profiled. 22

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be copied out of the base library and into the profiling one using a tool such as **ar**.

27

## <sup>28</sup> Fortran Support Methods

The different Fortran support methods and possible options for the support of subarrays (depending on whether the compiler can support TYPE(\*), DIMENSION(..) choice buffers) imply different specific procedure names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 18.1.5.

 $\frac{34}{35}$ 

## 14.2.8 Multiple Levels of Interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation that would allow multiple levels of call interception, however we were unable to construct an implementation of this that did not have the following disadvantages

40 41

42

43

- assuming a particular implementation language,
- imposing a run time cost even when no profiling was taking place.

<sup>44</sup> Since one of the objectives of MPI is to permit efficient, low latency implementations, and <sup>45</sup> it is not the business of a standard to require a particular implementation language, we <sup>46</sup> decided to accept the scheme outlined above.

<sup>47</sup> Note, however, that it is possible to use the scheme above to implement a multi-level <sup>48</sup> system, since the function called by the user may call many different profiling functions

before calling the underlying MPI function. This capability has been demonstrated in the  $P^N$ MPI tool infrastructure [51].

## 14.3 The MPI Tool Information Interface

MPI implementations often use internal variables to control their operation and performance. Understanding and manipulating these variables can provide a more efficient execution environment or improve performance for many applications. This section describes the MPI tool information interface, which provides a mechanism for MPI implementors to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation. The interface is split into two parts: the first part provides information about and supports the setting of control variables through which the MPI implementation tunes its configuration. The second part provides access to performance variables that can provide insight into internal performance information of the MPI implementation.

To avoid restrictions on the MPI implementation, the MPI tool information interface allows the implementation to specify which control and performance variables exist. Additionally, the user of the MPI tool information interface can obtain metadata about each available variable, such as its datatype, and a textual description. The MPI tool information interface provides the necessary routines to find all variables that exist in a particular MPI implementation, to query their properties, to retrieve descriptions about their meaning, and to access and, if appropriate, to alter their values.

Variables and categories across connected processes with equivalent names are required to have the same meaning (see the definition of "equivalent" as related to strings in Section 14.3.3). Furthermore, enumerations with equivalent names across connected processes are required to have the same meaning, but are allowed to comprise different enumeration items. Enumeration items that have equivalent names across connected 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 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\_INIT (or equivalent) and after MPI\_FINALIZE. In order to support this behavior cleanly, the MPI tool information interface uses separate initialization and finalization routines. All identifiers used in the MPI tool information interface have the prefix MPI\_T\_.

On success, all MPI tool information interface routines return MPI\_SUCCESS, otherwise they return an appropriate and unique return code indicating the reason why the call was not successfully completed. Details on return codes can be found in Section 14.3.9. However, unsuccessful calls to the MPI tool information interface are not fatal and do not impact the execution of subsequent MPI routines.

#### 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 Since the MPI tool information interface primarily focuses on tools and support li- $\mathbf{2}$ braries, MPI implementations are only required to provide C bindings for functions and 3 constants introduced in this section. Except where otherwise noted, all conventions and 4 principles governing the C bindings of the MPI API also apply to the MPI tool information  $\mathbf{5}$ interface, which is available by including the mpi.h header file. All routines in this interface 6 have local semantics.

The number and type of control variables and performance variables Advice to users. 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 processes.

This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. When maximum portability is desired, application programmers should either avoid using the MPI tool information interface or avoid being dependent on the existence of a particular control or performance variable. (End of advice to users.)

14.3.1 Verbosity Levels

22 The MPI tool information interface provides access to internal configuration and performance information through a set of control and performance variables defined by the MPI 23 $^{24}$ implementation. Since some implementations may export a large number of variables, 25variables are classified by a verbosity level that categorizes both their intended audience 26(end users, performance tuners or MPI implementors) and a relative measure of level of detail (basic, detailed or all). These verbosity levels are described by a single integer. 27Table 14.1 lists the constants for all possible verbosity levels. The values of the con-28stants are monotonic in the order listed in the table; i.e., MPI\_T\_VERBOSITY\_USER\_BASIC 29 < MPI\_T\_VERBOSITY\_USER\_DETAIL < ... < MPI\_T\_VERBOSITY\_MPIDEV\_ALL. 30

<sup>32</sup> MPI_T_VERBOSITY_USER_BASIC Basic information of interest to users	
<sup>33</sup> MPI_T_VERBOSITY_USER_DETAIL Detailed information of interest to users	
<sup>34</sup> MPI_T_VERBOSITY_USER_ALL All remaining information of interest to us	ers
35 MPI_T_VERBOSITY_TUNER_BASIC Basic information required for tuning	
<sup>36</sup> MPI_T_VERBOSITY_TUNER_DETAIL Detailed information required for tuning	
37 MPI_T_VERBOSITY_TUNER_ALL All remaining information required for tur	ing
38 MPI_T_VERBOSITY_MPIDEV_BASIC Basic information for MPI implementors	
<sup>39</sup> MPI_T_VERBOSITY_MPIDEV_DETAIL Detailed information for MPI implementor	;
40 MPI_T_VERBOSITY_MPIDEV_ALL All remaining information for MPI implem	entors

Table 14.1: MPI tool information interface verbosity levels

#### Binding MPI Tool Information Interface Variables to MPI Objects 14.3.2

47Each MPI tool information interface variable provides access to a particular control setting or performance property of the MPI implementation. A variable may refer to a specific 48

7

8

9

10

11

12

13 14

15

16

17

18

1920

21

43 44 45

MPI object such as a communicator, datatype, or one-sided communication window, or the variable may refer more generally to the MPI environment of the process. Except for the last case, the variable must be bound to exactly one MPI object before it can be used. Table 14.2 lists all MPI object types to which an MPI tool information interface variable can be bound, together with the matching constant that MPI tool information interface routines return to identify the object type.

Constant	MPI object
MPI_T_BIND_NO_OBJECT	N/A; applies globally to entire MPI process
MPI_T_BIND_MPI_COMM	MPI communicators
MPI_T_BIND_MPI_DATATYPE	MPI datatypes
MPI_T_BIND_MPI_ERRHANDLER	MPI error handlers
MPI_T_BIND_MPI_FILE	MPI file handles
MPI_T_BIND_MPI_GROUP	MPI groups
MPI_T_BIND_MPI_OP	MPI reduction operators
MPI_T_BIND_MPI_REQUEST	MPI requests
MPI_T_BIND_MPI_WIN	MPI windows for one-sided communication
MPI_T_BIND_MPI_MESSAGE	MPI message object
MPI_T_BIND_MPI_INFO	MPI info object

#### Table 14.2: Constants to identify associations of variables

*Rationale.* Some variables have meanings tied to a specific MPI object. Examples include the number of send or receive operations that use a particular datatype, the number of times a particular error handler has been called, or the communication protocol and "eager limit" used for a particular communicator. Creating a new MPI tool information interface variable for each MPI object would cause the number of variables to grow without bound, since they cannot be reused to avoid naming conflicts. By associating MPI tool information interface variables with a specific MPI object, the MPI implementation only must specify and maintain a single variable, which can then be applied to as many MPI objects of the respective type as created during the program's execution. (*End of rationale.*)

#### 14.3.3 Convention for Returning Strings

Several MPI tool information interface functions return one or more strings. These functions have two arguments for each string to be returned: an OUT parameter that identifies a pointer to the buffer in which the string will be returned, and an IN/OUT parameter to pass the length of the buffer. The user is responsible for the memory allocation of the buffer and must pass the size of the buffer (n) as the length argument. Let n be the length value specified to the function. On return, the function writes at most n-1 of the string's characters into the buffer, followed by a null terminator. If the returned string's length is greater than or equal to n, the string will be truncated to n-1 characters. In this case, the length of the string plus one (for the terminating null character) is returned in the length argument. If the user passes the null pointer as the buffer argument or passes 0 as the length argument, the function does not return the string and only returns the length of the string plus one in the length argument. If the user passes the null pointer as the length argument, the buffer argument is ignored and nothing is returned.

#### Unofficial Draft for Comment Only

 $^{24}$ 

<sup>1</sup> MPI implementations behave as if they have an internal character array that is copied <sup>2</sup> to the output character array supplied by the user. Such output strings are only defined <sup>3</sup> to be equivalent if their notional source-internal character arrays are identical (up to and <sup>4</sup> including the null terminator), even if the output string is truncated due to a small input <sup>5</sup> length parameter n.

14.3.4 Initialization and Finalization

The MPI tool information interface requires a separate set of initialization and finalization routines.

10 11 12

13 14

6 7

8

9

MPI\_T\_INIT\_THREAD(required, provided)

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)

15 16 17

18

37

38 39

40

41

42

43

int MPI\_T\_init\_thread(int required, int \*provided)

All programs or tools that use the MPI tool information interface must initialize the 19MPI tool information interface in the processes that will use the interface before calling 20any other of its routines. A user can initialize the MPI tool information interface by calling 21MPI\_T\_INIT\_THREAD, which can be called multiple times. In addition, this routine initial-22izes the thread environment for all routines in the MPI tool information interface. Calling 23this routine when the MPI tool information interface is already initialized has no effect  $^{24}$ beyond increasing the reference count of how often the interface has been initialized. The 25argument required is used to specify the desired level of thread support. The possible values 26and their semantics are identical to the ones that can be used with MPI\_INIT\_THREAD 27listed in Section 12.4. The call returns in provided information about the actual level of 28thread support that will be provided by the MPI implementation for calls to MPI tool 29information interface routines. It can be one of the four values listed in Section 12.4. 30

The MPI specification does not require all MPI processes to exist before the call to MPI\_INIT. If the MPI tool information interface is used before MPI\_INIT has been called, the user is responsible for ensuring that the MPI tool information interface is initialized on all processes it is used in. Processes created by the MPI implementation during MPI\_INIT inherit the status of the MPI tool information interface (whether it is initialized or not as well as all active sessions and handles) from the process from which they are created.

Processes created at runtime as a result of calls to MPI's dynamic process management require their own initialization before they can use the MPI tool information interface.

Advice to users. If MPI\_T\_INIT\_THREAD is called before MPI\_INIT\_THREAD, the requested and granted thread level for MPI\_T\_INIT\_THREAD may influence the behavior and return value of MPI\_INIT\_THREAD. The same is true for the reverse order. (*End of advice to users.*)

Advice to implementors. MPI implementations should strive to make as many control
 or performance variables available before MPI\_INIT (instead of adding them within
 MPI\_INIT) to allow tools the most flexibility. In particular, control variables should
 be available before MPI\_INIT if their value cannot be changed after MPI\_INIT. (End
 of advice to implementors.)

#### MPI\_T\_FINALIZE()

#### int MPI\_T\_finalize(void)

This routine finalizes the use of the MPI tool information interface and may be called as often as the corresponding MPI\_T\_INIT\_THREAD routine up to the current point of execution. Calling it more times returns a corresponding error code. As long as the number of calls to MPI\_T\_FINALIZE is smaller than the number of calls to MPI\_T\_INIT\_THREAD up to the current point of execution, the MPI tool information interface remains initialized and calls to its routines are permissible. Further, additional calls to MPI\_T\_INIT\_THREAD after one or more calls to MPI\_T\_FINALIZE are permissible.

Once MPI\_T\_FINALIZE is called the same number of times as the routine MPI\_T\_INIT\_THREAD up to the current point of execution, the MPI tool information interface is no longer initialized. The interface can be reinitialized by subsequent calls to MPI\_T\_INIT\_THREAD.

At the end of the program execution, unless MPI\_ABORT is called, an application must have called MPI\_T\_INIT\_THREAD and MPI\_T\_FINALIZE an equal number of times.

#### 14.3.5 Datatype System

All variables managed through the MPI tool information interface represent their values through typed buffers of a given length and type using an MPI datatype (similar to regular send/receive buffers). Since the initialization of the MPI tool information interface is separate from the initialization of MPI, MPI tool information interface routines can be called before MPI\_INIT. Consequently, these routines can also use MPI datatypes before MPI\_INIT. Therefore, within the context of the MPI tool information interface, it is permissible to use a subset of MPI datatypes as specified below before a call to MPI\_INIT (or equivalent).

MPI_INT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_UNSIGNED_LONG_LONG
MPI_COUNT
MPI_CHAR
MPI_DOUBLE

Table 14.3: MPI datatypes that can be used by the MPI tool information interface

*Rationale.* The MPI tool information interface relies mainly on unsigned datatypes for integer values since most variables are expected to represent counters or resource sizes. MPI\_INT is provided for additional flexibility and is expected to be used mainly for control variables and enumeration types (see below).

Providing all basic datatypes, in particular providing all signed and unsigned variants of integer types, would lead to a larger number of types, which tools need to interpret. This would cause unnecessary complexity in the implementation of tools based on the MPI tool information interface. (*End of rationale.*)

 $\overline{7}$ 

 $^{24}$ 

The MPI tool information interface only relies on a subset of the basic MPI datatypes
 and does not use any derived MPI datatypes. Table 14.3 lists all MPI datatypes that can
 be returned by the MPI tool information interface to represent its variables.
 The use of the datatype MPI\_CHAR in the MPI tool information interface implies a null-

terminated character array, i.e., a string in the C language. If a variable has type MPI\_CHAR,
 the value of the count parameter returned by MPI\_T\_CVAR\_HANDLE\_ALLOC and

<sup>7</sup> MPI\_T\_PVAR\_HANDLE\_ALLOC must be large enough to include any valid value, including
 <sup>8</sup> its terminating null character. The contents of returned MPI\_CHAR arrays are only defined
 <sup>9</sup> from index 0 through the location of the first null character.

Rationale. The MPI tool information interface requires a significantly simpler type system than MPI itself. Therefore, only its required subset must be present before MPI\_INIT (or equivalent) and MPI implementations do not need to initialize the complete MPI datatype system. (*End of rationale.*)

For variables of type MPI\_INT, an MPI implementation can provide additional infor-16mation by associating names with a fixed number of values. We refer to this information 17in the following as an enumeration. In this case, the respective calls that provide addi-18 tional metadata for each control or performance variable, i.e., MPI\_T\_CVAR\_GET\_INFO 19 (Section 14.3.6) and MPI\_T\_PVAR\_GET\_INFO (Section 14.3.7), return a handle of type 20MPI\_T\_enum that can be passed to the following functions to extract additional informa-21tion. Thus, the MPI implementation can describe variables with a fixed set of values that 22each represents a particular state. Each enumeration type can have N different values, with 23a fixed N that can be queried using  $MPI_T_ENUM_GET_INFO$ .  $^{24}$ 

25 26

27

36

37

38

39

40

41

42

47 48

10

11

12

13

14 15

MPI\_T\_ENUM\_GET\_INFO(enumtype, num, name, name\_len)

28	IN	enumtype	enumeration to be queried (handle)
29 30 31	OUT	num	number of discrete values represented by this enumer- ation (integer)
31 32 33	OUT	name	buffer to return the string containing the name of the enumeration (string)
34 35	INOUT	name_len	length of the string and/or buffer for $name\xspace$ (integer)

### 

If enumtype is a valid enumeration, this routine returns the number of items represented by this enumeration type as well as its name. N must be greater than 0, i.e., the enumeration must represent at least one value.

The arguments name and name\_len are used to return the name of the enumeration as described in Section 14.3.3.

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.

		· · · · · · · · · · · · · · · · · · ·	
IN	enumtype	enumeration to be queried (handle)	2
	51		3
IN	index	number of the value to be queried in this enumeration	4
		(integer)	5
OUT	value	variable value (integer)	6
OUT	2222	buffer to return the string containing the name of the	7
001	name	buffer to return the string containing the name of the	8
		enumeration item (string)	9
INOUT	name_len	length of the string and/or buffer for name (integer)	10
inte e i	hame_ten	longen of the string and/of stater for heme (integer)	
			11

#### MPI\_T\_ENUM\_GET\_ITEM(enumtype, index, value, name, name\_len)

#### int MPI\_T\_enum\_get\_item(MPI\_T\_enum enumtype, int index, int \*value, char \*name, int \*name\_len)

The arguments name and name\_len are used to return the name of the enumeration item as described in Section 14.3.3.

If completed successfully, the routine returns the name/value pair that describes the enumeration at the specified index. The call is further required to return a name of at least length one. This name must be unique with respect to all other names of items for the same enumeration.

#### 14.3.6 Control Variables

The routines described in this section of the MPI tool information interface specification focus on the ability to list, query, and possibly set control variables exposed by the MPI implementation. These variables can typically be used by the user to fine tune properties and configuration settings of the MPI implementation. On many systems, such variables can be set using environment variables, although other configuration mechanisms may be available, such as configuration files or central configuration registries. A typical example that is available in several existing MPI implementations is the ability to specify an "eager limit," i.e., an upper bound on the size of messages sent or received using an eager protocol.

#### Control Variable Query Functions

An MPI implementation exports a set of N control variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any control variables, otherwise the provided control variables are indexed from 0 to N-1. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of control variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a control variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

Advice to users. While the MPI tool information interface guarantees that indices or variable properties do not change during a particular run of an MPI program, it does not provide a similar guarantee between runs. (End of advice to users.)

#### **Unofficial Draft for Comment Only**

1

12

13 14

15

16

17

18

19

2021

22 23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 32

33

34

35

36

37

38

39

40

41

42

43 44

45

The fe	ollowing function car	be used to query the number of control variables, <i>num_cvar</i> :
MPI_T_C	/AR_GET_NUM(nun	n_cvar)
OUT	num_cvar	returns number of control variables (integer)
int MPI_	[_cvar_get_num(in	t *num_cvar)
The f each varia		$AR\_GET\_INFO$ provides access to additional information for
MPI_T_C	/AR_GET_INFO(cva desc_len, bind,	r_index, name, name_len, verbosity, datatype, enumtype, desc, scope)
IN	cvar_index	index of the control variable to be queried, value be- tween 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 data type of the information stored in the control variable (handle)
OUT	enumtype	optional descriptor for enumeration information (han- dle)
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)
int MPI_7	int *verbosi	nt cvar_index, char *name, int *name_len, ty, MPI_Datatype *datatype, MPI_T_enum *enumtype, int *desc_len, int *bind, int *scope)
calls to th informatic If any tation will The a as describe If con	is routine that query on. An MPI implement of OUT parameter to a ignore the paramet arguments name and red in Section 14.3.3. Impleted successfully,	<pre>MPI_T_CVAR_GET_INFO for a particular variable, subsequent y information about the same variable must return the same entation is not allowed to alter any of the returned values. MPI_T_CVAR_GET_INFO is a NULL pointer, the implemen- er and not return a value for the parameter. name_len are used to return the name of the control variable the routine is required to return a name of at least length ue with respect to all other names for control variables used</pre>

 $^{\rm 48}$   $\,$  by the MPI implementation.

The argument verbosity returns the verbosity level of the variable (see Section 14.3.1).

The argument datatype returns the MPI datatype that is used to represent the control variable.

If the variable is of type MPI\_INT, MPI can optionally specify an enumeration for the values represented by this variable and return it in enumtype. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 14.3.5. Otherwise, enumtype is set to MPI\_T\_ENUM\_NULL. If the datatype is not MPI\_INT or the argument enumtype is the null pointer, no enumeration type is returned.

The arguments desc and desc\_len are used to return a description of the control variable as described in Section 14.3.3.

Returning a description is optional. If an MPI implementation does not return a description, the first character for desc must be set to the null character and desc\_len must be set to one at the return of this call.

The parameter bind returns the type of the MPI object to which the variable must be bound or the value MPI\_T\_BIND\_NO\_OBJECT (see Section 14.3.2).

The scope of a variable determines whether changing a variable's value is either local to the process or must be done by the user across multiple processes. The latter is further split into variables that require changes in a group of processes and those that require collective changes among all connected processes. Both cases can require all processes either to be set to consistent (but potentially different) values or to equal values on every participating process. The description provided with the variable must contain an explanation about the requirements and/or restrictions for setting the particular variable.

On successful return from MPI\_T\_CVAR\_GET\_INFO, the argument scope will be set to one of the constants listed in Table 14.4.

If the name of a control variable is equivalent across connected processes, the following OUT parameters must be identical: verbosity, datatype, enumtype, bind, and scope. The returned description must be equivalent.

Scope Constant	Description
MPI_T_SCOPE_CONSTANT	read-only, value is constant
MPI_T_SCOPE_READONLY	read-only, cannot be written, but can change
MPI_T_SCOPE_LOCAL	may be writeable, writing is a local operation
MPI_T_SCOPE_GROUP	may be writeable, must be done to a group of processes,
	all processes in a group must be set to consistent values
MPI_T_SCOPE_GROUP_EQ	may be writeable, must be done to a group of processes,
	all processes in a group must be set to the same value
MPI_T_SCOPE_ALL	may be writeable, must be done to all processes,
	all connected processes must be set to consistent values
MPI_T_SCOPE_ALL_EQ	may be writeable, must be done to all processes,
	all connected processes must be set to the same value

Table 14.4: Scopes for control variables

Advice to users. The scope of a variable only indicates if a variable might be changeable; it is not a guarantee that it can be changed at any time. (*End of advice to users.*)

#### **Unofficial Draft for Comment Only**

 $\mathbf{2}$ 

 $\mathbf{5}$ 

1 MPI\_T\_CVAR\_GET\_INDEX(name, cvar\_index) 2 IN name of the control variable (string) name 3 OUT cvar\_index index of the control variable (integer) 4 56 int MPI\_T\_cvar\_get\_index(const char \*name, int \*cvar\_index) 7 MPI\_T\_CVAR\_GET\_INDEX is a function for retrieving the index of a control variable 8 given a known variable name. The name parameter is provided by the caller, and cvar\_index 9 is returned by the MPI implementation. The **name** parameter is a string terminated with a 10 null character. 11 This routine returns MPI\_SUCCESS on success and returns MPI\_T\_ERR\_INVALID\_NAME 12if name does not match the name of any control variable provided by the implementation 13 at the time of the call. 14 15Rationale. This routine is provided to enable fast retrieval of control variables by 16a tool, assuming it knows the name of the variable for which it is looking. The 17 number of variables exposed by the implementation can change over time, so it is not 18 possible for the tool to simply iterate over the list of variables once at initialization. 19 Although using MPI implementation specific variable names is not portable across MPI 20implementations, tool developers may choose to take this route for lower overhead at 21runtime because the tool will not have to iterate over the entire set of variables to 22 find a specific one. (End of rationale.) 23 $^{24}$ Example: Printing All Control Variables 2526Example 14.4 27The following example shows how the MPI tool information interface can be used to 28query and to print the names of all available control variables. 29 30 #include <stdio.h> 31 #include <stdlib.h> 32 #include <mpi.h> 33 34 int main(int argc, char \*argv[]) { 35 int i, err, num, namelen, bind, verbose, scope; 36 int threadsupport; 37 char name[100]; 38 MPI\_Datatype datatype; 39 40err=MPI\_T\_init\_thread(MPI\_THREAD\_SINGLE,&threadsupport); 41 if (err!=MPI\_SUCCESS) 42return err; 43 44 err=MPI\_T\_cvar\_get\_num(&num); 45 if (err!=MPI\_SUCCESS) 46 return err; 47

Handle Allocation and Deallocation

Before reading or writing the value of a variable, a user must first allocate a handle of type MPI\_T\_cvar\_handle for the variable by binding it to an MPI object (see also Section 14.3.2).

*Rationale.* Handles used in the MPI tool information interface are distinct from handles used in the remaining parts of the MPI standard because they must be usable before MPI\_INIT and after MPI\_FINALIZE. Further, accessing handles, in particular for performance variables, can be time critical and having a separate handle space enables optimizations. (*End of rationale.*)

MPI_T_CVAR_HANDLE_ALLOC(cv	r_index, obj_handle,	handle, count)
----------------------------	----------------------	----------------

	( -	, 3 , , ,	30
IN	cvar_index	index of control variable for which handle is to be al- located (index)	30 31 32
IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)	32 33 34
OUT	handle	allocated handle (handle)	35
OUT	count	number of elements used to represent this variable (in- teger)	$\frac{36}{37}$
		leger)	38
· · MDT T			39

This routine binds the control variable specified by the argument index to an MPI object. <sup>42</sup> The object is passed in the argument obj\_handle as an address to a local variable that stores <sup>43</sup> the object's handle. The argument obj\_handle is ignored if the MPI\_T\_CVAR\_GET\_INFO <sup>44</sup> call for this control variable returned MPI\_T\_BIND\_NO\_OBJECT in the argument bind. The <sup>45</sup> 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 <sup>47</sup> MPI\_T\_CVAR\_GET\_INFO call) used to represent this variable. <sup>48</sup>

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $^{24}$ 

1 2 3	Advice to users. The count can be different based on the MPI object to which the control variable was bound. For example, variables bound to communicators could have a count that matches the size of the communicator.		
4 5 6 7 8 9	It is not portable to pass references to predefined MPI object handles, such as MPI_COMM_WORLD to this routine, since their implementation depends on the MPI library. Instead, such object handles should be stored in a local variable and the address of this local variable should be passed into MPI_T_CVAR_HANDLE_ALLOC. ( <i>End of advice to users.</i> )		
10 11 12 13 14 15	is the num MPI_T_C∖	ber of available con /AR_GET_NUM. Th	hould be in the range 0 to $num\_cvar - 1$ , where $num\_cvar$ ntrol variables as determined from a prior call to e type of the MPI object it references must be consistent bind argument in a prior call to MPI_T_CVAR_GET_INFO.
16	MPI_T_C\	/AR_HANDLE_FREE	(handle)
17 18	INOUT	handle	handle to be freed (handle)
19	int MPI_T	_cvar_handle_fre	e(MPI_T_cvar_handle *handle)
20 21 22 23 24	When a handle is no longer needed, a user of the MPI tool information interface should call MPI_T_CVAR_HANDLE_FREE to free the handle and the associated resources in the MPI implementation. On a successful return, MPI sets the handle to MPI_T_CVAR_HANDLE_NULL.		
25 26 27	Control Va	riable Access Functic	ns
28 29	MPI_T_C\	/AR_READ(handle, I	puf)
30	IN	handle	handle to the control variable to be read (handle)
31 32 33	OUT	buf	initial address of storage location for variable value (choice)
34 35	int MPI_T	_cvar_read(MPI_T	_cvar_handle handle, void* buf)
36 37 38 39 40 41 42	This routine queries the value of a control variable identified by the argument handle and stores the result in the buffer identified by the parameter buf. The user must ensure that the buffer is of the appropriate size to hold the entire value of the control variable (based on the returned datatype and count from prior corresponding calls to MPI_T_CVAR_GET_INFO and MPI_T_CVAR_HANDLE_ALLOC, respectively).		
43	MPI_T_C\	/AR_WRITE(handle,	buf)
44 45	IN	handle	handle to the control variable to be written (handle)
46 47 48	IN	buf	initial address of storage location for variable value (choice)

#### int MPI\_T\_cvar\_write(MPI\_T\_cvar\_handle handle, const void\* buf)

This routine sets the value of the control variable identified by the argument handle to the data stored in the buffer identified by the parameter buf. The user must ensure that the buffer is of the appropriate size to hold the entire value of the control variable (based on the returned datatype and count from prior corresponding calls to MPI\_T\_CVAR\_GET\_INFO and MPI\_T\_CVAR\_HANDLE\_ALLOC, respectively).

If the variable has a global scope (as returned by a prior corresponding MPI\_T\_CVAR\_GET\_INFO call), any write call to this variable must be issued by the user in all connected (as defined in Section 10.5.4) MPI processes. If the variable has group scope, any write call to this variable must be issued by the user in all MPI processes in the group, which must be described by the MPI implementation in the description by the MPI\_T\_CVAR\_GET\_INFO.

In both cases, the user must ensure that the writes in all processes are consistent. If the scope is either MPI\_T\_SCOPE\_ALL\_EQ or MPI\_T\_SCOPE\_GROUP\_EQ this means that the variable in all processes must be set to the same value.

If it is not possible to change the variable at the time the call is made, the function returns either MPI\_T\_ERR\_CVAR\_SET\_NOT\_NOW, if there may be a later time at which the variable could be set, or MPI\_T\_ERR\_CVAR\_SET\_NEVER, if the variable cannot be set for the remainder of the application's execution.

Example: Reading the Value of a Control Variable

## Example 14.5

return err;

}

The following example shows a routine that can be used to query the value with a control variable with a given index. The example assumes that the variable is intended to be bound to an MPI communicator.

```
int getValue_int_comm(int index, MPI_Comm comm, int *val) {
    int err,count;
    MPI_T_cvar_handle handle;
```

```
/* This example assumes that the variable index */
/* can be bound to a communicator */
```

```
err=MPI_T_cvar_handle_alloc(index,&comm,&handle,&count);
if (err!=MPI_SUCCESS) return err;
```

```
/* The following assumes that the variable is */
/* represented by a single integer */
```

```
err=MPI_T_cvar_read(handle,val);
if (err!=MPI_SUCCESS) return err;
err=MPI_T_cvar_handle_free(&handle);
```

 $\mathbf{2}$ 

# 14.3.7 Performance Variables

The following section focuses on the ability to list and to query performance variables provided by the MPI implementation. Performance variables provide insight into MPI implementation specific internals and can represent information such as the state of the MPI implementation (e.g., waiting blocked, receiving, not active), aggregated timing data for submodules, or queue sizes and lengths.

Rationale. The interface for performance variables is separate from the interface for control variables, since performance variables have different requirements and parameters. By keeping them separate, the interface provides cleaner semantics and allows for more performance optimization opportunities. (End of rationale.)

<sup>13</sup> <sub>14</sub> Performance Variable Classes

<sup>15</sup> Each performance variable is associated with a class that describes its basic semantics, <sup>16</sup> possible datatypes, basic behavior, its starting value, whether it can overflow, and when <sup>17</sup> and how an MPI implementation can change the variable's value. The starting value is the <sup>18</sup> 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 14.3.5. The starting value is the current state of the implementation at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

MPI\_T\_PVAR\_CLASS\_LEVEL

A performance variable in this class represents a value that describes the utilization level of a resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_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

# Unofficial Draft for Comment Only

of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current size of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

# • MPI\_T\_PVAR\_CLASS\_PERCENTAGE

The value of a performance variable in this class represents the percentage utilization of a finite resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. It will be returned as an MPI\_DOUBLE datatype. The value must always be between 0.0 (resource not used at all) and 1.0 (resource completely used). The starting value is the current percentage utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

## • MPI\_T\_PVAR\_CLASS\_HIGHWATERMARK

A performance variable in this class represents a value that describes the high watermark utilization of a resource. The value of a variable of this class is non-negative and grows monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utilization level of the resource at the time that the variable is started or reset. MPI implementations must ensure that variables of this class cannot overflow.

# • MPI\_T\_PVAR\_CLASS\_LOWWATERMARK

A performance variable in this class represents a value that describes the low watermark utilization of a resource. The value of a variable of this class is non-negative and decreases monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utilization level of the resource at the time that the variable is started or reset. MPI implementations must ensure that variables of this class cannot overflow.

## • MPI\_T\_PVAR\_CLASS\_COUNTER

A performance variable in this class counts the number of occurrences of a specific event (e.g., the number of memory allocations within an MPI library). The value of a variable of this class increases monotonically from the initialization or reset of the performance variable by one for each specific event that is observed. Values must be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG. The starting value for variables of this class is 0. Variables of this class can overflow.

## • MPI\_T\_PVAR\_CLASS\_AGGREGATE

The value of a performance variable in this class is an an aggregated value that represents a sum of arguments processed during a specific event (e.g., the amount of memory allocated by all memory allocations). This class is similar to the counter class, but instead of counting individual events, the value can be incremented by arbitrary amounts. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG,

## Unofficial Draft for Comment Only

7 8 9

10

11

12

13 14

15

16

17

18

19

1 2

3

4

5

6

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

584

1

2

3

4

5

6

7

MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value for variables of this class is 0. Variables of this class can overflow.

MPI\_T\_PVAR\_CLASS\_TIMER

The value of a performance variable in this class represents the aggregated time that the MPI implementation spends executing a particular event, type of event, or section of the MPI library. This class has the same basic semantics as

MPI\_T\_PVAR\_CLASS\_AGGREGATE, but explicitly records a timing value. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG,

- MPI\_DOUBLE. The starting value for variables of this class is 0. If the type MPI\_DOUBLE is used, the units that represent time in this datatype must match the units used by MPI\_WTIME. Otherwise, the time units should be documented, e.g., in the description returned by MPI\_T\_PVAR\_GET\_INFO. Variables of this class can overflow.
- 17 18

19

20

21

MPI\_T\_PVAR\_CLASS\_GENERIC

This class can be used to describe a variable that does not fit into any of the other classes. For variables in this class, the starting value is variable-specific and implementation-defined.

#### <sup>22</sup> <sub>23</sub> Performance Variable Query Functions

<sup>24</sup> An MPI implementation exports a set of N performance variables through the MPI tool <sup>25</sup> information interface. If N is zero, then the MPI implementation does not export any <sup>26</sup> performance variables; otherwise the provided performance variables are indexed from 0 to <sup>27</sup> N-1. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of performance variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a performance variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

- The following function can be used to query the number of performance variables, N:
- 34 35 36

37 38

39 40

41

42

43

MPI\_T\_PVAR\_GET\_NUM(num\_pvar)

OUT num\_pvar returns number of performance variables (integer)

```
int MPI_T_pvar_get_num(int *num_pvar)
```

The function MPI\_T\_PVAR\_GET\_INFO provides access to additional information for each variable.

- 44
- 45 46
- 46
- 47 48

MPI_T_PVAR_GET_INFO(pvar_index, name, name_len, verbosity, varclass, datatype, enumtype, desc, desc_len, bind, readonly, continuous, atomic)			
IN	pvar_index	index of the performance variable to be queried be- tween 0 and $num_pvar - 1$ (integer)	2 3 4
OUT	name	buffer to return the string containing the name of the performance variable (string)	5 6 7
INOUT	name_len	length of the string and/or buffer for name (integer)	8
OUT	verbosity	verbosity level of this variable (integer)	9 10
OUT	var_class	class of performance variable (integer)	11
OUT	datatype	MPI data type of the information stored in the performance variable (handle)	12 13
OUT	enumtype	optional descriptor for enumeration information (han- dle)	14 15 16
OUT	desc	buffer to return the string containing a description of the performance variable (string)	16 17 18
INOUT	desc_len	length of the string and/or buffer for desc (integer)	19
OUT	bind	type of MPI object to which this variable must be	20 21
0.UT		bound (integer)	22
OUT	readonly	flag indicating whether the variable can be written/reset (integer)	23 24
OUT	continuous	flag indicating whether the variable can be started and stopped or is continuously active (integer)	25 26
OUT	atomic	flag indicating whether the variable can be atomically read and reset (integer)	27 28 29

# int MPI\_T\_pvar\_get\_info(int pvar\_index, char \*name, int \*name\_len, int \*verbosity, int \*var\_class, MPI\_Datatype \*datatype, MPI\_T\_enum \*enumtype, char \*desc, int \*desc\_len, int \*bind, int \*readonly, int \*continuous, int \*atomic)

After a successful call to MPI\_T\_PVAR\_GET\_INFO for a particular variable, subsequent calls to this routine that query information about the same variable must return the same information. An MPI implementation is not allowed to alter any of the returned values.

If any OUT parameter to MPI\_T\_PVAR\_GET\_INFO is a NULL pointer, the implementation will ignore the parameter and not return a value for the parameter.

The arguments name and name\_len are used to return the name of the performance variable as described in Section 14.3.3. If completed successfully, the routine is required to return a name of at least length one.

The argument verbosity returns the verbosity level of the variable (see Section 14.3.1). The class of the performance variable is returned in the parameter var\_class. The class must be one of the constants defined in Section 14.3.7.

The combination of the name and the class of the performance variable must be unique with respect to all other names for performance variables used by the MPI implementation.

30

31

32

33

3435

36

37

38

39

40

41

42

4344

45

46

585

#### **Unofficial Draft for Comment Only**

Advice to implementors. Groups of variables that belong closely together, but have different classes, can have the same name. This choice is useful, e.g., to refer to multiple variables that describe a single resource (like the level, the total size, as well as high and low watermarks). (End of advice to implementors.)

The argument datatype returns the MPI datatype that is used to represent the perfor mance variable.

<sup>8</sup> If the variable is of type MPI\_INT, MPI can optionally specify an enumeration for the <sup>9</sup> values represented by this variable and return it in enumtype. In this case, MPI returns an <sup>10</sup> enumeration identifier, which can then be used to gather more information as described in <sup>11</sup> Section 14.3.5. Otherwise, enumtype is set to MPI\_T\_ENUM\_NULL. If the datatype is not <sup>12</sup> MPI\_INT or the argument enumtype is the null pointer, no enumeration type is returned.

Returning a description is optional. If an MPI implementation does not return a de scription, the first character for desc must be set to the null character and desc\_len must
 be set to one at the return from this function.

<sup>16</sup> The parameter bind returns the type of the MPI object to which the variable must be <sup>17</sup> bound or the value MPI\_T\_BIND\_NO\_OBJECT (see Section 14.3.2).

<sup>18</sup> Upon return, the argument readonly is set to zero if the variable can be written or reset <sup>19</sup> 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.

<sup>24</sup> Upon return, the argument **atomic** is set to zero if the variable cannot be read and <sup>25</sup> reset atomically. Only variables for which the call sets **atomic** to one can be used in a call <sup>26</sup> to MPI\_T\_PVAR\_READRESET.

If a performance variable has an equivalent name and has the same class across con nected processes, the following OUT parameters must be identical: verbosity, varclass,
 datatype, enumtype, bind, readonly, continuous, and atomic. The returned description must
 be equivalent.

31 32

MPI\_T\_PVAR\_GET\_INDEX(name, var\_class, pvar\_index)

IN	name	the name of the performance variable (string)
IN	var_class	the class of the performance variable (integer)
OUT	pvar_index	the index of the performance variable (integer)

37 38 39

40

41

42

43

int MPI\_T\_pvar\_get\_index(const char \*name, int var\_class, int \*pvar\_index)

MPI\_T\_PVAR\_GET\_INDEX is a function for retrieving the index of a performance variable given a known variable name and class. The name and var\_class parameters are provided by the caller, and pvar\_index is returned by the MPI implementation. The name parameter is a string terminated with a null character.

This routine returns MPI\_SUCCESS on success and returns MPI\_T\_ERR\_INVALID\_NAME if name does not match the name of any performance variable of the specified var\_class provided by the implementation at the time of the call.

48

1

2

3

4

*Rationale.* This routine is provided to enable fast retrieval of performance variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (*End of rationale.*)

#### Performance Experiment Sessions

Within a single program, multiple components can use the MPI tool information interface. To avoid collisions with respect to accesses to performance variables, users of the MPI tool information interface must first create a session. Subsequent calls that access performance variables can then be made within the context of this session. Any call executed in a session must not influence the results in any other session.

	VAR_SESSION_CRE	ATF(session)	18
			19
OUT	session	identifier of performance session (handle)	20
			21
int MPI_	T_pvar_session_cr	eate(MPI_T_pvar_session *session)	22
This	call creates a new sea	ssion for accessing performance variables and returns a handle	23
		nt session of type MPI_T_pvar_session.	24
101 11115 50	ssion in the argumer	In session of type with 1_1 _pval_session.	25
			26
MPI_T_P	VAR_SESSION_FREI	E(session)	27
			28
INOUT session		identifier of performance experiment session (handle)	29
			30
int MPI_	T_pvar_session_fr	ee(MPI_T_pvar_session *session)	31
This	call frees an existing	g session. Calls to the MPI tool information interface can no	32
	-	text of a session after it is freed. On a successful return, MPI	33
0		IPI_T_PVAR_SESSION_NULL.	34
5005 0110 50			35
Handlo All	location and Dealloca	ation	36
			37
Before usi	ing a performance v	ariable, a user must first allocate a handle of type	38
MPI_T_p	/ar_handle for the var	riable by binding it to an MPI object (see also Section 14.3.2).	39
			40
			41
			42
			43

**Unofficial Draft for Comment Only** 

 $\mathbf{5}$ 

MPI_T_P	VAR_HANDLE_ALLO	C(session, pvar_index, obj_handle, handle, count)		
IN	session	identifier of performance experiment session (handle)		
IN pvar_index index of performance variable for which hand be allocated (integer)				
IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)		
OUT	handle	allocated handle (handle)		
OUT	count	number of elements used to represent this variable (integer)		
int MPI_	-	c(MPI_T_pvar_session session, int pvar_index, dle, MPI_T_pvar_handle *handle, int *count)		
MPI object argument variable r reference contains	ct in the session ident obj_handle as an add obj_handle is ignored eturned MPI_T_BIND_ the variable is return the number of elemen	rformance variable specified by the argument index to an ified by the parameter session. The object is passed in the ress to a local variable that stores the object's handle. The if the MPI_T_PVAR_GET_INFO call for this performance NO_OBJECT in the argument bind. The handle allocated to ed in the argument handle. Upon successful return, count its (of the datatype returned by a previous used to represent this variable.		
perf	formance variable was	unt can be different based on the MPI object to which the bound. For example, variables bound to communicators natches the size of the communicator.		
MPI libra add	_COMM_WORLD, to the ary. Instead, such an	references to predefined MPI object handles, such as his routine, since their implementation depends on the MPI object handle should be stored in a local variable and the ble should be passed into MPI_T_PVAR_HANDLE_ALLOC.		
number of MPI_T_P with the f For a or INOUT	f available performan VAR_GET_NUM. The type returned in the b all routines in the res	be in the range 0 to $num_pvar - 1$ , where $num_pvar$ is the ce variables as determined from a prior call to e type of the MPI object it references must be consistent ind argument in a prior call to MPI_T_PVAR_GET_INFO. et of this section that take both handle and session as IN ndle argument passed in is not associated with the session _HANDLE is returned.		
MPI_T_P	VAR_HANDLE_FREE(	session, handle)		
IN	session	identifier of performance experiment session (handle)		
INOUT	handle	handle to be freed (handle)		
int MPI_	-	(MPI_T_pvar_session session, ndle *handle)		

### 14.3. THE MPI TOOL INFORMATION INTERFACE

When a handle is no longer needed, a user of the MPI tool information interface should call MPI\_T\_PVAR\_HANDLE\_FREE to free the handle in the session identified by the parameter session and the associated resources in the MPI implementation. On a successful return, MPI sets the handle to MPI\_T\_PVAR\_HANDLE\_NULL.

#### Starting and Stopping of Performance Variables

Performance variables that have the continuous flag set during the query operation are continuously operating once a handle has been allocated. Such variables may be queried at any time, but they cannot be started or stopped by the user. All other variables are in a stopped state after their handle has been allocated; their values are not updated until they have been started by the user.

#### MPI\_T\_PVAR\_START(session, handle)

IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)

#### int MPI\_T\_pvar\_start(MPI\_T\_pvar\_session session, MPI\_T\_pvar\_handle handle)

This functions starts the performance variable with the handle identified by the parameter handle in the session identified by the parameter session.

If the constant MPI\_T\_PVAR\_ALL\_HANDLES is passed in handle, the MPI implementation attempts to start all variables within the session identified by the parameter session for which handles have been allocated. In this case, the routine returns MPI\_SUCCESS if all variables are started successfully (even if there are no non-continuous variables to be started), otherwise MPI\_T\_ERR\_PVAR\_NO\_STARTSTOP is returned. Continuous variables and variables that are already started are ignored when MPI\_T\_PVAR\_ALL\_HANDLES is specified.

MPI\_T\_PVAR\_STOP(session, handle)

IN	session	identifier of performance experiment session (handle)
IN	handle	handle of a performance variable (handle)

#### int MPI\_T\_pvar\_stop(MPI\_T\_pvar\_session session, MPI\_T\_pvar\_handle handle)

This functions stops the performance variable with the handle identified by the parameter handle in the session identified by the parameter session.

If the constant MPI\_T\_PVAR\_ALL\_HANDLES is passed in handle, the MPI implementation attempts to stop all variables within the session identified by the parameter session for which handles have been allocated. In this case, the routine returns MPI\_SUCCESS if all variables are stopped successfully (even if there are no non-continuous variables to be stopped), otherwise MPI\_T\_ERR\_PVAR\_NO\_STARTSTOP is returned. Continuous variables and variables that are already stopped are ignored when MPI\_T\_PVAR\_ALL\_HANDLES is specified.

1

 $\mathbf{2}$ 

3

4

5 6

 $\overline{7}$ 

8

9

10

11

12 13 14

20

21

22

23

 $^{24}$ 

25

26

27

28 29 30

36

37

38

39

40

41

42

43

44

1	Performance Variable Access Functions					
2 3						
4	MPI_T_PVAR_READ(session, handle, buf)					
5	IN	session	identifier of performance experiment session (handle)			
6 7	IN	handle	handle of a performance variable (handle)			
8			-			
9	OUT	buf	initial address of storage location for variable value (choice)			
10 11						
12 13	int MPI_T	_pvar_read(MPI_T_pvar_ses void* buf)	ssion session, MPI_T_pvar_handle handle,			
14	The M	IPI T PVAR READ call queri	es the value of the performance variable with the			
15		-	the parameter session and stores the result in the			
16			The user is responsible to ensure that the buffer			
17	-		ntire value of the performance variable (based on			
18			e corresponding previous calls to			
19 20			VAR_HANDLE_ALLOC, respectively). IDLES cannot be used as an argument for the func-			
20		PVAR_READ.	VDLES cannot be used as an argument for the func-			
22						
23						
24	MPI_I_PV	AR_WRITE(session,handle, but	t)			
25	IN	session	identifier of performance experiment session (handle)			
26 27	IN	handle	handle of a performance variable (handle)			
28	IN	buf	initial address of storage location for variable value			
29			(choice)			
30						
31 32	int MPI_T	_pvar_write(MPI_T_pvar_se const void* buf)	ession session, MPI_T_pvar_handle handle,			
33	The M	PLT PVAR WRITE call atten	mpts to write the value of the performance variable			
34			ter handle in the session identified by the parameter			
35		о <u>т</u>	l in the buffer identified by the parameter <b>buf</b> . The			
36 37		—	appropriate size to hold the entire value of the per-			
38		( ° *	e and count returned by the corresponding previous			
39			IPI_T_PVAR_HANDLE_ALLOC, respectively).			
40			variable, the function returns			
41		R_PVAR_NO_WRITE.	JDLES cannot be used as an argument for the func-			
42		PVAR_WRITE.	The second of an argument for the func-			
43						
44						
45 46						
47						
48						

#### MPI\_T\_PVAR\_RESET(session, handle) IN session identifier of performance experiment session (handle) IN handle handle of a performance variable (handle) int MPI\_T\_pvar\_reset(MPI\_T\_pvar\_session session, MPI\_T\_pvar\_handle handle) The MPI\_T\_PVAR\_RESET call sets the performance variable with the handle identified by the parameter handle to its starting value specified in Section 14.3.7. If it is not possible to change the variable, the function returns MPI\_T\_ERR\_PVAR\_NO\_WRITE. If the constant MPI\_T\_PVAR\_ALL\_HANDLES is passed in handle, the MPI implementation attempts to reset all variables within the session identified by the parameter session for which handles have been allocated. In this case, the routine returns MPI\_SUCCESS if all variables are reset successfully (even if there are no valid handles or all are read-only), otherwise MPI\_T\_ERR\_PVAR\_NO\_WRITE is returned. Read-only variables are ignored when MPI\_T\_PVAR\_ALL\_HANDLES is specified. MPI\_T\_PVAR\_READRESET(session, handle, buf) IN session identifier of performance experiment session (handle) IN handle handle of a performance variable (handle) OUT buf initial address of storage location for variable value (choice)

# 

This call atomically combines the functionality of MPI\_T\_PVAR\_READ and MPI\_T\_PVAR\_RESET with the same semantics as if these two calls were called separately. If atomic operations on this variable are not supported, this routine returns MPI\_T\_ERR\_PVAR\_NO\_ATOMIC.

The constant MPI\_T\_PVAR\_ALL\_HANDLES cannot be used as an argument for the function MPI\_T\_PVAR\_READRESET.

Advice to implementors. Sampling-based tools rely on the ability to call the MPI tool information interface, in particular routines to start, stop, read, write, and reset performance variables, from any program context, including asynchronous contexts such as signal handlers. MPI implementations should strive, if possible in their particular environment, to enable these usage scenarios for all or a subset of the routines mentioned above. If implementing only a subset, the read, write, and reset routines are typically the most critical for sampling based tools. An MPI implementation should clearly document any restrictions on the program contexts in which the MPI tool information interface can be used. Restrictions might include guaranteeing usage outside of all signals or outside a specific set of signals. Any restrictions could be documented, for example, through the description returned by MPI\_T\_PVAR\_GET\_INFO. (*End of advice to implementors.*)

Rationale. All routines to read, to write or to reset performance variables require the session argument. This requirement keeps the interface consistent and allows the use 48

# Unofficial Draft for Comment Only

1 2

3

4 5

6 7

8

9

10

11

12

13

14

15

16 17 18

19

20

21

22 23

24 25

26

27

28

29

30

31

32

33 34

35

36

37

38

39

40

41

42

43

44

45

1

 $\mathbf{2}$ 

3 4

5 6

7

8

9

10

11

of MPI\_T\_PVAR\_ALL\_HANDLES where appropriate. Further, this opens up additional performance optimizations for the implementation of handles. (*End of rationale.*)

# Example: Tool to Detect Receives with Long Unexpected Message Queues

# Example 14.6

The following example shows a sample tool to identify receive operations that occur during times with long message queues. This examples assumes that the MPI implementation exports a variable with the name "MPI\_T\_UMQ\_LENGTH" to represent the current length of the unexpected message queue. The tool is implemented as a PMPI tool using the MPI profiling interface.

The tool consists of three parts: (1) the initialization (by intercepting the call to MPI\_INIT), (2) the test for long unexpected message queues (by intercepting calls to MPI\_RECV), and (3) the clean-up phase (by intercepting the call to MPI\_FINALIZE). To capture all receives, the example would have to be extended to have similar wrappers for

 $_{17}$  all receive operations.

Part 1— Initialization: During initialization, the tool searches for the variable and, once
 the right index is found, allocates a session and a handle for the variable with the found
 index, and starts the performance variable.

```
22
     #include <stdio.h>
23
     #include <stdlib.h>
24
     #include <string.h>
25
     #include <assert.h>
26
     #include <mpi.h>
27
28
     /* Global variables for the tool */
29
     static MPI_T_pvar_session session;
30
     static MPI_T_pvar_handle handle;
31
32
     int MPI_Init(int *argc, char ***argv ) {
33
            int err, num, i, index, namelen, verbosity;
34
            int var_class, bind, threadsup;
35
            int readonly, continuous, atomic, count;
36
            char name[18];
37
           MPI_Comm comm;
38
           MPI_Datatype datatype;
39
           MPI_T_enum enumtype;
40
41
           err=PMPI_Init(argc,argv);
42
            if (err!=MPI_SUCCESS) return err;
43
44
           err=PMPI_T_init_thread(MPI_THREAD_SINGLE,&threadsup);
45
            if (err!=MPI_SUCCESS) return err;
46
47
           err=PMPI_T_pvar_get_num(&num);
48
```

```
1
      if (err!=MPI_SUCCESS) return err;
                                                                                    2
      index=-1;
                                                                                    3
      i=0;
      while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {</pre>
                                                                                    4
            /* Pass a buffer that is at least one character longer than */
                                                                                    5
                                                                                    6
            /* the name of the variable being searched for to avoid */
                                                                                    7
            /* finding variables that have a name that has a prefix */
                                                                                     8
            /* equal to the name of the variable being searched. */
                                                                                    9
            namelen=18;
                                                                                    10
            err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
                                                                                    11
                     &var_class, &datatype, &enumtype, NULL, NULL, &bind,
                     &readonly, &continuous, &atomic);
                                                                                    12
            if (strcmp(name,"MPI_T_UMQ_LENGTH")==0) index=i;
                                                                                    13
                                                                                    14
            i++; }
                                                                                    15
      if (err!=MPI_SUCCESS) return err;
                                                                                    16
                                                                                    17
      /* this could be handled in a more flexible way for a generic tool */
                                                                                    18
      assert(index>=0);
                                                                                    19
      assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
      assert(datatype==MPI_INT);
                                                                                    20
                                                                                    21
      assert(bind==MPI_T_BIND_MPI_COMM);
                                                                                    22
                                                                                    23
      /* Create a session */
                                                                                    24
      err=PMPI_T_pvar_session_create(&session);
                                                                                    25
      if (err!=MPI_SUCCESS) return err;
                                                                                    26
      /* Get a handle and bind to MPI_COMM_WORLD */
                                                                                    27
      comm=MPI_COMM_WORLD;
                                                                                    28
                                                                                    29
      err=PMPI_T_pvar_handle_alloc(session, index, &comm, &handle, &count);
                                                                                    30
      if (err!=MPI_SUCCESS) return err;
                                                                                    31
                                                                                    32
      /* this could be handled in a more flexible way for a generic tool */
                                                                                    33
      assert(count==1);
                                                                                    34
      /* Start variable */
                                                                                    35
                                                                                    36
      err=PMPI_T_pvar_start(session, handle);
                                                                                    37
      if (err!=MPI_SUCCESS) return err;
                                                                                    38
                                                                                    39
      return MPI_SUCCESS;
                                                                                    40
                                                                                    41
                                                                                    42
Part 2 — Testing the Queue Lengths During Receives: During every receive operation, the
                                                                                    43
```

tool reads the unexpected queue length through the matching performance variable and compares it against a predefined threshold.

```
#define THRESHOLD 5
```

}

int MPI\_Recv(void \*buf, int count, MPI\_Datatype datatype, int source,

44

4546

```
1
                     int tag, MPI_Comm comm, MPI_Status *status)
2
      {
3
               int value, err;
4
5
               if (comm==MPI_COMM_WORLD) {
6
                        err=PMPI_T_pvar_read(session, handle, &value);
7
                        if ((err==MPI_SUCCESS) && (value>THRESHOLD))
8
                        {
9
                                  /* tool identified receive called with long UMQ */
10
                                  /* execute tool functionality, */
11
                                  /* e.g., gather and print call stack */
12
                        }
               }
13
14
15
               return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
16
      }
17
18
      Part 3 — Termination: In the wrapper for MPI_FINALIZE, the MPI tool information inter-
19
      face is finalized.
20
21
      int MPI_Finalize(void)
22
      {
23
               int err;
^{24}
               err=PMPI_T_pvar_handle_free(session, &handle);
25
               err=PMPI_T_pvar_session_free(&session);
26
               err=PMPI_T_finalize();
27
               return PMPI_Finalize();
28
      }
29
30
      14.3.8
             Variable Categorization
^{31}
      MPI implementations can optionally group performance and control variables into categories
32
      to express logical relationships between various variables. For example, an MPI implemen-
33
      tation could group all control and performance variables that refer to message transfers in
34
      the MPI implementation and thereby distinguish them from variables that refer to local
35
      resources such as memory allocations or other interactions with the operating system.
36
          Categories can also contain other categories to form a hierarchical grouping. Categories
37
      can never include themselves, either directly or transitively within other included categories.
38
      Expanding on the example above, this allows MPI to refine the grouping of variables referring
39
      to message transfers into variables to control and to monitor message queues, message
40
      matching activities and communication protocols. Each of these groups of variables would
41
      be represented by a separate category and these categories would then be listed in a single
42
     category representing variables for message transfers.
43
          The category information may be queried in a fashion similar to the mechanism for
44
      querying variable information. The MPI implementation exports a set of N categories via
45
      the MPI tool information interface. If N = 0, then the MPI implementation does not export
46
      any categories, otherwise the provided categories are indexed from 0 to N-1. This index
47
48
```

number is used in subsequent calls to functions of the MPI tool information interface t identify the individual categories.	$\begin{array}{c} 0 & 1 \\ & 2 \end{array}$				
An MPI implementation is permitted to increase the number of categories during the					
execution of an $MPI$ program when new categories become available through dynamic load-					
ng. However, MPI implementations are not allowed to change the index of a category or $5$					
delete it once it has been added to the set.	6				
Similarly, MPI implementations are allowed to add variables to categories, but the					
are not allowed to remove variables from categories or change the order in which they ar returned.	.е с 9				
The following function can be used to query the number of categories, $N$ .	10				
	11				
	12				
MPI_T_CATEGORY_GET_NUM(num_cat)	13				
OUT         num_cat         current number of categories (integer)	14				
	15 16				
<pre>int MPI_T_category_get_num(int *num_cat)</pre>	10				
Individual category information can then be queried by calling the following function	1: 18				
	19				
MPI_T_CATEGORY_GET_INFO(cat_index, name, name_len, desc, desc_len, num_cvars,	20				
num_pvars, num_categories)	21 22				
IN cat_index index of the category to be queried (integer)	23				
OUT     name     buffer to return the string containing the name of the category (string)	1e 24 25				
INOUT name_len length of the string and/or buffer for name (integer)	26 27				
OUT desc buffer to return the string containing the descriptio	21				
of the category (string)	29				
INOUT desc_len length of the string and/or buffer for desc (integer)	30 31				
OUT         num_cvars         number of control variables in the category (integer)					
OUT num_pvars number of performance variables in the category (in	1- 33				
$\operatorname{teger})$	34				
OUT num_categories number of categories contained in the category (inte	e- <sup>35</sup>				
$\operatorname{ger})$	36				
	37 38				
<pre>int MPI_T_category_get_info(int cat_index, char *name, int *name_len,</pre>	39				
<pre>char *desc, int *desc_len, int *num_cvars, int *num_pvars,</pre>	40				
<pre>int *num_categories)</pre>	41				
The arguments name and name_len are used to return the name of the category a	42 42				
described in Section 14.3.3.	43				
The routine is required to return a name of at least length one. This name must b					
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 a NULL pointer, the im-	45 )- 46				
lementation will ignore the parameter and not return a value for the parameter.					

```
48
```

1		_	nd desc_len are used to return the description of the category as			
2	described in Section 14.3.3.					
3 4	Returning a description is optional. If an MPI implementation decides not to return a description, the first character for desc must be set to the null character and desc_len must					
5	be set to one at the return of this call.					
6			he number of control variables, performance variables and other			
7	-		e queried category in the arguments num_cvars, num_pvars, and			
8	•	gories, respectivel				
9 10			gory is equivalent across connected processes, then the returned			
11	descriptio	n must be equiva	lent.			
12						
13	MPI_T_C	ATEGORY_GET_	INDEX(name, cat_index)			
14	IN	name	the name of the category (string)			
15	OUT	cat_index	the index of the category (integer)			
16		-				
17	int MPI	T categorv get	_index(const char *name, int *cat_index)			
18 19						
20			<b>GET_INDEX</b> is a function for retrieving the index of a category			
21	0	0.0	me. The name parameter is provided by the caller, and cat_index			
22	null chara		blementation. The name parameter is a string terminated with a			
23			IPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME			
24			e name of any category provided by the implementation at the			
25	time of th		e name of any category provided by the imprementation at the			
26						
27	Rationale. This routine is provided to enable fast retrieval of a category index					
28	by a tool, assuming it knows the name of the category for which it is looking. The					
29	number of categories exposed by the implementation can change over time, so it is not					
30	possible for the tool to simply iterate over the list of categories once at initialization.					
31 32	Although using MPI implementation specific category names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead					
33	at runtime because the tool will not have to iterate over the entire set of categories					
34	to find a specific one. ( <i>End of rationale.</i> )					
35	00 11	ind a specific one				
36						
37	MPLT C	ATEGORY GET	CVARS(cat_index, len, indices)			
38						
39	IN	cat_index	index of the category to be queried, in the range $[0, N-1]$ (integer)			
40						
41	IN	len	the length of the indices array (integer)			
42	OUT	indices	an integer array of size len, indicating control variable			
43 44			indices (array of integers)			
45						
46	int MPI_	T_category_get	_cvars(int cat_index, int len, int indices[])			
47	MPI	T CATEGORY (	GET_CVARS can be used to query which control variables are			
48	contained in a particular category. A category contains zero or more control variables.					

MPI_T_CATEGORY_GET_PVARS(cat_index, len, indices) <sup>1</sup>					
IN	cat_index	index of the category to be queried, in the range $\begin{bmatrix} 0, N- \\ 3 \end{bmatrix}$ (integer)			
IN	len	the length of the indices array (integer) 5			
OUT	indices	an integer array of size len, indicating performance variable indices (array of integers) 7			
int MPI_T	_category_get_pvars(int c	at_index, int len, int indices[]) 9			
		an be used to query which performance variables A category contains zero or more performance			
MPI_T_CA	TEGORY_GET_CATEGORIES	(cat_index, len, indices)			
IN	cat_index	index of the category to be queried, in the range $[0, N-1]$ 1] (integer)			
IN	len	the length of the indices array (integer)			
OUT	indices	an integer array of size len, indicating category indices (array of integers) 22 23 24 24 25 25 25 25 25 25 25 25 25 25 25 25 25			
int MPI_T	_category_get_categories(	int cat_index, int len, int indices[]) 22			
are contain As mer as the num of the MPI added or m virtual time	MPI_T_CATEGORY_GET_CATEGORIES can be used to query which other categories       25         are contained in a particular category. A category contains zero or more other categories.       26         As mentioned above, MPI implementations can grow the number of categories as well       27         as the number of variables or other categories within a category. In order to allow users       28         of the MPI tool information interface to check quickly whether new categories have been       29         added or new variables or categories have been added to a category, MPI maintains a       30         virtual timestamp. This timestamp is monotonically increasing during the execution and is       31         returned by the following function:       32				
MPI_T_CA	TEGORY_CHANGED(stamp)	34 34			
OUT	stamp	a virtual time stamp to indicate the last change to the categories (integer) 34			
int MPI_T	_category_changed(int *st	amp) 33			
If two subsequent calls to this routine return the same timestamp, it is guaranteed that the category information has not changed between the two calls. If the timestamp retrieved from the second call is higher, then some categories have been added or expanded.					
Advice to users. The timestamp value is purely virtual and only intended to check for changes in the category information. It should not be used for any other purpose. (End of advice to users.) 47					

# Unofficial Draft for Comment Only

1 The index values returned in indices by MPI\_T\_CATEGORY\_GET\_CVARS,  $\mathbf{2}$ MPI\_T\_CATEGORY\_GET\_PVARS and MPI\_T\_CATEGORY\_GET\_CATEGORIES can be used 3 as input to MPI\_T\_CVAR\_GET\_INFO, MPI\_T\_PVAR\_GET\_INFO and 4 MPI\_T\_CATEGORY\_GET\_INFO, respectively. 5The user is responsible for allocating the arrays passed into the functions 6 MPI\_T\_CATEGORY\_GET\_CVARS, MPI\_T\_CATEGORY\_GET\_PVARS and  $\overline{7}$ MPI\_T\_CATEGORY\_GET\_CATEGORIES. Starting from array index 0, each function writes 8 up to len elements into the array. If the category contains more than len elements, the 9 function returns an arbitrary subset of size len. Otherwise, the entire set of elements is 10 returned in the beginning entries of the array, and any remaining array entries are not 11modified. 1213Return Codes for the MPI Tool Information Interface 14.3.914All functions defined as part of the MPI tool information interface return an integer error 15code (see Table 14.5) to indicate whether the function was completed successfully or was 16aborted. In the latter case the error code indicates the reason for not completing the routine. 17Such errors neither impact the execution of the MPI process nor invoke MPI error handlers. 18 The MPI process continues executing regardless of the return code from the call. The MPI 19 implementation is not required to check all user-provided parameters; if a user passes invalid 20parameter values to any routine the behavior of the implementation is undefined. 21All error codes with the prefix MPI\_T\_ must be unique values and cannot overlap with 22 any other error codes or error classes returned by the MPI implementation. Further, they 23shall be treated as MPI error classes as defined in Section 8.4 and follow the same rules and  $^{24}$ restrictions. In particular, they must satisfy: 2526 $0 = MPI_SUCCESS < MPI_T_ERR_XXX \le MPI_ERR_LASTCODE.$ 2728 Rationale. All MPI tool information interface functions must return error classes, 29 because applications cannot portably call MPI\_ERROR\_CLASS before 30 MPI\_INIT or MPI\_INIT\_THREAD to map an arbitrary error code to an error class.  $^{31}$ (End of rationale.) 32 33 3414.3.10 Profiling Interface 35 36 All requirements for the profiling interfaces, as described in Section 14.2, also apply to 37 the MPI tool information interface. All rules, guidelines, and recommendations from Sec-38tion 14.2 apply equally to calls defined as part of the MPI tool information interface. 39 40 41 4243 444546

Return Code	Description
	Description
Return Codes for All Functions in t	
MPI_SUCCESS	Call completed successfully
MPI_T_ERR_INVALID	Invalid use of the interface or bad parameter
	values(s)
MPI_T_ERR_MEMORY	Out of memory
MPI_T_ERR_NOT_INITIALIZED	Interface not initialized
MPI_T_ERR_CANNOT_INIT	Interface not in the state to be initialized
Return Codes for Datatype Functio	
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)
Return Codes for Variable and Cate	egory Query Functions: MPI_T_*_GET_*
MPI_T_ERR_INVALID_INDEX	The variable or category index is invalid
MPI_T_ERR_INVALID_NAME	The variable or category name is invalid
Return Codes for Handle Functions	: MPI_T_*_{ALLOC FREE}
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 PVAR SESSION *
MPI_T_ERR_OUT_OF_SESSIONS	No more sessions available
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session
Return Codes for Control Variable	
MPI_T_CVAR_READ, WRITE	
MPI_T_ERR_CVAR_SET_NOT_NOW	Variable cannot be set at this moment
MPI_T_ERR_CVAR_SET_NEVER	Variable cannot be set until end of execution
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
Return Codes for Performance Vari	
MPI_T_PVAR_{START STOP REAL	
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session
MPI_T_ERR_PVAR_NO_STARTSTOP	Variable cannot be started or stopped
	(for MPI_T_PVAR_START and
	MPI_T_PVAR_STOP)
MPI_T_ERR_PVAR_NO_WRITE	Variable cannot be written or reset
	(for MPI_T_PVAR_WRITE and
	MPI_T_PVAR_RESET)
MPI_T_ERR_PVAR_NO_ATOMIC	Variable cannot be read and written atomically
	(for MPI_T_PVAR_READRESET)
Return Codes for Category Function	
	The category index is invalid
MPI_T_ERR_INVALID_INDEX	The category muex is invalid

Table 14.5: Return codes used in functions of the MPI tool information interface

# Chapter 15

# **Process Fault Tolerance**

#### 15.1Introduction

In distributed systems with numerous or complex components, a serious risk is that a component fault manifests as a process failure that disrupts the normal execution of a long running application. A process failure is a common outcome for many hardware, network, or software faults that cause a process to crash; it can be more formally defined as a fail-stop failure: the affected process stops communicating permanently. This chapter introduces MPI features that support the development of applications, libraries, and programming languages that can tolerate process failures. The primary goal is to specify error classes and interfaces that permit users to continue simple MPI communication operations after failures have impacted the execution and rebuild MPI objects (communicators, files, etc.) as needed to restore the full capability of MPI to carry out elaborate communication operations (like collective communications). This specification does not include mechanisms to restore the data lost due to process failures. The literature is rich with diverse fault tolerance techniques that the users may employ at their discretion, including checkpointrestart, algorithmic dataset recovery, and continuation ignoring failed processes. All these fault tolerance approaches benefit from, and often require, the definitions and interfaces specified in this chapter in order to resume communicating after a failure.

The expected behavior of MPI in the case of a process failure is defined by the following statements: any MPI operation that involves a failed process must not block indefinitely 34but either succeed or raise an MPI error (see Section 15.2); an MPI operation that does not involve a failed process will complete normally, unless interrupted by the user through provided functionality. Errors indicate only the local impact of the failure on an operation, 37 and make no guarantee that other processes have also been notified of the same failure. Asynchronous failure propagation is not guaranteed or required, and users must exercise caution when determining the set of processes where a failure has been detected and raised an error. If an application needs global knowledge of failures, it can use the interfaces defined in Section 15.3 to explicitly propagate the notification of locally detected failures.

The typical usage pattern on some reliable machines may not require fault tolerance. An MPI implementation that does not tolerate process failures must never raise a process failure error (as listed in Section 15.4). Fault-tolerant applications using the interfaces defined in this chapter must be portable across MPI implementations (including those which do not provide resilience, but in this case the interfaces may exhibit undefined behavior after a crash failure at any process.) Fault tolerant applications may determine if the

7 8 9

1516

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

35

36

38

39

40

41

42

43

44

45

46

47

implementation supports fault tolerance by querying the predefined attribute MPI\_FT on MPI\_COMM\_WORLD (see 8.1.2.)

Advice to users. Many of the operations and semantics described in this chapter are applicable only when the MPI application has replaced the default error handler MPI\_ERRORS\_ARE\_FATAL on, at least, MPI\_COMM\_WORLD. (*End of advice to users.*)

# 15.2 Failure Notification

This section specifies the behavior of an MPI communication operation when failures occur on processes involved in the communication. A process is considered involved in a communication (for the purpose of this chapter) if any of the following is true:

- The process is in the group over which the operation is collective.
- The process is a destination or a specified or matched source in a point-to-point communication.
- The operation is an MPI\_ANY\_SOURCE receive operation and the process belongs to the source group.
  - The process is a specified target in a remote memory operation.

An operation involving a failed process must always complete in a finite amount of time (possibly by raising one of the process failure error classes listed in Section 15.4). If an operation does not involve a failed process (such as a point-to-point message between two non-failed processes), it must not raise a process failure error.

26 27 28

29

30

31

32 33 Advice to implementors. An MPI implementation may provide failure detection only for processes involved in an ongoing operation and may postpone detection of other failures until necessary. Moreover, as long as an implementation can complete operations, it may choose to delay raising an error. Another valid implementation might choose to raise an error as quickly as possible. (*End of advice to implementors.*)

When a communication operation raises a process failure 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. Exceptions to this rule are explicitly stated in the remainder of this chapter. Error codes returned from a function, output in arrays of error codes, or in status objects remain defined after an operation raised a process failure error.

Non-blocking operations do not raise an error about process failures during creation or
 initiation. All process failure error raising is postponed until the corresponding completion
 function is called.

44 15.2.1 Startup and Finalize

<sup>45</sup> Initialization does not have any new semantics related to fault tolerance.

46

43

47 48

1

 $\mathbf{2}$ 

3 4

5

6 7 8

9 10

11

12

13 14

15

16

17 18

19

20 21

22 23

 $^{24}$ 

Advice to implementors. If a process fails during MPI\_INIT but its peers are able to complete the MPI\_INIT successfully, then a high quality implementation will return MPI\_SUCCESS and delay the reporting of the process failure to a subsequent MPI operation. (*End of advice to implementors.*)

MPI\_FINALIZE will complete even in the presence of process failures. If process 0 in MPI\_COMM\_WORLD has failed, it is possible that no process returns from MPI\_FINALIZE.

Advice to users. Fault tolerant applications are encouraged to implement all rankspecific code before the call to MPI\_FINALIZE. In Example 8.10 in Section 8.7, the process with rank 0 in MPI\_COMM\_WORLD may have failed before, during, or after the call to MPI\_FINALIZE, possibly leading to this code never being executed. (*End* of advice to users.)

# 15.2.2 Point-to-Point and Collective Communication

An MPI implementation raises errors of the following classes in order to notify users that a point-to-point communication operation could not complete successfully because of the failure of at least one involved process:

- MPI\_ERR\_PROC\_FAILED\_PENDING indicates, for a non-blocking communication, that the communication is a receive operation from MPI\_ANY\_SOURCE and no send operation has matched, yet a potential sending process has failed. Neither the operation nor the request identifying the operation is completed.
- In all other cases, the operation raises an error of class MPI\_ERR\_PROC\_FAILED to indicate that the failure prevents the operation from following its failure-free specification. If there is a request identifying a point-to-point communication, it is completed. Communication involving the failed process, initiated on this communicator after the error raised, must also raise an error of class MPI\_ERR\_PROC\_FAILED.

When a collective operation cannot be completed because of the failure of an involved process, the collective operation raises an error of class MPI\_ERR\_PROC\_FAILED.

## Advice to users.

Depending on how the collective operation is implemented and when a process failure occurs, some participating processes may raise an error while other processes return successfully from the same collective operation. For example, in MPI\_BCAST, the root process may succeed before a failed process disrupts the operation, resulting in some other processes raising an error.

(End of advice to users.)

## Advice to users.

Note that communicator creation functions (e.g., MPI\_COMM\_DUP or MPI\_COMM\_SPLIT) are collective operations. As such, if a failure happened during the call, an error might be raised at some processes while others succeed and obtain a new communicator handle. Although it is valid to communicate between processes that succeeded in creating the new communicator handle, the user is responsible for 

# Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11 12

13

14

15

16

ensuring a consistent view of the communicator creation, if needed. A conservative solution is to check the global outcome of the communicator creation function with MPI\_COMM\_AGREE (defined in Section 15.3.1), as illustrated in Example 15.1. (*End of advice to users.*)

After a process failure, MPI\_COMM\_FREE (as with all other collective operations) may not complete successfully at all processes. For any process that receives the return code MPI\_SUCCESS, the behavior is defined in Section 6.4.3. If a process raises a process failure error (classes MPI\_ERR\_PROC\_FAILED or MPI\_ERR\_REVOKED), the communicator handle comm is set to MPI\_COMM\_NULL; however, the implementation makes no guarantee about the success or failure of the MPI\_COMM\_FREE operation, locally or remotely.

- Advice to users. Users are encouraged to call MPI\_COMM\_FREE on communicators they do not wish to use anymore, even when they contain failed processes. Although the operation may raise a process failure error and not synchronize properly, this gives a high quality implementation an opportunity to release local resources and memory consumed by the object. (*End of advice to users.*)
- 17 18 19

20

21

22

23

24

25

26

27

28

29

30 31

32

33

15.2.3 Dynamic Process Management

Rationale. As with communicator creation functions, if a failure happens during a dynamic process management operation, an error might be raised at some processes while others succeed and obtain a new valid communicator. For most communicator creation functions, users can validate the success of the operation by communicating on a pre-existing communicator spanning over the same group of processes (in the worst case, from MPI\_COMM\_WORLD). This is however not always possible for dynamic process management operations, since these operations can create a new intercommunicator between previously disconnected processes. The following additional failure case semantics allow for users to validate, on the created intercommunicator itself, the success of the dynamic process management operation. (*End of rationale.*)

If the MPI implementation raises a process failure error at the root process in MPI\_COMM\_ACCEPT or MPI\_COMM\_CONNECT, the corresponding operation must also raise a process failure error at its root process.

34 35 36

37

38

43

Advice to users. The root process of an operation can succeed when a process failure error is raised at some other non-root process. (End of advice to users.)

When using the intercommunicator returned from MPI\_COMM\_SPAWN,

MPI\_COMM\_SPAWN\_MULTIPLE, or MPI\_COMM\_GET\_PARENT, a communication for which
 the root process of the spawn operation is the source or the destination must not deadlock.
 When the root process raises a process failure error from a spawn operation, no processes
 are spawned.

- Advice to implementors. An implementation is allowed to abort a spawned process
   during MPI\_INIT when it cannot setup an intercommunicator with the root process
   of the spawn operation because of a process failure.
- <sup>47</sup> An implementation may report it spawned all the requested processes even when a <sup>48</sup> process created from MPI\_COMM\_SPAWN or MPI\_COMM\_SPAWN\_MULTIPLE failed,

# Unofficial Draft for Comment Only

and instead delay raising a process failure error to a later communication involving this process. (*End of advice to implementors.*)

Advice to users. To determine how many new processes have effectively been spawned, the normal semantic for hard and soft spawn applies: if the requested number of processes is unavailable for a hard spawn, an error of class MPI\_ERR\_SPAWN is raised (possibly leaving MPI in an undefined state), and an appropriate error code is set in the array\_of\_errcodes parameter. Note however that an implementation may report that it has spawned the requested number of processes even when some processes have failed before exiting MPI\_INIT. This condition can be detected by communicating over the created intercommunicator with these processes.(*End of advice to users.*)

Advice to implementors. MPI\_COMM\_JOIN does not require any supplementary semantic. When the remote process on the fd socket has failed, the operation succeeds and sets intercomm to MPI\_COMM\_NULL. (*End of advice to implementors.*)

After a process failure, MPI\_COMM\_DISCONNECT (as with all other collective operations) may not complete successfully at all processes. For any process that receives the return code MPI\_SUCCESS, the behavior is defined in 10.5.4. If a process raises a process failure error (classes MPI\_ERR\_PROC\_FAILED or MPI\_ERR\_REVOKED), the communicator handle comm is set to MPI\_COMM\_NULL; however, the implementation makes no guarantee about the success or failure of the MPI\_COMM\_DISCONNECT operation, locally or remotely.

Advice to users. Users are encouraged to call MPI\_COMM\_DISCONNECT on communicators they do not wish to use anymore, even when they contain failed processes. Although the operation may raise a process failure error and not synchronize properly, this gives a high quality implementation an opportunity to release local resources and memory consumed by the object. (*End of advice to users.*)

# 15.2.4 One-Sided Communication

When an operation on a window raises an error related to process failure, the state of all data held in memory exposed by that window becomes undefined at all processes for which a one-sided communication operation could have modified local data (a target in a remote write, or accumulate operation, or an origin in a remote read operation), and the operation completion has not been semantically guaranteed (as an example by a successful synchronization between the origin and the target, after the origin had issued an MPI\_WIN\_FLUSH).

Advice to users. Assessing if a particular portion of the exposed memory remains correct is the responsibility of the user. Note that in passive target mode, when an error is raised at the origin, the target memory may become undefined before a synchronization raises an error at the target.

The exposed memory becomes undefined for all uses, not only the window in which the error was raised. Any overlapping windows or uses involving shared memory are also undefined (even if they do not involve MPI calls). (*End of advice to users.*)

Advice to implementors. A high quality implementation should limit the scope of the exposed memory that becomes undefined (for example, only the memory addresses

# 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

and ranges that have been targeted by a remote write, or accumulate, or have been an origin in a remote read). In that case, we encourage implementations to document the provided behavior, and to expose the availability of this feature at runtime, as an example by caching an implementation specific attribute on the window. (*End of advice to implementors.*)

Non-synchronizing one-sided communication operations (as an example MPI\_GET, MPI\_PUT) whose outputs are undefined, due to a process failure, are not required to raise a process failure error. However, if a communication cannot complete correctly due to process failures, the synchronization operation must raise a process failure error at least at the origin.

Advice to implementors. Non-synchronizing operations (MPI\_WIN\_FLUSH\_LOCAL, MPI\_WIN\_FLUSH\_LOCAL\_ALL) are not required to raise a process failure error. (End of advice to implementors.)

Advice to users. As with collective operations over MPI communicators, active target one-sided synchronization operations may raise a process failure error at some process while the corresponding operation returned MPI\_SUCCESS at some other process. (End of advice to users.)

Passive target synchronization operations may raise a process failure error when any process in the window has failed (even when the target specified in the argument of the passive target synchronization has not failed).

*Rationale.* An implementation of passive target synchronization may need to communicate with non-target processes in the window, as an example, a previous owner of an access epoch on the target window. (*End of rationale.*)

After a process failure, MPI\_WIN\_FREE (as with all other collective operations) may not complete successfully at all processes. For any process that receives the return code MPI\_SUCCESS, the behavior is defined in Section 11.2.5. If a process raises a process failure error (classes MPI\_ERR\_PROC\_FAILED or MPI\_ERR\_REVOKED), the window handle win is set to MPI\_WIN\_NULL; however, the implementation makes no guarantee about the success or failure of the MPI\_WIN\_FREE operation, locally or remotely.

Advice to users. Users are encouraged to call MPI\_WIN\_FREE on windows they do not wish to use anymore, even when they contain failed processes. Although the operation may raise a process failure error and not synchronize properly, this gives a high quality implementation an opportunity to release local resources and memory consumed by the object. Before calling MPI\_WIN\_FREE, it may be required to call MPI\_WIN\_REVOKE to close an epoch that couldn't be completed as a consequence of a process failure (see Section 15.3.2). (End of advice to users.)

<sup>44</sup><sub>45</sub> 15.2.5 I/O

This section defines the behavior of I/O operations when MPI process failures prevent their
 successful completion. I/O backend failure error classes and their consequences are defined
 in Section 13.7.

 $^{31}$ 

If a process failure prevents a file operation from completing, an MPI error of class MPI\_ERR\_PROC\_FAILED is raised. Once an MPI implementation has raised an error of class MPI\_ERR\_PROC\_FAILED, the state of the file pointers involved in the operation that raised the error is *undefined*.

Advice to users. Since collective I/O operations may not synchronize with other processes, process failures may not be reported during a collective I/O operation. Users are encouraged to use MPI\_COMM\_AGREE on a communicator containing the same group as the file handle when they need to deduce the completion status of collective operations on file handles and maintain a consistent view of file pointers. The file pointer can be reset by using MPI\_FILE\_SEEK with the MPI\_SEEK\_SET update mode. (*End of advice to users.*)

After a process failure, MPI\_FILE\_CLOSE (as with all other collective operations) may not complete successfully at all processes. For any process that receives the return code MPI\_SUCCESS, the behavior is defined in Section 13.2.2. If a process raises a process failure error (classes MPI\_ERR\_PROC\_FAILED or MPI\_ERR\_REVOKED), the file handle fh is set to MPI\_FILE\_NULL; however, the implementation makes no guarantee about the success or failure of the MPI\_FILE\_CLOSE operation, locally or remotely.

Advice to users. Users are encouraged to call MPI\_FILE\_CLOSE on files they do not wish to use anymore, even when they contain failed processes. Although the operation may raise a process failure error and not synchronize properly, this gives a high quality implementation an opportunity to release local resources and memory consumed by the object. (End of advice to users.)

# 15.3 Failure Mitigation Functions

# 15.3.1 Communicator Functions

Process failure notification is not global in MPI. Processes that do not call operations involving a failed process are possibly never notified of its failure (see Section 15.2). If a notification must be propagated, MPI provides a function to revoke a communicator at all members.

MPI\_COMM\_REVOKE( comm )

```
IN comm communicator (handle)
int MPI_Comm_revoke(MPI_Comm comm)
MPI_Comm_revoke(comm, ierror)
   TYPE(MPI_Comm), INTENT(IN) :: comm
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_COMM_REVOKE(COMM, IERROR)
   INTEGER COMM, IERROR
```

 $\mathbf{2}$ 

 $\overline{7}$ 

1 This function notifies all processes in the groups (local and remote) associated with  $\mathbf{2}$ the communicator comm that this communicator is revoked. The revocation of a commu-3 nicator by any process completes non-local MPI operations on comm at all processes by 4 raising an error of class MPI\_ERR\_REVOKED (with the exception of MPI\_COMM\_SHRINK,  $\mathbf{5}$ MPI\_COMM\_AGREE, and MPI\_COMM\_IAGREE). This function is not collective and there-6 fore does not have a matching call on remote processes. All non-failed processes belonging 7to comm will be notified of the revocation despite failures. 8 A communicator is revoked at a given process either when MPI\_COMM\_REVOKE is

locally called on it, or when any MPI operation on comm raises an error of class
 MPI\_ERR\_REVOKED at that process. Once a communicator has been revoked at a process, all
 subsequent non-local operations on that communicator (with the same exceptions as above),
 are considered local and must complete by raising an error of class MPI\_ERR\_REVOKED at
 that process.

14 15 16

26

27

28

43

44

48

MPI\_COMM\_SHRINK( comm, newcomm )

17	IN	comm			communicate	or (handle)
18	OUT	newcomm			communicate	or (handle)
19						()
20 21	int MPI C	omm_shrink	(MPI Comm	comm.	MPI Comm*	newcomm)
21				-		,
22		shrink(com	•			
23		MPI_Comm),				
25	TYPE(	MPI_Comm),	INTENT(OU	•		~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI\_COMM\_SHRINK(COMM, NEWCOMM, IERROR)

INTEGER COMM, NEWCOMM, IERROR

<sup>29</sup> This collective operation creates a new intra- or intercommunicator

<sup>30</sup> newcomm from the intra- or intercommunicator comm, respectively, by excluding the group
 <sup>31</sup> of failed processes as agreed upon during the operation. The groups of newcomm must
 <sup>32</sup> include every process that returns from MPI\_COMM\_SHRINK, and it must exclude every
 <sup>33</sup> process whose failure caused an operation on comm to raise an MPI error of class

<sup>34</sup> MPI\_ERR\_PROC\_FAILED or MPI\_ERR\_PROC\_FAILED\_PENDING at a member of the groups of <sup>35</sup> newcomm, before that member initiated MPI\_COMM\_SHRINK. This call is semantically <sup>36</sup> equivalent to an MPI\_COMM\_SPLIT operation that would succeed despite failures, where <sup>37</sup> members of the groups of newcomm participate with the same color and a key equal to their <sup>38</sup> rank in comm.

<sup>39</sup> This function never raises an error of class MPI\_ERR\_PROC\_FAILED or

<sup>40</sup> MPI\_ERR\_REVOKED. The defined semantic of MPI\_COMM\_SHRINK is maintained when <sup>41</sup> comm is revoked, or when the group of comm contains failed processes.

*Advice to users.* MPI\_COMM\_SHRINK is a collective operation, even when comm is revoked.

<sup>45</sup> <sup>46</sup> <sup>47</sup> The group of **newcomm** may still contain failed processes, whose failure will be detected <sup>47</sup> in subsequent MPI operations. (*End of advice to users.*)

MPI_COM	1M_FAILURE_ACK( comm )		1		
IN	comm	communicator (handle)	2 3		
			4		
int MPI_	Comm_failure_ack(MPI_Comm	n comm)	5		
MPI_Comm_failure_ack(comm, ierror)			6		
	(MPI_Comm), INTENT(IN) ::		7		
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	8		
MPI_COMM	_FAILURE_ACK(COMM, IERROR	.)	9 10		
	GER COMM, IERROR		11		
This	local operation gives the user	s a way to <i>acknowledge</i> all locally notified failures	12		
		PI_ANY_SOURCE receive operations that would have	13		
		<b></b> FAILED_PENDING due to process failure (see Sec-	14		
tion $15.2.2$	tion 15.2.2) proceed without further raising errors due to those acknowledged failures. Also				
	*	l not raise an error of class MPI_ERR_PROC_FAILED	16 17		
	ose acknowledged failures (acc	cording to the specification found later in this sec-	17		
tion).			19		
Adv	ice to users.		20		
Call	ing MPI_COMM_FAILURE_A	CK on a communicator with failed processes has no	21		
effec	t on collective operations (ex	cept for MPI_COMM_AGREE). If a collective oper-	22		
	ation would raise an error due to the communicator containing a failed process (as				
	defined in Section 15.2.2), it can continue to raise an error even after the failure has				
	8	se collective operations between processes of a com-	25 26		
	municator that contains failed processes, users should create a new communicator by calling MPI_COMM_SHRINK.				
	d of advice to users.)		28		
(Eno	u of uuvice to users.)		29		
			30		
MPI COM	1M_FAILURE_GET_ACKED( c	comm. failedgrp )	31		
IN	× ·	<b>C</b> · · <i>,</i>	32 33		
	comm	communicator (handle)	34		
OUT	failedgrp	group of failed processes (handle)	35		
int MDT		NT Comme comme MDT Concernity footloderers)	36		
int MP1_	Comm_1allure_get_acked(MP	PI_Comm comm, MPI_Group* failedgrp)	37		
	_failure_get_acked(comm,		38		
	(MPI_Comm), INTENT(IN) ::		39 40		
	TYPE(MPI_Group), INTENT(OUT) :: failedgrp INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
	GER, OFIIONAL, INIENI(UUI	) :: ierror	41 42		
	MPI_COMM_FAILURE_GET_ACKED(COMM, FAILEDGRP, IERROR) INTEGER COMM, FAILEDGRP, IERROR				
INTE	UR	44			
This	local operation returns the gr	coup failedgrp of processes, from the communicator	45		
	comm, that have been locally acknowledged as failed by preceding calls to				
MPI_COMM_FAILURE_ACK. The <i>failedgrp</i> can be empty, that is, equal to			47 48		

MPI\_GROUP\_EMPTY.

# Unofficial Draft for Comment Only

1	Advice to users. When they are not separated by a call to				
2	MPI_COMM_FAILURE_ACK, multiple calls to MPI_COMM_FAILURE_GET_ACKED				
3	produce similar failedgrp groups; that is, the result when providing these groups to				
4	MPI_GROUP_DIFFERENCE is MPI_EMPTY. (End of advice to users.)				
5					
6 7					
7 8	MPI_COMM_AGREE( comm, flag )				
9	IN comm communicator (handle)				
10 11	INOUT flag integer flag				
12 13	<pre>int MPI_Comm_agree(MPI_Comm comm, int* flag)</pre>				
14	MPI_Comm_agree(comm, flag, ierror)				
15	TYPE(MPI_Comm), INTENT(IN) :: comm				
16	INTEGER, INTENT(INOUT) :: flag				
17	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
18	MPI_COMM_AGREE(COMM, FLAG, IERROR)				
19	INTEGER COMM, FLAG, IERROR				
20	The numbers of this collective communication is to agree on the integer value flag and				
21 22	The purpose of this collective communication is to agree on the integer value flag and on the group of failed processes in comm.				
23	On completion, all non-failed processes have agreed to set the output integer value of				
24	flag to the result of a bitwise 'AND' operation over the contributed input values of flag.				
25	If comm is an intercommunicator, the value of flag is a bitwise 'AND' operation over the				
26	values contributed by the remote group.				
27	When a process fails before contributing to the operation, the flag is computed ignoring				
28	its contribution, and $MPI\_COMM\_AGREE$ raises an error of class $MPI\_ERR\_PROC\_FAILED.$				
29	However, if all processes have acknowledged this failure prior to the call to				
30	MPI_COMM_AGREE, using MPI_COMM_FAILURE_ACK, the error related to this failure is				
31	not raised. When an error of class MPI_ERR_PROC_FAILED is raised, it is consistently raised				
32	at all processes, in both the local and remote groups (if applicable).				
33 34	After MPI_COMM_AGREE raised an error of class MPI_ERR_PROC_FAILED, a subse- quent call to MPI_COMM_FAILURE_ACK on comm acknowledges the failure of every process				
35	that didn't contribute to the computation of flag.				
36	that that t contribute to the computation of hdg.				
37	Advice to users. Using a combination of MPI_COMM_FAILURE_ACK and				
38	$MPI_COMM_AGREE$ as illustrated in Example 15.3, users can propagate and syn-				
39	chronize the knowledge of failures across all processes in comm. When MPI_SUCCESS				
40	is returned locally from MPI_COMM_AGREE, the operation has not raised an error				
41	of class MPI_ERR_PROC_FAILED at any process and thereby returned MPI_SUCCESS at				
42	all other processes. (End of advice to users.)				
43	This function never raises an error of class MPI_ERR_REVOKED. The defined semantic				
44	of MPI_COMM_AGREE is maintained when comm is revoked, or when the group of comm				
45	contains failed processes.				
46 47					
48	Advice to users. MPI_COMM_AGREE is a collective operation, even when comm is revoked. ( <i>End of advice to users.</i> )				

MPI_COMM_IAGREE( comm, flag, req ) <sup>1</sup>				
IN	comm	communicator (handle)	2	
INOUT	flag	integer flag	3	
	0	0 0	4 5	
OUT	req	request (handle)	6	
			7	
<pre>int MPI_Comm_iagree(MPI_Comm comm, int* flag, MPI_Request* req)</pre>			8	
MPI_Comm_	_iagree(comm, flag, req,	ierror)	9	
TYPE	(MPI_Comm), INTENT(IN) ::	comm	10	
	ER, INTENT(INOUT) :: fl	•	11	
	(MPI_Request), INTENT(OUT	-	12	
INTEC	SER, OPTIONAL, INTENT(OUT)	) :: ierror	13 14	
MPI_COMM_	MPI_COMM_IAGREE(COMM, FLAG, REQ, IERROR)			
INTEC	GER COMM, FLAG, REQ, IERR	DR	15 16	
This	function has the same seman	tics as MPI_COMM_AGREE except that it is non-	17	
blocking.	*			
0			19	
15.3.2 O	15.3.2 One-Sided Functions			
			21	
			22	
MPI_WIN_	_REVOKE( win )		23	
IN	win	window (handle)	24 25	
			26	
int MPI_Win_revoke(MPI_Win win)			27	
MDT Uim a	<pre>MPI_Win_revoke(win, ierror)     TYPE(MPI_Win), INTENT(IN) :: win</pre>			
	GER, OPTIONAL, INTENT(OUT)		30	
			31	
MPI_WIN_REVOKE(WIN, IERROR)			32	
INTEC	GER WIN, IERROR		33	
This function notifies all MPI processes in the group associated with		esses in the group associated with the window win	34 35	
that this window is revoked. The revocation of a window by any process completes RM.			36	
oporations	on win at all proceeded and D	MA synchronizations on win raise an error of alass		

that this window is revoked. The revocation of a window by any process completes RMA operations on win at all processes and RMA synchronizations on win raise an error of class MPI\_ERR\_REVOKED. This function is not collective and therefore does not have a matching call on remote processes. All non-failed processes belonging to win will be notified of the revocation despite failures.

A window is revoked at a given process either when MPI\_WIN\_REVOKE is locally called on it, or when any MPI operation on win raises an error of class MPI\_ERR\_REVOKED at that process. Once a window has been revoked at a process, all subsequent RMA operations on that window are considered local and RMA synchronizations must complete by raising an error of class MPI\_ERR\_REVOKED at that process. In addition, the current epoch is closed and RMA operations originating from this process are interrupted and completed with undefined outputs.

46 47 48

37

38

39

40

41

42

43

44

```
1
     MPI_WIN_GET_FAILED( win, failedgrp )
2
       IN
                                               window (handle)
                 win
3
       OUT
                 failedgrp
                                               group of failed processes (handle)
4
5
6
      int MPI_Win_get_failed(MPI_Win win, MPI_Group* failedgrp)
\overline{7}
     MPI_Win_get_failed(win, failedgrp, ierror)
8
          TYPE(MPI_Win), INTENT(IN) :: win
9
          TYPE(MPI_Group), INTENT(OUT) :: failedgrp
10
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
11
12
     MPI_WIN_GET_FAILED(WIN, FAILEDGRP, IERROR)
          INTEGER COMM, FAILEDGRP, IERROR
13
14
          This local operation returns the group failedgrp of processes from the window win that
15
      are locally known to have failed.
16
17
           Advice to users.
                               MPI makes no assumption about asynchronous progress of the
18
           failure detection. A valid MPI implementation may choose to update only the group
19
           of locally known failed processes when it enters a synchronization function and must
20
           raise a process failure error. (End of advice to users.)
21
22
           Advice to users. It is possible that only the calling process has detected the reported
23
           failure. If global knowledge is necessary, processes detecting failures should use the
^{24}
           call MPI_WIN_REVOKE. (End of advice to users.)
25
26
      15.3.3 I/O Functions
27
28
29
      MPI_FILE_REVOKE( fh )
30
^{31}
       IN
                 fh
                                               file (handle)
32
33
      int MPI_File_revoke(MPI_File fh)
34
     MPI_File_revoke(fh, ierror)
35
          TYPE(MPI_File), INTENT(IN) ::
                                               fh
36
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                    ierror
37
38
     MPI_FILE_REVOKE(FH, IERROR)
39
          INTEGER FH, IERROR
40
          This function notifies all MPI processes in the group associated with the file handle fh
41
      that this file handle is revoked. The revocation of a file handle by any process completes
42
      non-local MPI operations on fh at all processes by raising an error of class
43
      MPI_ERR_REVOKED. This function is not collective and therefore does not have a matching
44
```

call on remote processes. All non-failed MPI processes belonging to fh will be notified of
 the revocation despite failures.

A file handle is revoked at a given process either when MPI\_FILE\_REVOKE is locally called on it, or when any MPI operation on fh raises an error of class MPI\_ERR\_REVOKED at that process. Once a file handle has been revoked at a process, all subsequent non-local operations on that file handle are considered local and must complete by raising an error of class MPI\_ERR\_REVOKED at that process.

# 15.4 Process Failure Error Codes and Classes

The following process failure error classes are added to those defined in Section 8.4:

MPI_ERR_PROC_FAILED	The operation could not complete because
	of a process failure (a fail-stop failure).
MPI_ERR_PROC_FAILED_PENDING	The operation was interupted by a process
	failure (a fail-stop failure). The request
	is still pending and the operation may be
	completed later.
MPI_ERR_REVOKED	The communication object used in the op-
	eration has been revoked.

Table 15.1: Additional process fault tolerance error classes

# 15.5 Examples

# 15.5.1 Safe Communicator Creation

The example below illustrates how a new communicator can be safely created despite disruption by process failures. A child communicator is created with MPI\_COMM\_SPLIT, then the global success of the operation is verified with MPI\_COMM\_AGREE. If any process failed to create the child communicator handle, all processes are notified by the value of the integer agreed on. Processes that had successfully created the child communicator handle destroy it, as it cannot be used consistently.

**Example 15.1** Fault Tolerant Communicator Split Example

```
int Comm_split_consistent(MPI_Comm parent, int color, int key, MPI_Comm* child)<sup>34</sup>
{
                                                                                    35
    rc = MPI_Comm_split(parent, color, key, child);
                                                                                    36
    split_ok = (MPI_SUCCESS == rc);
                                                                                    37
    MPI_Comm_agree(parent, &split_ok);
                                                                                    38
    if(split_ok) {
                                                                                    39
        /* All surviving processes have created the "child" comm
                                                                                    40
         * It may contain supplementary failures and the first
                                                                                    41
         * operation on it may raise an error, but it is a
                                                                                    42
         * workable object that will yield well specified outcomes */
                                                                                    43
        return MPI_SUCCESS;
                                                                                    44
    }
                                                                                    45
    else {
                                                                                    46
        /* At least one process did not create the child comm properly
                                                                                    47
         * if the local process did succeed in creating it, it disposes
                                                                                    48
```

```
1
               * of it, as it is a broken, inconsistent object */
\mathbf{2}
              if(MPI_SUCCESS == rc) {
3
                  MPI_Comm_free(child);
4
              }
5
              return MPI_ERR_PROC_FAILED;
6
         }
\overline{7}
     }
8
9
     15.5.2
            Obtaining the consistent group of failed processes
10
     Users can invoke MPI_COMM_FAILURE_ACK, MPI_COMM_FAILURE_GET_ACKED,
11
     MPI_WIN_GET_FAILED, to obtain the group of failed processes, as detected at the local
12
     process. However, these operations are local, thereby the invocation of the same function
13
     at another process can result in a different group of failed processes being returned.
14
         In the following examples, we illustrate two different approaches that permit obtaining
15
     the consistent group of failed processes across all processes of a communicator. The first
16
     one employs MPI_COMM_SHRINK to create a temporary communicator excluding failed
17
     processes. The second one employs MPI_COMM_AGREE to synchronize the set of acknowl-
18
     edged failures.
19
20
     Example 15.2
                       Fault-Tolerant Consistent Group of Failures Example (Shrink variant)
21
22
     Comm_failure_allget(MPI_Comm c, MPI_Group * g) {
23
         MPI_Comm s; MPI_Group c_grp, s_grp;
^{24}
25
         /* Using shrink to create a new communicator, the underlying
26
           * group is necessarily consistent across all processes, and excludes
27
           * all processes detected to have failed before the call */
28
         MPI_Comm_shrink(c, &s);
29
          /* Extracting the groups from the communicators */
30
         MPI_Comm_group(c, &c_grp);
31
         MPI_Comm_group(s, &s_grp);
32
         /* s_grp is the group of still alive processes, we want to
33
           * return the group of failed processes. */
34
         MPI_Group_difference(c_grp, s_grp, g);
35
36
         MPI_Group_free(&c_grp); MPI_Group_free(&s_grp);
37
         MPI_Comm_free(&s);
38
     }
39
40
                      Fault-Tolerant Consistent Group of Failures Example (Agree variant)
     Example 15.3
41
42
     Comm_failure_allget2(MPI_Comm c, MPI_Group * g) {
43
         int rc; int T=1;
44
45
         do {
46
              /* this routine is not pure: calling MPI_Comm_failure_ack
47
               * affects the state of the communicator c */
48
              MPI_Comm_failure_ack(comm);
```

```
/* we simply ignore the value in this example */
    rc = MPI_Comm_agree(comm, &T);
} while( rc != MPI_SUCCESS );
/* after this loop, MPI_Comm_agree has returned MPI_SUCCESS at
    * all processes, so all processes have Acknowledged the same set of
    * failures. Let's get that set of failures in the g group. */
    MPI_Comm_failure_get_acked(comm, g);
}
```

# 15.5.3 Fault-Tolerant Master/Worker

The example below presents a master code that handles worker failures by discarding failed worker processes and resubmitting the work to the remaining workers. It demonstrates the different failure cases that may occur when posting receptions from MPI\_ANY\_SOURCE as discussed in the advice to users in Section 15.2.2.

```
Example 15.4 Fault-Tolerant Master Example
```

```
18
int master(void)
                                                                                    19
{
    MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN);
                                                                                    20
                                                                                    21
    MPI_Comm_size(comm, &size);
                                                                                    22
                                                                                    23
    /* ... submit the initial work requests ... */
                                                                                    24
                                                                                    25
    /* Progress engine: Get answers, send new requests,
                                                                                    26
       and handle process failures */
    MPI_Irecv( buffer, 1, MPI_INT, MPI_ANY_SOURCE, tag, comm, &req );
                                                                                    27
                                                                                    28
    while( (active_workers > 0) && work_available ) {
                                                                                    29
        rc = MPI_Wait( &req, &status );
        if( MPI_SUCCESS == rc ) {
                                                                                    30
                                                                                    31
            /* ... process the answer and update work_available ... */
                                                                                    32
        }
                                                                                    33
        else {
                                                                                    34
            MPI_Error_class(rc, &ec);
                                                                                    35
            if( (MPI_ERR_PROC_FAILED == ec) ||
                                                                                    36
                 (MPI_ERR_PROC_FAILED_PENDING == ec) ) {
                                                                                    37
                 MPI_Comm_failure_ack(comm);
                                                                                    38
                 MPI_Comm_failure_get_acked(comm, &g);
                                                                                    39
                 MPI_Group_size(g, &gsize);
                                                                                    40
                                                                                    41
                 /* ... find the lost work and requeue it ... */
                                                                                    42
                 active_workers = size - gsize - 1;
                                                                                    43
                                                                                    44
                 MPI_Group_free(&g);
                                                                                    45
                                                                                    46
                 /* no need to repost when the request is still pending */
                                                                                    47
                 if( ec == MPI_ERR_PROC_FAILED_PENDING )
                                                                                    48
                     continue;
```

Unofficial Draft for Comment Only

1

2

4

5 6

9 10

11

12

13

14

15 16

```
1
                  }
2
              }
3
              /* get ready to receive more notifications from workers */
4
              MPI_Irecv( buffer, 1, MPI_INT, MPI_ANY_SOURCE, tag, comm, &req );
5
         }
6
         /* ... cancel request and cleanup ... */
7
     }
8
9
     15.5.4 Fault-Tolerant Iterative Refinement
10
     The example below demonstrates a method of fault tolerance for detecting and handling
11
     failures. At each iteration, the algorithm checks the return code of the
12
     MPI_ALLREDUCE. If the return code indicates a process failure for at least one process,
13
     the algorithm revokes the communicator, agrees on the presence of failures, and shrinks it
14
     to create a new communicator. By calling MPI_COMM_REVOKE, the algorithm ensures
15
     that all processes will be notified of process failure and enter the MPI_COMM_AGREE. If
16
     a process fails, the algorithm must complete at least one more iteration to ensure a correct
17
     answer.
18
19
     Example 15.5
                       Fault-tolerant iterative refinement with shrink and agreement
20
21
     while( gnorm > epsilon ) {
22
         /* Add a computation iteration to converge and
23
             compute local norm in lnorm */
24
         rc = MPI_Allreduce(&lnorm, &gnorm, 1, MPI_DOUBLE, MPI_MAX, comm);
25
         MPI_Error_class(rc, &ec);
26
27
         if( (MPI_ERR_PROC_FAILED == ec) ||
28
              (MPI_ERR_REVOKED == ec) ||
29
              (gnorm <= epsilon) ) {</pre>
30
31
              /* This process detected a failure, but other processes may have
32
               * proceeded into the next MPI_Allreduce. Since this process
33
               * will not match that following MPI_Allreduce, these other
34
               * processes would be at risk of deadlocking. This process thus
35
               * calls MPI_Comm_revoke to interrupt other processes and notify
36
               * them that it has detected a failure and is leaving the
37
               * failure free execution path to go into recovery. */
38
              if( MPI_ERR_PROC_FAILED == ec )
39
                  MPI_Comm_revoke(comm);
40
41
              /* About to leave: let's be sure that everybody
42
                 received the same information */
43
              allsucceeded = (rc == MPI_SUCCESS);
44
              rc = MPI_Comm_agree(comm, &allsucceeded);
45
              MPI_Error_class(rc, &ec);
46
              if( ec == MPI_ERR_PROC_FAILED || !allsucceeded ) {
47
                  MPI_Comm_shrink(comm, &comm2);
48
                  MPI_Comm_free(comm); /* Release the revoked communicator */
```

```
comm = comm2;
gnorm = epsilon + 1.0; /* Force one more iteration */
}
}
```

 $\mathbf{2}$ 

## Chapter 16

# **Deprecated Functions**

#### Deprecated since MPI-2.0 16.1

The following function is deprecated and is superseded by MPI\_COMM\_CREATE\_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 18.2.7. The language bindings are modified.

MPI\_KEYVAL\_CREATE(copy\_fn, delete\_fn, keyval, extra\_state)

IN	copy_fn	Copy callback function for keyval	23		
		· ·	24		
IN	delete_fn	Delete callback function for keyval	25		
OUT	keyval	key value for future access (integer)	26		
IN	extra_state	Extra state for callback functions	27		
	Skild_State		28		
int MDT V	ound amonts (MDT Const fun	ation woony fr	29		
IIIC MPI_K	eyval_create(MPI_Copy_fun	10	30		
		<pre>*delete_fn, int *keyval,</pre>	31		
	void* extra_state)				
For this rou	For this routine, an interface within the mpi_f08 module was never defined.				
PI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)					
EXTERNAL COPY_FN, DELETE_FN			36		
INTEG	ER KEYVAL, EXTRA_STATE, I	ERRUR	37		
The co	py_fn function is invoked wh	nen a communicator is duplicated by	38		
MPI_COM	MPI_COMM_DUP. copy_fn should be of type MPI_Copy_function, which is defined as follows:				
			40		
	4				
typedef i	nt MPI_Copy_function(MPI_	Comm oldcomm, int keyval,	42		
	void	<pre>*extra_state, void *attribute_val_in,</pre>	43		

A Fortran declaration for such a function is as follows: For this routine, an interface within the mpi\_f08 module was never defined.

void \*attribute\_val\_out, int \*flag)

```
620
                                              CHAPTER 16. DEPRECATED FUNCTIONS
1
     SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
\mathbf{2}
                    ATTRIBUTE_VAL_OUT, FLAG, IERR)
3
          INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
4
          ATTRIBUTE_VAL_OUT, IERR
5
          LOGICAL FLAG
6
          copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or
7
     FORTRAN; MPI_NULL_COPY_FN is a function that does nothing other than returning
8
     flag = 0 and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag =
9
     1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note
10
     that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated.
11
          Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn
12
     function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call
13
     is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function,
14
     which is defined as follows:
15
16
     typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
17
     void *attribute_val, void *extra_state);
18
19
          A Fortran declaration for such a function is as follows:
20
     For this routine, an interface within the mpi_f08 module was never defined.
21
     SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
22
          INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
23
^{24}
          delete_fn may be specified as MPI_NULL_DELETE_FN from either C or FORTRAN;
25
     MPI_NULL_DELETE_FN is a function that does nothing, other than returning
26
     MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated.
27
          The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL
28
     in MPI-2.0. The language independent definition of the deprecated function is the same as
29
     of the new function, except of the function name. The language bindings are modified.
30
^{31}
32
     MPI_KEYVAL_FREE(keyval)
33
       INOUT
                 keyval
                                             Frees the integer key value (integer)
34
35
     int MPI_Keyval_free(int *keyval)
36
37
     For this routine, an interface within the mpi_f08 module was never defined.
38
     MPI_KEYVAL_FREE(KEYVAL, IERROR)
39
          INTEGER KEYVAL, IERROR
40
41
          The following function is deprecated and is superseded by MPI_COMM_SET_ATTR in
42
     MPI-2.0. The language independent definition of the deprecated function is the same as of
43
     the new function, except of the function name. The language bindings are modified.
44
45
46
47
48
```

MPI_ATTI	R_PUT(comm, keyval, attri	bute_val)	1
INOUT	comm	communicator to which attribute will be attached (han- dle)	2 3 4
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE (integer)	4 5 6
IN	attribute_val	attribute value	7 8
int MPI_A	Attr_put(MPI_Comm comm	, int keyval, void* attribute_val)	9 10
For this re	outine, an interface within	the $mpi_f08$ module was never defined.	11
	_PUT(COMM, KEYVAL, ATTH GER COMM, KEYVAL, ATTR		12 13 14
MPI-2.0.	The language independent	ated and is superseded by MPI_COMM_GET_ATTR in definition of the deprecated function is the same as of tion name. The language bindings are modified.	15 16 17 18
MPI_ATT	R_GET(comm, keyval, attri	bute_val, flag)	19 20
IN	comm	communicator to which attribute is attached (handle)	21
IN	keyval	key value (integer)	22
OUT	attribute_val	attribute value, unless $flag = false$	23 24
OUT	flag	true if an attribute value was extracted; false if no attribute is associated with the key	25 26 27
int MPI_A	Attr_get(MPI_Comm comm	, int keyval, void *attribute_val, int *flag)	28
For this re	outine, an interface within	the mpi_f08 module was never defined.	29 30
INTE	_GET(COMM, KEYVAL, ATTH GER COMM, KEYVAL, ATTR CAL FLAG	RIBUTE_VAL, FLAG, IERROR) IBUTE_VAL, IERROR	31 32 33
The following function is deprecated and is superseded by MPI_COMM_DELETE_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.			
MPI_ATT	R_DELETE(comm, keyval)		39
INOUT	comm	communicator to which attribute is attached (handle)	40 41
IN	keyval	The key value of the deleted attribute (integer)	42
4:			
int MPI_Attr_delete(MPI_Comm comm, int keyval) 4			
For this re	outine, an interface within	the mpi_f08 module was never defined.	46
MPI_ATTR	_DELETE(COMM, KEYVAL, I	IERROR)	47 48

### Unofficial Draft for Comment Only

1 2	INTEGEF	R COMM, KEYVAL, IERROR	
3 4	16.2 Dep	precated since MPI-2.2	
5 6 7 8 9 10	Interfaces for The foll names. Othe	r more information. lowing function typedefs hav er than the typedef names, th	a have been removed. See Chapter 17, Removed e been deprecated and are superseded by new he function signatures are exactly the same; the s of other function typedef names.
11		Deprecated Name	New Name
12 13 14 15		MPI_Comm_errhandler_fn MPI_File_errhandler_fn MPI_Win_errhandler_fn	MPI_Comm_errhandler_function MPI_File_errhandler_function MPI_Win_errhandler_function
16			
17			
18			
19			
20			
21			
22			
23			
24			
25 26			
20 27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37 38			
30 39			
40			
41			
42			
43			
44			
45			
46			
47			
48			

## Chapter 17

# **Removed Interfaces**

### 17.1 Removed MPI-1 Bindings

#### 17.1.1 Overview

The following MPI-1 bindings were deprecated as of MPI-2 and are removed in MPI-3. They may be provided by an implementation for backwards compatibility, but are not required. Removal of these bindings affects all language-specific definitions thereof. Only the language-neutral bindings are listed when possible.

#### 17.1.2 Removed MPI-1 Functions

Table 17.1 shows the removed MPI-1 functions and their replacements.

Removed	MPI-2 Replacement
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT

Table 17.1: Removed MPI-1 functions and their replacements

#### 17.1.3 Removed MPI-1 Datatypes

Table 17.2 shows the removed MPI-1 datatypes and their replacements.

#### 17.1.4 Removed MPI-1 Constants

Table 17.3 shows the removed MPI-1 constants. There are no MPI-2 replacements.

 $44 \\ 45$ 

	624 CHAPTER 17. REMOVED INTERFACES
1	Removed MPI-2 Replacement
2	MPI_LB MPI_TYPE_CREATE_RESIZED
3	MPI_UB MPI_TYPE_CREATE_RESIZED
4	
5	
6	Table 17.2: Removed MPI-1 datatypes and their replacements
7	Removed MPI-1 Constants
8	C type: const int (or unnamed enum)
9	Fortran type: INTEGER
10	MPI_COMBINER_HINDEXED_INTEGER
11	MPI_COMBINER_HVECTOR_INTEGER
12	MPI_COMBINER_STRUCT_INTEGER
13	
14	Table 17.9. David MDI 1 and the
15	Table 17.3: Removed MPI-1 constants
16 17	
18	17.1.5 Removed MPI-1 Callback Prototypes
19	Table 17.4 shows the removed MPI-1 callback prototypes and their MPI-2 replacements.
20	
21	Removed MPI-2 Replacement
22	MPI_Handler_function MPI_Comm_errhandler_function
23	
24	Table 17.4: Removed MPI-1 callback prototypes and their replacements
25	
26	
27	17.2 C++ Bindings
28	
29 30	The C++ bindings were deprecated as of MPI-2.2. The C++ bindings are removed in
31	MPI-3.0. The namespace is still reserved, however, and bindings may only be provided by
32	an implementation as described in the MPI-2.2 standard.
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45 46	
46 47	
48	

## Chapter 18

# Language Bindings

### 18.1 Fortran Support

#### 18.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [40] + TS 29113 [41].

Rationale. Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 + TS 29113, the major new language features used are the ASYNCHRONOUS attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 18.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

- 1. USE mpi\_f08: This method is described in Section 18.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.
- 2. USE mpi: This method is described in Section 18.1.3 and requires compile-time argument checking. Handles are defined as INTEGER. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.
- 3. **INCLUDE 'mpif.h':** This method is described in Section 18.1.4. The use of the include file mpif.h is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

Compliant MPI-3 implementations providing a Fortran interface must provide one or  $\mathbf{2}$ 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 18.1.6 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 19 .TRUE. if all buffer choice arguments are defined in explicit interfaces with assumed-type 20and assumed-rank [41]; otherwise it is set to .FALSE.. The LOGICAL compile-time constant 21MPI\_ASYNC\_PROTECTS\_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was 22 added to the choice buffer arguments of all nonblocking interfaces and the underlying 23Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of  $^{24}$ TS 29113), otherwise it is set to .FALSE.. These constants exist for each Fortran support 25method, but not in the C header file. The values may be different for each Fortran support 26method. All other constants and the integer values of handles must be the same for each 27Fortran support method.

28Section 18.1.2 through 18.1.4 define the Fortran support methods. The Fortran in-29terfaces of each MPI routine are shorthands. Section 18.1.5 defines the corresponding 30 full interface specification together with the specific procedure names and implications for  $^{31}$ the profiling interface. Section 18.1.6 the implementation of the MPI routines for differ-32 ent versions of the Fortran standard. Section 18.1.7 summarizes major requirements for 33 valid MPI-3.0 implementations with Fortran support. Section 18.1.8 and Section 18.1.9 de-34scribe additional functionality that is part of the Fortran support. MPI\_F\_SYNC\_REG is 35 needed for one of the methods to prevent register optimization problems. A set of func-36 tions provides additional support for Fortran intrinsic numeric types, including parameter-37 ized types: MPI\_SIZEOF, MPI\_TYPE\_MATCH\_SIZE, MPI\_TYPE\_CREATE\_F90\_INTEGER, 38 MPI\_TYPE\_CREATE\_F90\_REAL and MPI\_TYPE\_CREATE\_F90\_COMPLEX. In the context 39 of MPI, parameterized types are Fortran intrinsic types which are specified using KIND type 40 parameters. Sections 18.1.10 through 18.1.19 give an overview and details on known prob-41 lems when using Fortran together with MPI; Section 18.1.20 compares the Fortran problems 42with those in C.

43

#### 44Fortran Support Through the mpi\_f08 Module 18.1.2 45

46An MPI implementation providing a Fortran interface must provide a module named mpi\_f08 47that can be used in a Fortran program. Section 18.1.6 describes restrictions if the compiler 48does 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 18.1.3.

Advice to users. It is strongly recommended that developers substitute calls to deprecated routines when upgrading from mpif.h or the mpi module to the mpi\_f08 module. (End of advice to users.)

- Define the derived type MPI\_Status, and define all MPI handles with uniquely named handle types (instead of INTEGER handles, as in the mpi module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).
- Overload the operators .EQ. and .NE. to allow the comparison of these MPI handles with .EQ., .NE., == and /=.
- Use the ASYNCHRONOUS attribute to protect the buffers of nonblocking operations, and set the LOGICAL compile-time constant MPI\_ASYNC\_PROTECTS\_NONBLOCKING to .TRUE. if the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113). See Section 18.1.6 for older compiler versions.
- Set the LOGICAL compile-time constant MPI\_SUBARRAYS\_SUPPORTED to .TRUE. and declare choice buffers using the Fortran 2008 TS 29113 features assumed-type and assumed-rank, i.e., TYPE(\*), DIMENSION(..) in all nonblocking, split collective and persistent communication routines, if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

*Rationale.* In all blocking routines, i.e., if the choice-buffer is not declared as ASYNCHRONOUS, the TS 29113 feature is not needed for the support of non-contiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the MPI\_SUBARRAYS\_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TS 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 18.1.6 for details.
- Declare each argument with an INTENT of IN, OUT, or INOUT as defined in this standard.

 $\frac{4}{5}$ 

 $^{31}$ 

Rationale. For these definitions in the mpi\_f08 bindings, in most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for OUT and INOUT dummy arguments that allow one of the non-ordinary Fortran constants (see MPI\_BOTTOM, etc. in Section 2.5.4) as input, an INTENT is not specified. (End of rationale.)

Advice to users. If a dummy argument is declared with INTENT(OUT), then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [40], 12.5.2.4 Ordinary dummy variables, Paragraph 17: "If a dummy argument has INTENT(OUT), the actual argument becomes undefined at the time the association is established, except [...]". For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of INTENT(OUT) the copy-in may be suppressed by the optimization and the routine starts execution using an array of undefined values. If the routine stores fewer elements into the dummy argument than is provided in the actual argument, then the remaining locations are overwritten with these undefined values. See also both advices to implementors in Section 18.1.3. (End of advice to users.)

• Declare all ierror output arguments as OPTIONAL, except for user-defined callback functions (e.g., COMM\_COPY\_ATTR\_FUNCTION) and predefined callbacks (e.g., MPI\_COMM\_NULL\_COPY\_FN).

*Rationale.* For user-defined callback functions (e.g., COMM\_COPY\_ATTR\_FUNCTION) and their predefined callbacks (e.g., MPI\_COMM\_NULL\_COPY\_FN), the ierror argument is not optional. The MPI library must always call these routines with an actual ierror argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

The MPI Fortran bindings in the mpi\_f08 module are designed based on the Fortran 2008 standard [40] together with the Technical Specification "TS 29113 Further Interoperability with C" [41] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The features in TS 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to [41], "an ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn."

The TS 29113 contains the following language features that are needed for the MPI bindings in the mpi\_f08 module: assumed-type and assumed-rank. It is important that any possible actual argument can be used for such dummy arguments, e.g., scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and with any element type, e.g., REAL, CHARACTER\*5, CHARACTER\*(\*), sequence derived types, or BIND(C) derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with mpif.h without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the mpi\_f08 module).

A further feature useful for MPI is the extension of the semantics of the ASYNCHRONOUS attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TS 29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (*End of rationale.*)

#### 18.1.3 Fortran Support Through the mpi Module

An MPI implementation providing a Fortran interface must provide a module named mpi that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists. If an implementation is paired with a compiler that either does not support TYPE(\*), DIMENSION(..) from TS 29113, or is otherwise unable to ignore the types of choice buffers, then the implementation must provide explicit interfaces only for MPI routines with no choice buffer arguments. See Section 18.1.6 for more details.
- Define all MPI handles as type INTEGER.
- Define the derived type MPI\_Status and all named handle types that are used in the mpi\_f08 module. For these named handle types, overload the operators .EQ. and .NE. to allow handle comparison via the .EQ., .NE., == and /= operators.

*Rationale.* They are needed only when the application converts old-style INTEGER handles into new-style handles with a named type. (*End of rationale.*)

- A high quality MPI implementation may enhance the interface by using the ASYNCHRONOUS attribute in the same way as in the mpi\_f08 module if it is supported by the underlying compiler.
- Set the LOGICAL compile-time constant MPI\_ASYNC\_PROTECTS\_NONBLOCKING to .TRUE. if the ASYNCHRONOUS attribute is used in all nonblocking interfaces and the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113), otherwise to .FALSE..

#### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{5}$ 

For an MPI implementation that fully supports nonblocking calls Advice to users. with the ASYNCHRONOUS attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses "contiguous" but not "simply contiguous" ASYNCHRONOUS arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying (1:n) for a whole dimension instead of using (:). This should be fixed to fulfill the Fortran constraints for ASYNCHRONOUS dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the ASYNCHRONOUS attribute to protect nonblocking calls. Another reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copy-in/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. See Section 18.1.12 for more details. (End of advice to users.)

- A high quality MPI implementation may enhance the interface by using TYPE(\*), DIMENSION(..) choice buffer dummy arguments instead of using non-standardized extensions such as !\$PRAGMA IGNORE\_TKR or a set of overloaded functions as described by M. Hennecke in [28], if the compiler supports this TS 29113 language feature. See Section 18.1.6 for further details.
  - Set the LOGICAL compile-time constant MPI\_SUBARRAYS\_SUPPORTED to .TRUE. if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with TYPE(\*), DIMENSION(..), otherwise set it to .FALSE.. When MPI\_SUBARRAYS\_SUPPORTED is defined as
    - .TRUE., non-contiguous sub-arrays can be used as buffers in nonblocking routines.
  - Set the MPI\_SUBARRAYS\_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TS 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in non-blocking calls may be disallowed. See Section 18.1.6 for details.

An MPI implementation may provide other features in the mpi module that enhance the usability of MPI while maintaining adherence to the standard. For example, it may provide INTENT information in these interface blocks.

Advice to implementors. The appropriate INTENT may be different from what is given in the MPI language-neutral bindings. Implementations must choose INTENT so that the function adheres to the MPI standard, e.g., by defining the INTENT as provided in the mpi\_f08 bindings. (*End of advice to implementors.*)

Rationale. The intent given by the MPI generic interface is not precisely defined
 and does not in all cases correspond to the correct Fortran INTENT. For instance,
 receiving into a buffer specified by a datatype with absolute addresses may require
 associating MPI\_BOTTOM with a dummy OUT argument. Moreover, "constants" such
 MPI\_BOTTOM and MPI\_STATUS\_IGNORE are not constants as defined by Fortran,
 but "special addresses" used in a nonstandard way. Finally, the MPI-1 generic intent

#### Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23

24

25

26

27 28

29

30

31

32

33 34

35

36

37 38

39

40

41

was changed in several places in MPI-2. For instance, MPI\_IN\_PLACE changes the intent of an OUT argument to be INOUT. (End of rationale.)

Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 4 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  $\overline{7}$ 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 ac-10 tual argument is definable." Applications that include mpif.h may not expect that 11 INTENT (OUT) is used. In particular, output array arguments are expected to keep their 12content as long as the MPI routine does not modify them. To keep this behavior, it is 13 recommended that implementations not use INTENT(OUT) in the mpi module and the 14mpif.h include file, even though INTENT(OUT) is specified in an interface description 15of the mpi\_f08 module. (End of advice to implementors.) 16

#### Fortran Support Through the mpif.h Include File 18.1.4

The use of the mpif.h include file is strongly discouraged and may be deprecated in a future version of MPI.

An MPI implementation providing a Fortran interface must provide an include file named mpif.h that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is supported by this include file. This include file must:

- Define all named MPI constants. • Declare MPI functions that return a value. • Define all handles as INTEGER. • Be valid and equivalent for both fixed and free source form. For each MPI routine, an implementation can choose to use an implicit or explicit interface for the second Fortran binding (in deprecated routines, the first one may be omitted). • Set the LOGICAL compile-time constants MPI\_SUBARRAYS\_SUPPORTED and MPI\_ASYNC\_PROTECTS\_NONBLOCKING according to the same rules as for the mpi module. In the case of implicit interfaces for choice buffer or nonblocking routines, the constants must be set to .FALSE.. Advice to users. Instead of using mpif.h, the use of the mpi\_f08 or mpi module is strongly encouraged for the following reasons:
  - Most mpif.h implementations do not include compile-time argument checking.
  - Therefore, many bugs in MPI applications remain undetected at compile-time, 45such as: 46
    - Missing ierror as last argument in most Fortran bindings.

#### **Unofficial Draft for Comment Only**

1  $\mathbf{2}$ 

3

5

6

8

9

17 18

19

20

21

22

23

24

2526

2728

29

30 31

32 33

34

35

36

37

38

39 40

41

4243

44

47

	632 CHAPTER 18. 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>
3 4	<ul> <li>Incorrect argument positions; e.g., interchanging the count and datatype arguments.</li> </ul>
5 6	<ul> <li>Passing incorrect MPI handles; e.g., passing a datatype instead of a commu- nicator.</li> </ul>
7 8 9 10	• The migration from mpif.h to the mpi module should be relatively straightforward (i.e., substituting include 'mpif.h' after an implicit statement by use mpi before that implicit statement) as long as the application syntax is correct.
11 12 13	• Migrating portable and correctly written applications to the mpi module is not expected to be difficult. No compile or runtime problems should occur because an mpif.h include file was always allowed to provide explicit Fortran interfaces.
14 15	(End of advice to users.)
16 17 18 19 20	Rationale. With MPI-3.0, the mpif.h include file was not deprecated in order to retain strong backward compatibility. Internally, mpif.h and the mpi module may be implemented so that essentially the same library implementation of the MPI routines can be used. ( <i>End of rationale.</i> )
21 22	18.1.5 Interface Specifications, Procedure Names, and the Profiling Interface
23 24 25 26 27 28 29 30 31 32	The Fortran interface specification of each MPI routine specifies the routine name that must be called by the application program, and the names and types of the dummy arguments together with additional attributes. The Fortran standard allows a given Fortran interface to be implemented with several methods, e.g., within or outside of a module, with or without BIND(C), or the buffers with or without TS 29113. Such implementation decisions imply different binary interfaces and different specific procedure names. The requirements for several implementation schemes together with the rules for the specific procedure names and its implications for the profiling interface are specified within this section, but not the implementation details.
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 37	• Portable implementation of the wrappers from the MPI Fortran interfaces to the MPI routines in C.
38 39	• Binary backward compatible implementation path when switching MPI_SUBARRAYS_SUPPORTED from .FALSE. to .TRUE
40 41 42 43	• The Fortran PMPI interface need not be backward compatible, but a method must be included that a tools layer can use to examine the MPI library about the specific procedure names and interfaces used.
43	• No performance drawbacks.
45	• Consistency between all three Fortran support methods.
46 47 48	• Consistent with Fortran 2008 + TS 29113.

No.	Specific pro- cedure name	Calling convention
1A	MPI_Isend_f08	Fortran interface and arguments, as in Annex A.3, except
		that in routines with a choice buffer dummy argument, this
		dummy argument is implemented with non-standard ex-
		tensions like !\$PRAGMA IGNORE_TKR, which provides a call-
		by-reference argument without type, kind, and dimension
		checking.
В	MPI_Isend_f08ts	Fortran interface and arguments, as in Annex A.3, but
		only for routines with one or more choice buffer dummy
		arguments; these dummy arguments are implemented with
		TYPE(*), DIMENSION().
А	MPI_ISEND	Fortran interface and arguments, as in Annex A.4, except
		that in routines with a choice buffer dummy argument, this
		dummy argument is implemented with non-standard ex-
		tensions like <b>!</b> \$PRAGMA IGNORE_TKR, which provides a call-
		by-reference argument without type, kind, and dimension
		checking.
2B	MPI_ISEND_FTS	Fortran interface and arguments, as in Annex A.4, but
		only for routines with one or more choice buffer dummy
		arguments; these dummy arguments are implemented with
		TYPE(*), DIMENSION().

Table 18.1: Specific Fortran procedure names and related calling conventions. MPI\_ISEND is used as an example. For routines without choice buffers, only 1A and 2A apply.

The design expected that all dummy arguments in the MPI Fortran interfaces are interoperable with C according to Fortran 2008 + TS 29113. This expectation was not fulfilled. The LOGICAL arguments are not interoperable with C, mainly because the internal representations for .FALSE. and .TRUE. are compiler dependent. The provided interface was mainly based on BIND(C) interfaces and therefore inconsistent with Fortran. To be consistent with Fortran, the BIND(C) had to be removed from the callback procedure interfaces and the predefined callbacks, e.g., MPI\_COMM\_DUP\_FN. Non-BIND(C) procedures are also not interoperable with C, and therefore the BIND(C) had to be removed from all routines with PROCEDURE arguments, e.g., from MPI\_OP\_CREATE.

Therefore, this section was rewritten as an erratum to MPI-3.0. (End of rationale.)

A Fortran call to an MPI routine shall result in a call to a procedure with one of the specific procedure names and calling conventions, as described in Table 18.1. Case is not significant in the names.

Note that for the deprecated routines in Section 16.1, which are reported only in Annex A.4, scheme 2A is utilized in the mpi module and mpif.h, and also in the mpi\_f08 module.

To set MPI\_SUBARRAYS\_SUPPORTED to .TRUE. within a Fortran support method, it is required that all non-blocking and split-collective routines with buffer arguments are 

#### **Unofficial Draft for Comment Only**

implemented according to 1B and 2B, i.e., with MPI\_Xxxx\_f08ts in the mpi\_f08 module,
 and with MPI\_XXXX\_FTS in the mpi module and the mpif.h include file.

The mpi and mpi\_f08 modules and the mpif.h include file will each correspond to exactly one implementation scheme from Table 18.1. However, the MPI library may contain multiple implementation schemes from Table 18.1.

Advice to implementors. This may be desirable for backwards binary compatibility in the scope of a single MPI implementation, for example. (*End of advice to implementors.*)

10 Rationale. After a compiler provides the facilities from TS 29113, i.e., TYPE(\*), 11 DIMENSION(...), it is possible to change the bindings within a Fortran support method 12to support subarrays without recompiling the complete application provided that the 13 previous interfaces with their specific procedure names are still included in the li-14brary. Of course, only recompiled routines can benefit from the added facilities. 15There is no binary compatibility conflict because each interface uses its own spe-16cific procedure names and all interfaces use the same constants (except the value of 17 MPI\_SUBARRAYS\_SUPPORTED and MPI\_ASYNC\_PROTECTS\_NONBLOCKING) and type 18 definitions. After a compiler also ensures that buffer arguments of nonblocking MPI 19 operations can be protected through the ASYNCHRONOUS attribute, and the proce-20dure declarations in the mpi\_f08 and mpi module and the mpif.h include file declare 21choice buffers with the ASYNCHRONOUS attribute, then the value of 22

- MPI\_ASYNC\_PROTECTS\_NONBLOCKING can be switched to .TRUE. in the module definition and include file. (*End of rationale.*)
  - Advice to users. Partial recompilation of user applications when upgrading MPI implementations is a highly complex and subtle topic. Users are strongly advised to consult their MPI implementation's documentation to see exactly what is and what is not supported. (*End of advice to users.*)

28 29 30

25

26

27

Within the mpi\_f08 and mpi modules and mpif.h, for all MPI procedures, a second procedure with the same calling conventions shall be supplied, except that the name is modified by prefixing with the letter "P", e.g., PMPI\_Isend. 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.

<sup>35</sup> A user-written or middleware profiling routine should provide the same specific Fortran <sup>36</sup> procedure names and calling conventions, and therefore can interpose itself as the MPI <sup>37</sup> library routine. The profiling routine can internally call the matching

PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments, choice buffer arguments, or that are attribute caching routines (

MPI\_{COMM|WIN|TYPE}\_{SET|GET}\_ATTR). In this case, the profiling software should
 invoke the corresponding PMPI routine using the same Fortran support method as used in
 the calling application program, because the C, mpi\_f08 and mpi callback prototypes are
 different or the meaning of the choice buffer or attribute\_val arguments are different.

- <sup>45</sup> Advice to users. Although for each support method and MPI routine (e.g.,
- <sup>46</sup> MPI\_ISEND in mpi\_f08), multiple routines may need to be provided to intercept
   <sup>47</sup> the specific procedures in the MPI library (e.g., MPI\_Isend\_f08 and MPI\_Isend\_f08ts),
   <sup>48</sup> each profiling routine itself uses only one support method (e.g., mpi\_f08) and calls

#### Unofficial Draft for Comment Only

7

8

the real MPI routine through the one PMPI routine defined in this support method (i.e., PMPI\_lsend in this example). (*End of advice to users.*)

Advice to implementors. If all of the following conditions are fulfilled:

- the handles in the mpi\_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a non-optional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as OPTIONAL,
- the internal argument passing mechanism for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

Advice to implementors. In the Fortran support method mpif.h, compile-time argument checking can be also implemented for all routines. For mpif.h, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that mpif.h must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the two longest subroutine declaration statements are

```
SUBROUTINE PMPI_Dist_graph_create_adjacent(a,b,c,d,e,f,g,h,i,j,k)
SUBROUTINE PMPI_Rget_accumulate(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 71 and 66 characters. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. The longest of such interface definitions is

```
INTERFACE PMPI_Rget_accumulate
SUBROUTINE PMPI_Rget_accumulate_fts(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 70 characters. In principle, continuation lines would be possible in mpif.h (spaces in columns 73–131, & in column 132, and in column 6 of the continuation line) but this would not be valid if the source line length is extended with a compiler flag to 132 characters. Column 133 is also not available for the continuation character because lines longer than 132 characters are invalid with some compilers by default.

The longest specific procedure names are PMPI\_Dist\_graph\_create\_adjacent\_f08 and PMPI\_File\_write\_ordered\_begin\_f08ts both with 35 characters in the mpi\_f08 module.

For example, the interface specifications together with the specific procedure names can be implemented with

#### Unofficial Draft for Comment Only

 $^{31}$ 

 $46 \\ 47$ 

```
1
              MODULE mpi_f08
2
                 TYPE, BIND(C) :: MPI_Comm
3
                    INTEGER :: MPI_VAL
                 END TYPE MPI_Comm
4
                 . . .
5
                 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
6
                    SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)
7
                       IMPORT :: MPI_Comm
8
                      TYPE(MPI_Comm),
                                                   INTENT(IN) :: comm
9
                                                   INTENT(OUT) :: rank
                      INTEGER,
                      INTEGER, OPTIONAL,
                                                   INTENT(OUT) :: ierror
10
                    END SUBROUTINE
11
                 END INTERFACE
12
              END MODULE mpi_f08
13
14
              MODULE mpi
15
                 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
16
                    SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
17
                       INTEGER, INTENT(IN) :: comm
                                                                 ! The INTENT may be added although
                      INTEGER, INTENT(OUT) :: rank
                                                                 ! it is not defined in the
18
                      INTEGER, INTENT(OUT) :: ierror ! official routine definition.
19
                   END SUBROUTINE
20
                 END INTERFACE
21
              END MODULE mpi
22
23
              And if interfaces are provided in mpif.h, they might look like this (outside of any
24
              module and in fixed source format):
25
               1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
26
                       INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)
27
                        SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
28
                         INTEGER, INTENT(IN) :: comm ! The argument names may be
29
                         INTEGER, INTENT(OUT) :: rank
                                                                    ! shortened so that the
30
                         INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the
31
                        END SUBROUTINE
                                                                    ! maximum of 72 characters.
32
                       END INTERFACE
33
34
              (End of advice to implementors.)
35
                                    The following is an example of how a user-written or middleware
              Advice to users.
36
              profiling routine can be implemented:
37
38
              SUBROUTINE MPI_Isend_f08ts(buf,count,datatype,dest,tag,comm,request,ierror)
39
                 USE :: mpi_f08, my_noname => MPI_Isend_f08ts
40
                 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
41
                 INTEGER,
                                             INTENT(IN)
                                                                :: count, dest, tag
42
                 TYPE(MPI_Datatype), INTENT(IN)
                                                                   :: datatype
                 TYPE(MPI_Comm),
                                            INTENT(IN)
                                                                   :: comm
43
                 TYPE(MPI_Request), INTENT(OUT)
                                                                    :: request
44
                 INTEGER, OPTIONAL,
                                            INTENT(OUT)
                                                                    :: ierror
45
                    ! ... some code for the begin of profiling
46
                 call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)
47
                    ! ... some code for the end of profiling
48
              END SUBROUTINE MPI_Isend_f08ts
```

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, 14 and profiling routines that are part of the application and have access to the appli-15cation data. With MPI-3.0, new Fortran interfaces and implementation schemes were 16 introduced that have several implications on how Fortran MPI routines are internally 17 implemented and optimized. For profiling layers, these schemes imply that several in-18 ternal interfaces with different specific procedure names may need to be intercepted, 19 as shown in the example code above. Therefore, for wrapper routines that are part 20of a Fortran application, it may be more convenient to make the name shift within 21the application, i.e., to substitute the call to the MPI routine (e.g., MPI\_ISEND) by a 22call to a user-written profiling wrapper with a new name (e.g., X\_MPI\_ISEND) and to 23call the Fortran MPI\_ISEND from this wrapper, instead of using the PMPI interface.  $^{24}$ (End of advice to users.) 25

Advice to implementors. An implementation that provides a Fortran interface must provide a combination of MPI library and module or include file that uses the specific procedure names as described in Table 18.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

#### 18.1.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

- For Fortran 77 with some extensions:
  - MPI identifiers may be up to 30 characters (31 with the profiling interface).
  - MPI identifiers may contain underscores after the first character.
  - An MPI subroutine with a choice argument may be called with different argument types.
  - Although not required by the MPI standard, the INCLUDE statement should be available for including mpif.h into the user application source code.

Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute addresses from MPI\_ADDRESS and MPI\_BOTTOM may cause problems if an address does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem is solved with MPI\_GET\_ADDRESS, but not for Fortran 77.)
45
46
47
48

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8 9

10

26

27

28

29

30 31

32 33

34

35

36 37

38

39 40

41

42

43

1	• For Fortran 90:
2	The major additional features that are needed from Fortran 90 are:
3	- The MODULE and INTERFACE concept.
4 5	- The KIND= and SELECTEDKIND concept.
6	-
7	- Fortran derived TYPEs and the SEQUENCE attribute.
8	- The OPTIONAL attribute for dummy arguments.
9 10	<ul> <li>Cray pointers, which are a non-standard compiler extension, are needed for the use of MPI_ALLOC_MEM.</li> </ul>
11 12 13 14 15 16	With these features, MPI-1.1 – MPI-2.2 can be implemented without restrictions. MPI-3.0 can be implemented with some restrictions. The Fortran support methods are abbreviated with $S1 = \text{the mpi_f08} \mod S2 = \text{the mpi} \mod S3 = \text{the mpif.f}$ include file. If not stated otherwise, restrictions exist for each method which prevent implementing the complete semantics of MPI-3.0.
17 18 19 20	<ul> <li>MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non- contiguous subarrays cannot be used as buffers in nonblocking routines, RMA, or split-collective I/O.</li> </ul>
21 22	- S1, S2, and S3 can be implemented, but for S1, only a preliminary implementation is possible.
23	- In this preliminary interface of S1, the following changes are necessary:
24 25	* TYPE(*), DIMENSION() is substituted by non-standardized extensions like !\$PRAGMA IGNORE_TKR.
26 27	* The ASYNCHRONOUS attribute is omitted.
28	* PROCEDURE() callback declarations are substituted by EXTERNAL.
29	- The specific procedure names are specified in Section 18.1.5.
30 31 32	<ul> <li>Due to the rules specified in Section 18.1.5, choice buffer declarations should be implemented only with non-standardized extensions like !\$PRAGMA IGNORE_TKR (as long as F2008+TS 29113 is not available).</li> </ul>
33 34 35 36 37	In S2 and S3: Without such extensions, routines with choice buffers should be provided with an implicit interface, instead of overloading with a different MPI function for each possible buffer type (as mentioned in Section 18.1.11). Such overloading would also imply restrictions for passing Fortran derived types as choice buffer, see also Section 18.1.15.
38 39 40 41	Only in S1: The implicit interfaces for routines with choice buffer arguments imply that the ierror argument cannot be defined as OPTIONAL. For this reason, it is recommended not to provide the mpi_f08 module if such an extension is not available.
42 43 44	<ul> <li>The ASYNCHRONOUS attribute can not be used in applications to protect buffers in nonblocking MPI calls (S1–S3).</li> </ul>
45 46	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines is not available.
47	
48	

	<ul> <li>In S1 and S2, the definition of the handle types (e.g., TYPE(MPI_Comm) and the status type TYPE(MPI_Status) must be modified: The SEQUENCE attribute must be used instead of BIND(C) (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the mpi_f08 module for Fortran 90 compilers. In this case, the mpi_f08 handle types and all routines, constants and types related to TYPE(MPI_Status) (see Section 18.2.5) are also not available in the mpi module and mpif.h.</li> </ul>	1 2 3 4 5 6 7 8 9
•	• For Fortran 95: The quality of the MPI interface and the restrictions are the same as with Fortran 90.	10 11 12
•	<i>For Fortran 2003:</i> The major features that are needed from Fortran 2003 are:	13 14 15
	<ul> <li>Interoperability with C, i.e.,</li> <li>* BIND(C) derived types.</li> <li>* The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.</li> <li>The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy arguments.</li> </ul>	16 17 18 19 20 21
	<ul> <li>The ability to overload the operators .EQ. and .NE. to allow the comparison of derived types (used in MPI-3.0 for MPI handles).</li> <li>The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O. This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TS 29113.</li> </ul>	22 23 24 25 26 27
	With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2 can be implemented without restrictions, but with one enhancement:	28 29 30
	<ul> <li>The user application can use TYPE(C_PTR) together with MPI_ALLOC_MEM as long as MPI_ALLOC_MEM is defined with an implicit interface because a C_PTR and an INTEGER(KIND=MPI_ADDRESS_KIND) argument must both map to a void * argument.</li> </ul>	31 32 33 34
	MPI-3.0 can be implemented with the following restrictions:	35 36
	- MPI_SUBARRAYS_SUPPORTED equals .FALSE	37
	<ul> <li>For \$1, only a preliminary implementation is possible. The following changes are necessary:</li> </ul>	38 39 40
	<pre>* TYPE(*), DIMENSION() is substituted by non-standardized extensions like !\$PRAGMA IGNORE_TKR.</pre>	41 42
	- The specific procedure names are specified in Section 18.1.5.	43
	- 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.	44 45 46 47
		-11

	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. The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can be used only for Fortran types that are C compatible. The same restriction as for Fortran 90 applies if non-standardized extensions like !\$PRAGMA_IGNORE_TKR are not available.
For F	Fortran $2008 + TS 29113$ and later and Fortran $2003 + TS 29113$ : major feature that are needed from TS 29113 are:
- 7 1 - 7	TYPE(*), DIMENSION() is available. The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI com- munication. 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.
Using	g these features, MPI-3.0 can be implemented without any restrictions.
	With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE The ASYNCHRONOUS at- tribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can be used for any Fortran type. With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation dependent. A high quality implementation will also provide MPI_SUBARRAYS_SUPPORTED==.TRUE. and will use the ASYNCHRONOUS attribute in the same way as in S1. If non-standardized extensions like !\$PRAGMA IGNORE_TKR are not available then S2 must be implemented with TYPE(*), DIMENSION().
argun	<i>ce to implementors.</i> If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice nent may be implemented with an explicit interface using compiler directives, cample:
II	NTERFACE SUBROUTINE MPI(buf,) !DEC\$ ATTRIBUTES NO_ARG_CHECK :: buf !\$PRAGMA IGNORE_TKR buf !DIR\$ IGNORE_TKR buf !IBM* IGNORE_TKR buf REAL, DIMENSION(*) :: buf ! declarations of the other arguments END SUBROUTINE

END INTERFACE

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $\overline{7}$ 

 $^{23}$  $^{24}$ 

#### 18.1.7 Requirements on Fortran Compilers

MPI-3.0 (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

Advice to users. Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI libraries are shipped together with special compilation scripts (e.g., mpif90, mpicc) that set these options automatically. (End of advice to users.)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by MPI\_GET\_VERSION, if all the solutions described in Sections 18.1.11 through 18.1.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the mpi\_f08 and mpi modules, and mpif.h) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 29113 [41] are available. This is required only for mpi\_f08. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the mpi\_f08 module with MPI\_SUBARRAYS\_SUPPORTED set to .FALSE..
- "Simply contiguous" arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the ASYNCHRONOUS attribute. See Section 18.1.12 for more details.
- SEQUENCE and BIND(C) derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI\_SUBARRAYS\_SUPPORTED== .FALSE., they are passed with call by reference, and passed by descriptor in the case of .TRUE..
- All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., CHARACTER(LEN=\*) strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods.
- The array dummy argument of the ISO\_C\_BINDING intrinsic module procedure C\_F\_POINTER is not restricted to Fortran types for which a corresponding type in C exists.
- The Fortran compiler shall not provide TYPE(\*) unless the ASYNCHRONOUS attribute protects MPI communication as described in TS 29113. Specifically, the TS 29113 must be implemented as a whole.

The following rules are required at least as long as the compiler does not provide the extension of the ASYNCHRONOUS attribute as part of TS 29113 and there still exists a Fortran support method with MPI\_ASYNC\_PROTECTS\_NONBLOCKING==.FALSE.. Observation of these rules by the MPI application developer is especially recomended for backward compatibility of existing applications that use the mpi module or the mpif.h include file. The rules are as follows:

 $\overline{7}$ 

 $^{24}$ 

1 2 3 4	• Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with BIND(C) Fortran interfaces (e.g., MPI_F_SYNC_REG on page 664 and Section 18.1.8, and DD on page 665) solve the problems described in Section 18.1.17.
5 6 7 8 9	• The problems with temporary data movement (described in detail in Section 18.1.18) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation.
10 11 12 13 14	• Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 18.1.19) are resolved <b>without</b> any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in invoking MPI procedures.
15 16	All of these rules are valid for the mpi_f08 and mpi modules and independently of whether mpif.h uses explicit interfaces.
17 18 19 20 21	Advice to implementors. Some of these rules are already part of the Fortran 2003 standard, some of these requirements require the Fortran TS 29113 [41], and some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (End of advice to implementors.)
22 23	18.1.8 Additional Support for Fortran Register-Memory-Synchronization
24 25 26 27	As described in Section 18.1.17, a dummy call may be necessary to tell the compiler that registers are to be flushed for a given buffer or that accesses to a buffer may not be moved across a given point in the execution sequence. Only a Fortran binding exists for this call.
28 29	MPI_F_SYNC_REG(buf)
30 31	INOUT     buf     initial address of buffer (choice)
32 33 34	<pre>MPI_F_sync_reg(buf)     TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf</pre>
35 36	<pre>MPI_F_SYNC_REG(buf)</pre>
37 38 39 40 41	This routine has no executable statements. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.
42 43 44 45	<i>Rationale.</i> This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. ( <i>End of rationale.</i> )
46 47 48	Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is

### Unofficial Draft for Comment Only

explicitly allowed to implement this routine within the mpi\_f08 module according to the definition for the mpi module or mpif.h to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (*End of advice to implementors.*)

*Rationale.* This routine is not defined with TYPE(\*), DIMENSION(\*), i.e., assumed size instead of assumed rank, because this would restrict the usability to "simply contiguous" arrays and would require overloading with another interface for scalar arguments. (*End of rationale.*)

Advice to users. If only a part of an array (e.g., defined by a subscript triplet) is used in a nonblocking routine, it is recommended to pass the whole array to MPI\_F\_SYNC\_REG anyway to minimize the overhead of this no-operation call. Note that this routine need not be called if MPI\_ASYNC\_PROTECTS\_NONBLOCKING is .TRUE. and the application fully uses the facilities of ASYNCHRONOUS arrays. (*End of advice to users*.)

#### 18.1.9 Additional Support for Fortran Numeric Intrinsic Types

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI\_INTEGER, MPI\_REAL, MPI\_INT, MPI\_DOUBLE, etc., as well as the optional types MPI\_REAL4, MPI\_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER) with an optional integer KIND parameter that selects from among one or more variants. The specific meaning of different KIND values themselves are implementation dependent and not specified by the language. Fortran provides the KIND selection functions selected\_real\_kind for REAL and COMPLEX types, and selected\_int\_kind for INTEGER types that allow users to declare variables with a minimum precision or number of digits. These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX, and INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two declarations are equivalent:

double precision x
real(KIND(0.0d0)) x

MPI provides two orthogonal methods for handling communication buffers of numeric intrinsic types. The first method (see the following section) can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained with the selected\_int\_kind or selected\_real\_kind functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion in heterogeneous environments. The second method (see "Support for size-specific MPI Datatypes" on page 647) gives the user complete control over communication by exposing machine representations.

46 47

48

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9 10

11

12

13

14

15

16 17

18 19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35 36

37

38

39

40

41

42

43

44

 $\frac{1}{2}$ 

#### 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  $\mathbf{p}$  is decimal digits of precision and  $\mathbf{r}$  is an exponent range. Implicitly 10 MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is 11 defined for each value of (p, r) supported by the compiler, including pairs for which one 12value is unspecified. Attempting to access an element of the array with an index (p, r) not 13 supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX 14 datatypes. For integers, there is a similar implicit array related to selected\_int\_kind and 15 indexed by the requested number of digits **r**. Note that the predefined datatypes contained 16 in these implicit arrays are not the same as the named MPI datatypes MPI\_REAL, etc., but 17a new set. 18

Advice to implementors. The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (End of advice to implementors.)

Advice to users. selected\_real\_kind() maps a large number of (p,r) pairs to a much smaller number of KIND parameters supported by the compiler. KIND parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and KIND parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same (p,r) value (REAL and COMPLEX) or r value (INTEGER). Thus MPI has many more datatypes than there are fundamental language types. (End of advice to users.)

31 32

19

20

21

22 23

24

25

26

27

28

29

30

```
33
34
```

MPI\_TYPE\_CREATE\_F90\_REAL(p, r, newtype)

35 36	IN	р	precision, in decimal digits (integer)
30 37	IN	r	decimal exponent range (integer)
38	OUT	newtype	the requested $MPI$ data type (handle)
39			
40	int MPI_T	<pre>ype_create_f90_real(int p</pre>	, int r, MPI_Datatype *newtype)
41			
42	MPI_Type_	<pre>create_f90_real(p, r, new</pre>	type, ierror)
43	INTEGER, INTENT(IN) :: p, r		
44	TYPE(	MPI_Datatype), INTENT(OUT	C) :: newtype
45	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror
46	MPI TYPE	CREATE_F90_REAL(P, R, NEW	TYPE. IERROR)
47		ER P, R, NEWTYPE, IERROR	,,
48		,,,,,	

This function returns a predefined MPI datatype that matches a REAL variable of KIND selected\_real\_kind(p, r). In the model described above it returns a handle for the element D(p, r). Either p or r may be omitted from calls to selected\_real\_kind(p, r) (but not both). Analogously, either p or r may be set to MPI\_UNDEFINED. In communication, an MPI datatype A returned by MPI\_TYPE\_CREATE\_F90\_REAL matches a datatype B if and only if B was returned by MPI\_TYPE\_CREATE\_F90\_REAL called with the same values for p and r or B is a duplicate of such a datatype. Restrictions on using the returned datatype with the "external32" data representation are given on page 647.

It is erroneous to supply values for p and r not supported by the compiler.

MPI_TYI	PE_CREATE_F90_C	OMPLEX(p, r, newtype)	11 12			
IN	р	precision, in decimal digits (integer)	13			
IN	r	decimal exponent range (integer)	14			
			15			
OUT	newtype	the requested $MPI$ data type (handle)	16			
			17			
int MPI	_Type_create_f90_	<pre>complex(int p, int r, MPI_Datatype *newtype)</pre>	18			
• -	e_create_f90_comp EGER, INTENT(IN)	lex(p, r, newtype, ierror)	19 20			
		INTENT(OUT) :: newtype	21			
	EGER, OPTIONAL, I		22			
			23			
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)						
	EGER P, R, NEWTYP	E, IERRUR	25 26			
COMPLEX calls to s to MPI_U the mate on using page 647	variable of KIND set elected_real_kind NDEFINED. Matchin hing rules for datat the returned datat	predefined MPI datatype that matches a lected_real_kind(p, r). Either p or r may be omitted from d(p, r) (but not both). Analogously, either p or r may be set g rules for datatypes created by this function are analogous to ypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions ype with the "external32" data representation are given on r values for p and r not supported by the compiler.	28 29 ) 30 31			
MPI_TY	PE_CREATE_F90_IN	ITEGER(r, newtype)	36			
IN	r	decimal exponent range, i.e., number of decimal digits	37			
	•	(integer)	30			
OUT	nowtypo		39			
001	newtype	the requested MPI datatype (handle)	40			
· · NDT	<b>T</b>		41 42			
int MPI.	_lype_create_190_	<pre>integer(int r, MPI_Datatype *newtype)</pre>	42			
MPI_Type	e_create_f90_inte	ger(r, newtype, ierror)	40			
	EGER, INTENT(IN)		45			
	INTENT(OUT) :: newtype	46				
INT	EGER, OPTIONAL, I	NTENT(OUT) :: ierror	47			
MPI TYP	1PI TYPE CREATE F90 INTEGER(R. NEWTYPE, IERROR)					

**Unofficial Draft for Comment Only** 

 $1 \\ 2$ 

 $\mathbf{5}$ 

1	INTEGER R, NEWTYPE, IERROR
2	This function returns a predefined MPI datatype that matches a INTEGER variable of
3 4	KIND selected_int_kind(r). Matching rules for datatypes created by this function are
5	analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL.
6	Restrictions on using the returned datatype with the "external32" data representation are
7	given on page 647.
8	It is erroneous to supply a value for r that is not supported by the compiler.
9	Example:
10	integer longtype, quadtype
11	integer, parameter :: long = selected_int_kind(15)
12	integer(long) ii(10)
13	real(selected_real_kind(30)) x(10)
14	call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
15 16	call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
17	
18	$\sim 11$ MDI GEND( $\dot{i}$ $\dot{i}$ $10$ $log structure )$
19	call MPI_SEND(ii, 10, longtype,) call MPI_SEND(x, 10, quadtype,)
20	call MPI_SEND(x, 10, quadtype,)
21	Advice to users. The datatypes returned by the above functions are predefined
22	datatypes. They cannot be freed; they do not need to be committed; they can be
23	used with predefined reduction operations. There are two situations in which they
24	behave differently syntactically, but not semantically, from the MPI named predefined
25 26	datatypes.
27	1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to
28	retrieve the values of <b>p</b> and <b>r</b> .
29	2. Because the datatypes are not named, they cannot be used as compile-time
30	initializers or otherwise accessed before a call to one of the
31	MPI_TYPE_CREATE_F90_XXX routines.
32 33	If a variable was declared specifying a non-default KIND value that was not obtained
33 34	with selected_real_kind() or selected_int_kind(), the only way to obtain a
35	matching MPI datatype is to use the size-based mechanism described in the next
36	section.
37	(End of advice to users.)
38	
39	Advice to implementors. An application may often repeat a call to
40	MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The appli-
41	cation is not allowed to free the returned predefined, unnamed datatype handles. To
42	prevent the creation of a potentially huge amount of handles, a high quality MPI imple-
43 44	mentation 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
44	to MPI_TYPE_CREATE_F90_XXX and using a hash table to find formerly generated
46	handles should limit the overhead of finding a previously generated datatype with
47	same combination of (XXX,p,r). ( <i>End of advice to implementors.</i> )
48	

*Rationale.* The MPI\_TYPE\_CREATE\_F90\_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 13.5.2) or user-defined (Section 13.5.3) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 13.5.2.

The external32 representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

The external32 representations of the datatypes returned by MPI\_TYPE\_CREATE\_F90\_REAL/COMPLEX/INTEGER are given by the following rules. For MPI\_TYPE\_CREATE\_F90\_REAL:

if	(r >	38)	then	external32 representati	on is undefined
else if	(r >	18)	then	external32_size = 16	
else if	(r >	9)	then	external32_size = 8	
else if	(r >	4)	then	external32_size = 4	
else if	(r >	2)	then	external32_size = 2	
else				external32_size = 1	

If the external32 representation of a datatype is undefined, the result of using the datatype directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) in operations that require the external32 representation is undefined. These operations include MPI\_PACK\_EXTERNAL, MPI\_UNPACK\_EXTERNAL, and many MPI\_FILE functions, when the "external32" data representation is used. The ranges for which the external32 representation is undefined are reserved for future standardization.

#### Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that contain explicit byte lengths — MPI\_REAL4, MPI\_INTEGER8, etc. This section describes a mechanism that generalizes this model to support all Fortran numeric intrinsic types.

We assume that for each **typeclass** (integer, real, complex) and each word size there is  $^{47}$  a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler,  $^{48}$ 

#### Unofficial Draft for Comment Only

 $\overline{7}$ 

 $41 \\ 42$ 

 $45 \\ 46$ 

1 MPI must provide a named size-specific datatype. The name of this datatype is of the form  $\mathbf{2}$ MPI\_<TYPE>n in C and Fortran where <TYPE> is one of REAL, INTEGER and COMPLEX, 3 and  $\mathbf{n}$  is the length in bytes of the machine representation. This datatype locally matches 4 all variables of type (typeclass, n) in Fortran. The list of names for such types includes: 5MPI\_REAL4 6 MPI\_REAL8 7 MPI\_REAL16 8 MPI\_COMPLEX8 9 MPI\_COMPLEX16 10 MPI\_COMPLEX32 11 MPI\_INTEGER1 12MPI\_INTEGER2 13 MPI\_INTEGER4 14 MPI\_INTEGER8 15MPI\_INTEGER16 1617 One datatype is required for each representation supported by the Fortran compiler. 18 19Particularly for the longer floating-point types, C and Fortran may use Rationale. 20different representations. For example, a Fortran compiler may define a 16-byte REAL 21type with 33 decimal digits of precision while a C compiler may define a 16-byte 22 long double type that implements an 80-bit (10 byte) extended precision floating point 23value. Both of these types are 16 bytes long, but they are not interoperable. Thus, 24these types are defined by Fortran, even though C may define types of the same length. 25(End of rationale.) 2627To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations REAL\*n, INTEGER\*n, always create a variable whose rep-28resentation is of size **n**. These datatypes may also be used for variables declared with 29KIND=INT8/16/32/64 or KIND=REAL32/64/128, which are defined in the ISO\_FORTRAN\_ENV 30  $^{31}$ intrinsic module. Note that the MPI datatypes and the REAL\*n. INTEGER\*n declarations count bytes whereas the Fortran KIND values count bits. All these datatypes are predefined. 32 33 The following functions allow a user to obtain a size-specific MPI datatype for any 34intrinsic Fortran type. 35 36 MPI\_SIZEOF(x, size) 37 38 IN a Fortran variable of numeric intrinsic type (choice) х 39 OUT size size of machine representation of that type (integer) 40 41 MPI\_Sizeof(x, size, ierror) 42TYPE(\*), DIMENSION(..) :: х 43 INTEGER, INTENT(OUT) :: size 44 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4546MPI\_SIZEOF(X, SIZE, IERROR) 47<type> X 48 INTEGER SIZE, IERROR

This function returns the size in bytes of the machine representation of the given variable. It is a generic Fortran routine and has a Fortran binding only.

Advice to users. This function is similar to the C size of operator but behaves slightly differently. If given an array argument, it returns the size of the base element, not the size of the whole array. (End of advice to users.)

*Rationale.* This function is not available in other languages because it would not be useful. (End of rationale.)

#### MPI\_TYPE\_MATCH\_SIZE(typeclass, size, datatype)

IN	typeclass	generic type specifier (integer)
IN	size	size, in bytes, of representation (integer)
OUT	datatype	datatype with correct type, size (handle)

int MPI\_Type\_match\_size(int typeclass, int size, MPI\_Datatype \*datatype)

<pre>MPI_Type_match_size(typeclass, size, datatype, ierror)</pre>
INTEGER, INTENT(IN) :: typeclass, size
TYPE(MPI_Datatype), INTENT(OUT) :: datatype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
    INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
```

typeclass is one of MPI\_TYPECLASS\_REAL, MPI\_TYPECLASS\_INTEGER and MPI\_TYPECLASS\_COMPLEX, corresponding to the desired **typeclass**. The function returns an MPI datatype matching a local variable of type (typeclass, size).

This function returns a reference (handle) to one of the predefined named datatypes, not a duplicate. This type cannot be freed. MPI\_TYPE\_MATCH\_SIZE can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling MPI\_SIZEOF in order to compute the variable size, and then calling MPI\_TYPE\_MATCH\_SIZE to find a suitable datatype. In C, one can use the C function sizeof(), instead of MPI\_SIZEOF. In addition, for variables of default kind the variable's size can be computed by a call to MPI\_TYPE\_GET\_EXTENT, if the typeclass is known. It is erroneous to specify a size not supported by the compiler.

*Rationale.* This is a convenience function. Without it, it can be tedious to find the correct named type. See note to implementors below. (End of rationale.)

Advice to implementors. This function could be implemented as a series of tests.

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
                                                                            44
{
  switch(typeclass) {
      case MPI_TYPECLASS_REAL: switch(size) {
        case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
```

**Unofficial Draft for Comment Only** 

1  $\mathbf{2}$ 

3

4

5

6 7

8

25

2627

28

29

30

31

32

33

34

35

36

37

38 39

40 41

4243

45

46

47

```
1
                    case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
2
                    default: error(...);
3
                  }
4
                  case MPI_TYPECLASS_INTEGER: switch(size) {
5
                     case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
6
                     case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
7
                     default: error(...);
8
                  }
9
                 ... etc. ...
10
              }
11
12
              return MPI_SUCCESS;
           }
13
14
           (End of advice to implementors.)
15
16
17
     Communication With Size-specific Types
18
     The usual type matching rules apply to size-specific datatypes: a value sent with datatype
19
     MPI_{TYPE>n} can be received with this same datatype on another process. Most modern
20
     computers use 2's complement for integers and IEEE format for floating point. Thus, com-
21
     munication using these size-specific datatypes will not entail loss of precision or truncation
22
     errors.
23
24
           Advice to users. Care is required when communicating in a heterogeneous environ-
25
           ment. Consider the following code:
26
27
           real(selected_real_kind(5)) x(100)
28
           call MPI_SIZEOF(x, size, ierror)
29
           call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
30
           if (myrank .eq. 0) then
31
                ... initialize x ...
32
                call MPI_SEND(x, xtype, 100, 1, ...)
33
           else if (myrank .eq. 1) then
34
                call MPI_RECV(x, xtype, 100, 0, ...)
35
           endif
36
37
           This may not work in a heterogeneous environment if the value of size is not the
38
           same on process 1 and process 0. There should be no problem in a homogeneous
39
           environment. To communicate in a heterogeneous environment, there are at least four
40
           options. The first is to declare variables of default type and use the MPI datatypes
41
           for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second
42
           is to use selected_real_kind or selected_int_kind and with the functions of the
43
           previous section. The third is to declare a variable that is known to be the same
44
           size on all architectures (e.g., selected_real_kind(12) on almost all compilers will
45
           result in an 8-byte representation). The fourth is to carefully check representation
46
           size before communication. This may require explicit conversion to a variable of size
47
           that can be communicated and handshaking between sender and receiver to agree on
48
           a size.
```

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
real(selected_real_kind(5)) x(100)
                                                                                          4
     call MPI_SIZEOF(x, size, ierror)
                                                                                          5
     call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
                                                                                          6
                                                                                          7
     if (myrank .eq. 0) then
        call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
                                                                        &
                                                                                          9
                              MPI_MODE_CREATE+MPI_MODE_WRONLY,
                                                                        &
                                                                                          10
                              MPI_INFO_NULL, fh, ierror)
                                                                                          11
        call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
                                                                                          12
                                  MPI_INFO_NULL, ierror)
                                                                                          13
                                                                                          14
        call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
                                                                                          15
        call MPI_FILE_CLOSE(fh, ierror)
     endif
                                                                                          16
                                                                                          17
                                                                                          18
     call MPI_BARRIER(MPI_COMM_WORLD, ierror)
                                                                                          19
     if (myrank .eq. 1) then
                                                                                          20
        call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,
                                                                                         21
                                                                           &
                        MPI_INFO_NULL, fh, ierror)
                                                                                         22
        call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
                                                                                         23
                                                                                          ^{24}
                                  MPI_INFO_NULL, ierror)
                                                                                          25
        call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
                                                                                          26
        call MPI_FILE_CLOSE(fh, ierror)
     endif
                                                                                         27
                                                                                          28
                                                                                         29
     If processes 0 and 1 are on different machines, this code may not work as expected if
                                                                                         30
     the size is different on the two machines. (End of advice to users.)
                                                                                          31
                                                                                          32
                                                                                         33
18.1.10
         Problems With Fortran Bindings for MPI
                                                                                         34
This section discusses a number of problems that may arise when using MPI in a Fortran
                                                                                         35
program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It
                                                                                         36
is intended to clarify, not add to, this standard.
                                                                                         37
    As noted in the original MPI specification, the interface violates the Fortran standard
                                                                                         38
in several ways. While these may cause few problems for Fortran 77 programs, they become
                                                                                         39
more significant for Fortran 90 programs, so that users must exercise care when using new
                                                                                          40
Fortran 90 features. With Fortran 2008 and the new semantics defined in TS 29113, most
                                                                                          41
violations are resolved, and this is hinted at in an addendum to each item. The violations
```

The following MPI features are inconsistent with Fortran 90 and Fortran 77.

of MPI. The rest of this section describes the potential problems in detail.

were originally adopted and have been retained because they are important for the usability

461. An MPI subroutine with a choice argument may be called with different argument 47types. When using the mpi\_f08 module together with a compiler that supports For-48 tran 2008 + TS 29113, this problem is resolved.

#### **Unofficial Draft for Comment Only**

1  $\mathbf{2}$ 

3

42

43

44

- 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(...).
- 3. Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied on entrance to or exit from the subroutine. This problem is solved with the use of the ASYNCHRONOUS attribute.
- 4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the ASYNCHRONOUS attribute as specified in TS 29113.
- 5. Several named "constants," such as MPI\_BOTTOM, MPI\_IN\_PLACE,
   MPI\_STATUS\_IGNORE, MPI\_STATUSES\_IGNORE, MPI\_ERRCODES\_IGNORE,
   MPI\_UNWEIGHTED, MPI\_WEIGHTS\_EMPTY, MPI\_ARGV\_NULL, and MPI\_ARGVS\_NULL
   are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 for more information.
- 19 6. The memory allocation routine MPI\_ALLOC\_MEM cannot be used from 20Fortran 77/90/95 without a language extension (for example, Cray pointers) that 21allows the allocated memory to be associated with a Fortran variable. Therefore, 22 address sized integers were used in MPI-2.0 – MPI-2.2. In Fortran 2003, 23TYPE(C\_PTR) entities were added, which allow a standard-conforming implementation 24of the semantics of MPI\_ALLOC\_MEM. In MPI-3.0 and later, MPI\_ALLOC\_MEM has 25an additional, overloaded interface to support this language feature. The use of Cray 26pointers is deprecated. The mpi\_f08 module only supports TYPE(C\_PTR) pointers. 27
  - Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.
    - MPI identifiers exceed 6 characters.
    - MPI identifiers may contain underscores after the first character.
    - MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.
    - Many routines in MPI have KIND-parameterized integers (e.g., MPI\_ADDRESS\_KIND and MPI\_OFFSET\_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, INTEGER\*8 or INTEGER should be used instead.
- <sup>40</sup> MPI-1 contained several routines that take address-sized information as input or return <sup>41</sup> address-sized information as output. In C such arguments were of type MPI\_Aint and in <sup>42</sup> Fortran of type INTEGER. On machines where integers are smaller than addresses, these <sup>43</sup> routines can lose information. In MPI-2 the use of these functions has been deprecated and <sup>44</sup> they have been replaced by routines taking INTEGER arguments of KIND=MPI\_ADDRESS\_KIND. <sup>45</sup> A number of new MPI-2 functions also take INTEGER arguments of non-default KIND. See <sup>46</sup> Section 2.6 and Section 4.1.1 for more information.
- 47 Sections 18.1.11 through 18.1.19 describe several problems in detail which concern
   48 the interaction of MPI and Fortran as well as their solutions. Some of these solutions

#### Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8 9

10

11

12

13

28

29

30 31

32

33 34

35

36

37

38

require special capabilities from the compilers. Major requirements are summarized in Section 18.1.7.

## 18.1.11 Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 18.1.6). In C, the use of void\* formal arguments avoids these problems. Similar to C, with Fortran 2008 + TS 29113 (and later) together with the mpi\_f08 module, the problem is avoided by declaring choice arguments with TYPE(\*), DIMENSION(...), i.e., as assumed-type and assumed-rank dummy arguments.

Using INCLUDE 'mpif.h', the following code fragment is technically invalid and may generate a compile-time error.

```
integer i(5)
real x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

In practice, it is rare for compilers to do more than issue a warning. When using either the mpi\_f08 or mpi module, the problem is usually resolved through the assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the mpi\_f08 or mpi module, the following code fragment usually generates an error since the dims and periods arguments to MPI\_CART\_CREATE are declared as assumed size arrays INTEGER :: DIMS(\*) and LOGICAL :: PERIODS(\*).

```
USE mpi_f08 ! or USE mpi
INTEGER size
CALL MPI_Cart_create( comm_old,1,size,.TRUE.,.TRUE.,comm_cart,ierror )
```

Although this is a non-conforming MPI call, compiler warnings are not expected (but may occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit interfaces.

18.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets

Arrays with subscript **triplets** describe Fortran subarrays with or without strides, e.g.,

REAL a(100,100,100) CALL MPI\_Send( a(11:17, 12:99:3, 1:100), 7\*30\*100, MPI\_REAL, ...)

The handling of subscript triplets depends on the value of the constant MPI\_SUBARRAYS\_SUPPORTED:

 $\mathbf{2}$ 

 $16 \\ 17$ 

 $^{24}$ 

1	• If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:
2	Choice buffer arguments are declared as TYPE(*), DIMENSION(). For example,
3	consider the following code fragment:
4	
5	REAL s(100), r(100)
6 7	CALL MPI_Isend(s(1:100:5), 3, MPI_REAL,, rq, ierror)
8	CALL MPI_Wait(rq, status, ierror)
9	CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL,, rq, ierror)
10	CALL MPI_Wait(rq, status, ierror)
11	
12	In this case, the individual elements $s(1)$ , $s(6)$ , and $s(11)$ are sent between the start
13	of $MPI\_ISEND$ and the end of $MPI\_WAIT$ even though the compiled code will not copy
14	${\tt s(1:100:5)}$ to a real contiguous temporary scratch buffer. Instead, the compiled code
15	will pass a descriptor to $MPI_ISEND$ that allows $MPI$ to operate directly on $s(1)$ , $s(6)$ ,
16	s(11),, s(96). The called MPI_ISEND routine will take only the first three of these
17	elements due to the type signature "3, MPI_REAL".
18	All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT,
19	MPI_FILE_WRITE_ALL_BEGIN) behave as if the user-specified elements of choice
20	buffers are copied to a contiguous scratch buffer in the MPI runtime environment.
21 22	All datatype descriptions (in the example above, "3, MPI_REAL") read and store
23	data from and to this virtual contiguous scratch buffer. Displacements in MPI de-
24	rived 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 cor-
25	responding MPI_Request returns), it is as if the received data has been copied from
26	the virtual contiguous scratch buffer back to the non-contiguous application buffer.
27	In the example above, $r(1)$ , $r(6)$ , and $r(11)$ are guaranteed to be defined with the
28	received data when MPI_WAIT returns.
29	Note that the above definition does not supercede restrictions about buffers used with
30	non-blocking operations (e.g., those specified in Section 3.7.2).
31	
32	Advice to implementors. The Fortran descriptor for $TYPE(*)$ , DIMENSION()
33 34	arguments contains enough information that, if desired, the $MPI$ library can make
35	a real contiguous copy of non-contiguous user buffers when the nonblocking op-
36	eration is started, and release this buffer not before the nonblocking communi-
37	cation has completed (e.g., the MPI_WAIT routine). Efficient implementations
38	may avoid such additional memory-to-memory data copying. (End of advice to
39	implementors.)
40	Rationale. If MPI_SUBARRAYS_SUPPORTED equals .TRUE., non-contiguous
41	buffers are handled inside the MPI library instead of by the compiler through
42	argument association conventions. Therefore, the scope of MPI library scratch
43	buffers can be from the beginning of a nonblocking operation until the completion
44	of the operation although beginning and completion are implemented in different
45	routines. (End of rationale.)
46	
47 48	• If MPI_SUBARRAYS_SUPPORTED equals .FALSE.:
40	

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>]... )

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $\overline{7}$ 

 $^{31}$ 

 $45 \\ 46$ 

<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.

	050	CHAI TER 16. LANGUAGE DINDINGS
1 2		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
3 4 5		subscript. The compiler can detect from analyzing the source code that the array is contiguous. Examples are
6 7		A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
8 9		Because of Fortran's column-major ordering, where the first index varies fastest, a "simply contiguous" section of a contiguous array will also be contiguous.
10 11 12 13 14		The same problem can occur with a scalar argument. A compiler may make a copy of scalar dummy arguments within a called procedure when passed as an actual argument to a choice buffer routine. That this can cause a problem is illustrated by the example
15 16 17 18 19		real :: a call user1(a,rq) call MPI_WAIT(rq,status,ierr) write (*,*) a
20 21 22 23 24		<pre>subroutine user1(buf,request) call MPI_IRECV(buf,,request,) end</pre>
24 25 26		If a is copied, MPI_IRECV will alter the copy when it completes the communication and will not alter a itself.
27 28 29 30 31		Note that copying will almost certainly occur for an argument that is a non-trivial expression (one with at least one operator or function call), a section that does not select a contiguous part of its parent (e.g., $A(1:n:2)$ ), a pointer whose target is such a section, or an assumed-shape array that is (directly or indirectly) associated with such a section.
32 33		If a compiler option exists that inhibits copying of arguments, in either the calling or called procedure, this must be employed.
34 35 36 37 38 39 40 41		If a compiler makes copies in the calling procedure of arguments that are explicit- shape or assumed-size arrays, "simply contiguous" array sections of such arrays, or scalars, and if no compiler option exists to inhibit such copying, then the compiler cannot be used for applications that use MPI_GET_ADDRESS, or any nonblocking MPI routine. If a compiler copies scalar arguments in the called procedure and there is no compiler option to inhibit this, then this compiler cannot be used for applications that use memory references across subroutine calls as in the example above.
42 43	18.1	.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts
44 45 46		ran arrays with <b>vector</b> subscripts describe subarrays containing a possibly irregular f elements
47 48		EAL a(100) CALL MPI_Send( A((/7,9,23,81,82/)), 5, MPI_REAL,)

CHAPTER 18. LANGUAGE BINDINGS

Fortran arrays with a vector subscript must not be used as actual choice buffer arguments in any nonblocking or split collective MPI operations. They may, however, be used in blocking MPI operations.

### 18.1.14 Special Constants

MPI requires a number of special "constants" that cannot be implemented as normal Fortran constants, e.g., MPI\_BOTTOM. The complete list can be found in Section 2.5.4. In C, these are implemented as constant pointers, usually as NULL and are used where the function prototype calls for a pointer to a variable, not the variable itself.

In Fortran, using special values for the constants (e.g., by defining them through **parameter** statements) is not possible because an implementation cannot distinguish these values from valid data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, the address of the actual choice buffer argument can be compared with the address of such a predefined static variable.

These special constants also cause an exception with the usage of Fortran INTENT: with USE mpi\_f08, the attributes INTENT(IN), INTENT(OUT), and INTENT(INOUT) are used in the Fortran interface. In most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for dummy arguments that may be modified and allow one of these special constants as input, an INTENT is not specified.

### 18.1.15 Fortran Derived Types

MPI supports passing Fortran entities of BIND(C) and SEQUENCE derived types to choice dummy arguments, provided no type component has the ALLOCATABLE or POINTER attribute.

The following code fragment shows some possible ways to send scalars or arrays of interoperable derived type in Fortran. The example assumes that all data is passed by address.

```
type, BIND(C) :: mytype
   integer :: i
   real :: x
   double precision :: d
   logical :: 1
end type mytype
type(mytype) :: foo, fooarr(5)
integer :: blocklen(4), type(4)
integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent
call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
call MPI_GET_ADDRESS(foo%1, disp(4), ierr)
base = disp(1)
disp(1) = disp(1) - base
disp(2) = disp(2) - base
```

```
1
          disp(3) = disp(3) - base
\mathbf{2}
          disp(4) = disp(4) - base
3
4
          blocklen(1) = 1
5
          blocklen(2) = 1
6
          blocklen(3) = 1
7
          blocklen(4) = 1
8
9
          type(1) = MPI_INTEGER
10
          type(2) = MPI_REAL
11
          type(3) = MPI_DOUBLE_PRECISION
12
          type(4) = MPI_LOGICAL
13
14
          call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
15
          call MPI_TYPE_COMMIT(newtype, ierr)
16
17
          call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
18
          ! or
19
          call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
20
          ! expects that base == address(foo%i) == address(foo)
21
22
          call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
23
          call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
24
          extent = disp(2) - disp(1)
25
          1b = 0
26
          call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
27
          call MPI_TYPE_COMMIT(newarrtype, ierr)
28
29
          call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
30
          Using the derived type variable foo instead of its first basic type element foo%i may
^{31}
     be impossible if the MPI library implements choice buffer arguments through overloading
32
     instead of using TYPE(*), DIMENSION(..), or through a non-standardized extension such
33
     as !$PRAGMA IGNORE_TKR; see Section 18.1.6.
34
         To use a derived type in an array requires a correct extent of the datatype handle
35
     to take care of the alignment rules applied by the compiler. These alignment rules may
36
     imply that there are gaps between the components of a derived type, and also between the
37
     subsuguent elements of an array of a derived type. The extent of an interoperable derived
38
     type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may
39
     be different because C and Fortran may apply different alignment rules. As recommended
40
     in the advice to users in Section 4.1.6, one should add an additional fifth structure element
41
     with one numerical storage unit at the end of this structure to force in most cases that
42
     the array of structures is contiguous. Even with such an additional element, one should
43
     keep this resizing due to the special alignment rules that can be used by the compiler for
44
     structures, as also mentioned in this advice.
45
          Using the extended semantics defined in TS 29113, it is also possible to use entities
46
```

or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer
 arguments; some additional constraints must be observed, e.g., no ALLOCATABLE or POINTER

### Unofficial Draft for Comment Only

type components may exist. In this case, the **base** address in the example must be changed to become the address of **foo** instead of **foo%i**, because the Fortran compiler may rearrange type components or add padding. Sending the structure **foo** should then also be performed by providing it (and not **foo%i**) as actual argument for MPI\_Send.

### 18.1.16 Optimization Problems, an Overview

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI\_IRECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. These problems are independent of the Fortran support method; i.e., they occur with the mpi\_f08 module, the mpi module, and the mpif.h include file.

This section shows four problematic usage areas (the abbreviations in parentheses are used in the table below):

- Use of nonblocking routines or persistent requests (Nonbl.).
- Use of one-sided routines (1-sided).
- Use of MPI parallel file I/O split collective operations (Split).
- Use of MPI\_BOTTOM together with absolute displacements in MPI datatypes, or relative displacements between two variables in such datatypes (*Bottom*).

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 18.1.17.
- Temporary data movement and temporary memory modifications; see Section 18.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 18.1.19.

Table 18.2 shows the only usage areas where these optimization problems may occur.

Optimization	may cause a problem in following usage areas			
		1-sided	0	
Code movement	yes	yes	no	yes
and register optimization				
Temporary data movement	yes	yes	yes	no
Permanent data movement	yes	yes	yes	yes

Table 18.2: Occurrence of Fortran optimization problems in several usage areas The solutions in the following sections are based on compromises:

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

14

15

16

17 18

19

20 21

22 23

24

25 26

27 28

29 30

 $^{31}$ 

32 33

 $45 \\ 46$ 

1 • to minimize the burden for the application programmer, e.g., as shown in Sections  $\mathbf{2}$ "Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on 3 pages 661–666, 4 • to minimize the drawbacks on compiler based optimization, and 56 • to minimize the requirements defined in Section 18.1.7. 7 8 18.1.17 Problems with Code Movement and Register Optimization 9 10 Nonblocking Operations 11 If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the 12compiler will assume that it cannot be modified by a called subroutine unless it is an actual 13 argument of the call. In the most common linkage convention, the subroutine is expected 14 to save and restore certain registers. Thus, the optimizer will assume that a register which 15held a valid copy of such a variable before the call will still hold a valid copy on return. 1617**Example 18.1** Fortran 90 register optimization — extreme. 18 19Source compiled as or compiled as 20REAL :: buf, b1 21REAL :: buf, b1 REAL :: buf, b1 call MPI\_IRECV(buf,..req) call MPI\_IRECV(buf,..req) call MPI\_IRECV(buf,..req) 22 register = buf b1 = buf23call MPI\_WAIT(req,..) call MPI\_WAIT(req,..) call MPI\_WAIT(req,..)  $^{24}$ b1 = bufb1 = register 2526Example 18.1 shows extreme, but allowed, possibilities. MPI\_WAIT on a concurrent 27thread modifies buf between the invocation of MPI\_IRECV and the completion of MPI\_WAIT. 28But the compiler cannot see any possibility that buf can be changed after MPI\_IRECV has 29returned, and may schedule the load of buf earlier than typed in the source. The compiler 30 has no reason to avoid using a register to hold **buf** across the call to MPI\_WAIT. It also may  $^{31}$ reorder the instructions as illustrated in the rightmost column. 32 33 34Example 18.2 Similar example with MPI\_ISEND 35 Source compiled as with a possible MPI-internal 36 execution sequence 37 38REAL :: buf, copy REAL :: buf, copy REAL :: buf, copy 39 buf = val buf = val buf = val call MPI\_ISEND(buf,..req) addr = &buf call MPI\_ISEND(buf,..req) 40copy = buf copy = bufcopy= buf 41 buf = val\_overwrite buf = val\_overwrite 42call MPI\_WAIT(req,..) call MPI\_WAIT(req,..) call send(\*addr) ! within 43 ! MPI\_WAIT 44buf = val\_overwrite 4546

<sup>47</sup> Due to valid compiler code movement optimizations in Example 18.2, the content of <sup>48</sup> buf may already have been overwritten by the compiler when the content of buf is sent.

### Unofficial Draft for Comment Only

The code movement is permitted because the compiler cannot detect a possible access to buf in MPI\_WAIT (or in a second thread between the start of MPI\_ISEND and the end of MPI\_WAIT).

Such register optimization is based on moving code; here, the access to buf was moved from after MPI\_WAIT to before MPI\_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined.

This register optimization/code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the ...\_BEGIN and ...\_END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI\_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication call, as well as in each parallel file I/O operation.

### Persistent Operations

With persistent requests, the buffer argument is hidden from the MPI\_START and MPI\_STARTALL calls, i.e., the Fortran compiler may move buffer accesses across the MPI\_START or MPI\_STARTALL call, similar to the MPI\_WAIT call as described in the Nonblocking Operations subsection in Section 18.1.17.

### **One-sided** Communication

An example with instruction reordering due to register optimization can be found in Section 11.7.4.

#### MPI\_BOTTOM and Combining Independent Variables in Datatypes

This section is only relevant if the MPI program uses a buffer argument to an MPI\_SEND, MPI\_RECV, etc., that hides the actual variables involved in the communication. MPI\_BOTTOM with an MPI\_Datatype containing *absolute addresses* is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI\_GET\_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 18.3 shows what Fortran compilers are allowed to do.

In Example 18.3, the compiler does not invalidate the register because it cannot see that MPI\_RECV changes the value of buf. The access to buf is hidden by the use of MPI\_GET\_ADDRESS and MPI\_BOTTOM.

In Example 18.4, several successive assignments to the same variable buf can be combined in a way such that only the last assignment is executed. "Successive" means that no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to MPI\_SEND statement is interfering because the load access to buf is hidden by the usage of MPI\_BOTTOM.

#### Solutions

The following sections show in detail how the problems with code movement and register optimization can be portably solved. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8 9

10 11

12

13 14

15

16

17

18

19 20

21 22

23

24 25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

42 43 44

45

46

47

```
1
     Example 18.3 Fortran 90 register optimization.
\mathbf{2}
3
     This source ...
                                                    can be compiled as:
4
     call MPI_GET_ADDRESS(buf, bufaddr,
                                                   call MPI_GET_ADDRESS(buf,...)
5
                      ierror)
6
     call MPI_TYPE_CREATE_STRUCT(1,1,
                                                   call MPI_TYPE_CREATE_STRUCT(...)
7
                      bufaddr,
8
                      MPI_REAL, type, ierror)
9
     call MPI_TYPE_COMMIT(type,ierror)
                                                   call MPI_TYPE_COMMIT(...)
10
     val_old = buf
                                                   register = buf
11
                                                   val_old = register
12
     call MPI_RECV(MPI_BOTTOM,1,type,...)
                                                   call MPI_RECV(MPI_BOTTOM,...)
13
     val_new = buf
                                                   val_new = register
14
15
16
     Example 18.4 Similar example with MPI_SEND
17
18
     This source ...
                                                    can be compiled as:
19
     ! buf contains val_old
                                                   ! buf contains val_old
20
21
     buf = val_new
     call MPI_SEND(MPI_BOTTOM,1,type,...)
                                                   call MPI_SEND(...)
22
     ! with buf as a displacement in type
                                                   ! i.e. val_old is sent
23
                                                   ŗ
^{24}
                                                   ! buf=val_new is moved to here
25
26
                                                   ! and detected as dead code
                                                   ! and therefore removed
27
                                                   L
28
     buf = val_overwrite
                                                   buf = val_overwrite
29
30
^{31}
     with the send and receive buffers used in nonblocking operations, or in operations in which
32
     MPI_BOTTOM is used, or if datatype handles that combine several variables are used:
33
34
         • Use of the Fortran ASYNCHRONOUS attribute.
35
36
         • Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy
37
           routine.
38
         • Declare the buffer as a Fortran module variable or within a Fortran common block.
39
40
         • Use of the Fortran VOLATILE attribute.
41
42
          Each of these methods solves the problems of code movement and register optimization,
43
     but may incur various degrees of performance impact, and may not be usable in every
44
     application context. These methods may not be guaranteed by the Fortran standard, but
45
     they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated
46
     compiler suite according to the requirements listed in Section 18.1.7. The performance
47
     impact of using MPI_F_SYNC_REG is expected to be low, that of using module variables
48
     or the ASYNCHRONOUS attribute is expected to be low to medium, and that of using the
```

### Unofficial Draft for Comment Only

VOLATILE attribute is expected to be high or very high. Note that there is one attribute that cannot be used for this purpose: the Fortran TARGET attribute does not solve code movement problems in MPI applications.

### The Fortran ASYNCHRONOUS Attribute

Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed while the buffer is affected by a pending asynchronous Fortran input/output operation (since Fortran 2003) or by an asynchronous communication (TS 29113 extension). Without the extensions specified in TS 29113, a Fortran compiler may totally ignore this attribute if the Fortran compiler implements asynchronous Fortran input/output operations with blocking I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through code movements across routine calls, and the buffer itself from temporary and permanent data movements. If the choice buffer dummy argument of a nonblocking MPI routine is declared with ASYNCHRONOUS (which is mandatory for the mpi\_f08 module, with allowable exceptions listed in Section 18.1.6), then the compiler has to guarantee call by reference and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The MPI\_ASYNC\_PROTECTS\_NONBLOCKING is set to .TRUE. if both the protection of the actual buffer argument through ASYNCHRONOUS according to the TS 2029113 extension and the declaration of the dummy argument with ASYNCHRONOUS in the 21Fortran 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."

42In Example 18.5 Case (a) on page 669, 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 43 44b 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 4546communication affector and is usable for input and output asynchronous communication 47between the MPI\_I... routines and MPI\_Waitall. Case (a) works fine because the read 48 accesses to b occur after the communication has completed.

# **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

22

23

 $^{24}$ 

2526

27

28

29

30

31

32

33

34

35

36

37

38

39

40

1 In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to  $\mathbf{2}$ a pending communication affector while input communication (i.e., the two MPI\_Irecv 3 calls) is pending. This is a contradiction to the rule that for input communication, a 4 pending communication affector shall not be referenced. The problem can be solved by using  $\mathbf{5}$ separate variables for the halos and the inner array, or by splitting a common array into 6 disjoint subarrays which are passed through different dummy arguments into a subroutine,  $\overline{7}$ as shown in Example 18.9.

If one does not overlap communication and computation on the same variable, then all optimization problems can be solved through the ASYNCHRONOUS attribute.

The problems with MPI\_BOTTOM, as shown in Example 18.3 and Example 18.4, can also be solved by declaring the buffer **buf** with the **ASYNCHRONOUS** attribute.

In some MPI routines, a buffer dummy argument is defined as ASYNCHRONOUS to guarantee passing by reference, provided that the actual argument is also defined as ASYNCHRONOUS.

13 14

16

21

22

34

35

36

37 38

39

40

41

48

8

9

10

11

12

15Calling MPI\_F\_SYNC\_REG

The compiler may be prevented from moving a reference to a buffer across a call to an 17MPI subroutine by surrounding the call by calls to an external subroutine with the buffer 18 as an actual argument. The MPI library provides the MPI\_F\_SYNC\_REG routine for this 19 purpose; see Section 18.1.8. 20

• The problems illustrated by the Examples 18.1 and 18.2 can be solved by calling MPI\_F\_SYNC\_REG(buf) once immediately after MPI\_WAIT.

Example 18.1 can be solved with call MPI_IRECV(buf,req)	Example 18.2 can be solved with buf = val call MPI_ISEND(buf,req)
<pre>call MPI_WAIT(req,) call MPI_F_SYNC_REG(buf) b1 = buf</pre>	<pre>copy = buf call MPI_WAIT(req,) call MPI_F_SYNC_REG(buf) buf = val_overwrite</pre>

The call to MPI\_F\_SYNC\_REG(buf) prevents moving the last line before the MPI\_WAIT call. Further calls to MPI\_F\_SYNC\_REG(buf) are not needed because it is still correct if the additional read access copy=buf is moved below MPI\_WAIT and before buf=val\_overwrite.

• The problems illustrated by the Examples 18.3 and 18.4 can be solved with two additional MPI\_F\_SYNC\_REG(buf) statements; one directly before MPI\_RECV/ MPI\_SEND, and one directly after this communication operation.

42	Example 18.3	Example 18.4
43	can be solved with	can be solved with
44	call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
45	<pre>call MPI_RECV(MPI_BOTTOM,)</pre>	<pre>call MPI_SEND(MPI_BOTTOM,)</pre>
46	call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
47		

The first call to MPI\_F\_SYNC\_REG(buf) is needed to finish all load and store references to buf prior to MPI\_RECV/MPI\_SEND; the second call is needed to assure that any subsequent access to buf is not moved before MPI\_RECV/SEND.

• In the example in Section 11.7.4, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 18.1, i.e., a call to MPI\_F\_SYNC\_REG(bbbb) is needed after the second MPI\_WIN\_FENCE to guarantee that further accesses to bbbb are not moved ahead of the call to MPI\_WIN\_FENCE. In Process 2, both calls to MPI\_WIN\_FENCE together act as a communication call with MPI\_BOTTOM as the buffer. That is, before the first fence and after the second fence, a call to MPI\_F\_SYNC\_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI\_WIN\_FENCE. Using MPI\_GET instead of MPI\_PUT, the same calls to MPI\_F\_SYNC\_REG are necessary.

Source of Process 1	Source of Process 2	15
bbbb = 777	buff = 999	16
	call MPI_F_SYNC_REG(buff)	17
call MPI_WIN_FENCE	call MPI_WIN_FENCE	18
call MPI_PUT(bbbb		19
into buff of process 2)		20
		21
call MPI_WIN_FENCE	call MPI_WIN_FENCE	22
call MPI_F_SYNC_REG(bbbb)	call MPI_F_SYNC_REG(buff)	23
	ccc = buff	24
		~ ~

- The temporary memory modification problem, i.e., Example 18.6, can **not** be solved with this method.
- A User Defined Routine Instead of MPI\_F\_SYNC\_REG

Instead of MPI\_F\_SYNC\_REG, one can also use a user defined external subroutine, which is separately compiled:

> subroutine DD(buf) integer buf end

Note that if the intent is declared in an explicit interface for the external subroutine, it must be OUT or INOUT. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, a call to MPI\_RECV with MPI\_BOTTOM as buffer might be replaced by

```
call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
```

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document 4647such usage in existing application programs although new applications should prefer 48 MPI\_F\_SYNC\_REG or one of the other possibilities. In an existing application, calls to

### **Unofficial Draft for Comment Only**

1 2

3

4

5

6

7

8

9

10

11

12

242526

27

2829

30

31

32 33

34

35

36 37

38

39

40

41 42

43

44

<sup>1</sup> such a user-written routine should be substituted by a call to MPI\_F\_SYNC\_REG because <sup>2</sup> the user-written routine may not be implemented in accordance with the rules specified in

- the user-written routine may not be implemented in accordance with the rules specified in
   Section 18.1.7.
- 4 5

6

13

14

15

16 17

18

# Module Variables and COMMON Blocks

An alternative to the previously mentioned methods is to put the buffer or variable into a module or a common block and access it through a USE or COMMON statement in each scope where it is referenced, defined or appears as an actual argument in a call to an MPI routine. The compiler will then have to assume that the MPI procedure may alter the buffer or variable, provided that the compiler cannot infer that the MPI procedure does not reference the module or common block.

- This method solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI\_BOTTOM and derived datatype handles.
- Unfortunately, this method does **not** solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.
- 19 20 21

22

# The (Poorly Performing) Fortran VOLATILE Attribute

The VOLATILE attribute gives the buffer or variable the properties needed to avoid register optimization or code movement problems, but it may inhibit optimization of any code containing references or definitions of the buffer or variable. On many modern systems, the performance impact will be large because not only register, but also cache optimizations will not be applied. Therefore, use of the VOLATILE attribute to enforce correct execution of MPI programs is discouraged.

29 30

# The Fortran TARGET Attribute

The TARGET attribute does not solve the code movement problem because it is not specified for the choice buffer dummy arguments of nonblocking routines. If the compiler detects that the application program specifies the TARGET attribute for an actual buffer argument used in the call to a nonblocking routine, the compiler may ignore this attribute if no pointer reference to this buffer exists.

*Rationale.* The Fortran standardization body decided to extend the ASYNCHRONOUS attribute within the TS 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the TARGET attribute. (*End of rationale.*)

39 40 41

42

37

38

# 18.1.18 Temporary Data Movement and Temporary Memory Modification

The compiler is allowed to temporarily modify data in memory. Normally, this problem may occur only when overlapping communication and computation, as in Example 18.5, Case (b) on page 669. Example 18.6 also shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 18.7, buf(100,100) from Example 18.6 is equivalenced with the 1-dimensional array buf\_1dim(10000). The nonblocking receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused loop is temporarily using this part of the buffer. When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

Example 18.8 shows a second possible optimization. The whole array is temporarily moved to local\_buf.

When storing local\_buf back to the original location buf, then this implies overwriting the section of buf that serves as a receive buffer in the nonblocking MPI call, i.e., this storing back of local\_buf is therefore likely to interfere with asynchronously received data in buf(1,1:100).

Note that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 11.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations between the ...\_BEGIN and ...\_END calls; see Section 13.4.5.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 663 of Section 18.1.17, the ASYNCHRONOUS attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the local references are separated into different variables, as shown in Example 18.9 and in Example 18.10.

Note also that the methods

- calling MPI\_F\_SYNC\_REG (or such a user-defined routine),
- using module variables and COMMON blocks, and
- the TARGET attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the code fragments shown in Example 18.6 and 18.7.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring **buf** as **VOLATILE** because the **VOLATILE** implies that all accesses to any storage unit (word) of **buf** must be directly done in the main memory exactly in the sequence defined by the application program. The **VOLATILE** attribute prevents all register and cache optimizations. Therefore, **VOLATILE** may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the ASYNCHRONOUS attribute. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

Rationale. This is a strong restriction for application programs. To weaken this <sup>46</sup> restriction, a new or modified asynchronous feature in the Fortran language would <sup>47</sup> be necessary: an asynchronous attribute that can be used on parts of an array and <sup>48</sup>

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $\overline{7}$ 

 $^{24}$ 

together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (End of rationale.)

In Example 18.9 (which is a solution for the problem shown in Example 18.5 and 5in Example 18.10 (which is a solution for the problem shown in Example 18.8), the ar-6 ray is split into inner and halo part and both disjoint parts are passed to a subroutine separated\_sections. This routine overlaps the receiving of the halo data and the calcu-8 lations on the inner part of the array. In a second step, the whole array is used to do the 9 calculation on the elements where inner+halo is needed. Note that the halo and the inner 10 area are strided arrays. Those can be used in non-blocking communication only with a TS 11 29113 based MPI library. 12

13 14

15

16

17

18

1920

21

22

23

 $^{24}$ 25

26

27

28

# 18.1.19 Permanent Data Movement

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. An implementation with automatic garbage collection is one use case. Such permanent data movement is in conflict with MPI in several areas:

- MPI datatype handles with absolute addresses in combination with MPI\_BOTTOM.
- All nonblocking MPI operations if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

This problem can be also solved by using the ASYNCHRONOUS attribute for such buffers. This MPI standard requires that the problems with permanent data movement do not occur by imposing suitable restrictions on the MPI library together with the compiler used; see Section 18.1.7.

2930  $^{31}$ 

#### Comparison with C 18.1.20

32 In C, subroutines which modify variables that are not in the argument list will not cause 33 register optimization problems. This is because taking pointers to storage objects by using 34the & operator and later referencing the objects by indirection on the pointer is an integral 35 part of the language. A C compiler understands the implications, so that the problem should 36 not occur, in general. However, some compilers do offer optional aggressive optimization 37 levels which may not be safe. Problems due to temporary memory modifications can also 38 occur in C. As above, the best advice is to avoid the problem: use different variables for 39 buffers in nonblocking MPI operations and computation that is executed while a nonblocking 40operation is pending.

- 41
- 42
- 43
- 4445
- 46
- 47
- 48

1

2

3

4

Example 18.5 Protecting nonblocking communication with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)
                               ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b( 0), ..., left, ..., req(1), ...)
CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
CALL MPI_Isend(b( 1), ..., left, ..., req(3), ...)
CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)
#ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (a)
  CALL MPI_Waitall(4,req,...)
  DO i=1,100 ! compute all new local data
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
#endif
#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (b)
  DO i=2,99 ! compute only elements for which halo data is not needed
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
  CALL MPI_Waitall(4,req,...)
  i=1 ! compute leftmost element
    bnew(i) = function(b(i-1), b(i), b(i+1))
  i=100 ! compute rightmost element
    bnew(i) = function(b(i-1), b(i), b(i+1))
#endif
Example 18.6 Overlapping Communication and Computation.
USE mpi_f08
REAL :: buf(100,100)
CALL MPI_Irecv(buf(1,1:100),...req,...)
DO j=1,100
  DO i=2,100
    buf(i,j)=...
  END DO
END DO
CALL MPI_Wait(req,...)
```

1 2

3 4

5

6

7

8

9

10

11

12

13

 $14 \\ 15$ 

16

17

18

19

20

21

22 23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

38

39

40

41

42

43

44

45

46

```
1
\mathbf{2}
3
4
5
6
     Example 18.7 The compiler may substitute the nested loops through loop fusion.
\overline{7}
8
     REAL :: buf(100,100), buf_1dim(10000)
9
     EQUIVALENCE (buf(1,1), buf_1dim(1))
10
     CALL MPI_Irecv(buf(1,1:100),...req,...)
11
     tmp(1:100) = buf(1,1:100)
12
     DO j=1,10000
13
       buf_1dim(h)=...
14
     END DO
15
     buf(1,1:100) = tmp(1:100)
16
     CALL MPI_Wait(req,...)
17
18
19
20
21
22
23
24
25
26
27
28
29
     Example 18.8 Another optimization is based on the usage of a separate memory storage
30
     area, e.g., in a GPU.
31
32
     REAL :: buf(100,100), local_buf(100,100)
33
     CALL MPI_Irecv(buf(1,1:100),...req,...)
34
     local_buf = buf
35
     DO j=1,100
36
       DO i=2,100
37
          local_buf(i,j)=....
38
       END DO
39
     END DO
40
     buf = local_buf ! may overwrite asynchronously received
41
                        ! data in buf(1,1:100)
42
     CALL MPI_Wait(req,...)
43
44
45
46
47
48
```

**Example 18.9** Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

```
10
USE mpi_f08
                                                                                  11
REAL :: b(0:101) ! elements 0 and 101 are halo cells
                                                                                  12
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                  13
INTEGER :: i
                                                                                  14
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
                                                                                  15
i=1 ! compute leftmost element
                                                                                  16
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                  17
i=100 ! compute rightmost element
                                                                                  18
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                  19
END
                                                                                  20
                                                                                  21
SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
                                                                                  22
USE mpi_f08
                                                                                  23
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
                                                                                  24
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                  25
TYPE(MPI_Request) :: req(4)
                                                                                  26
INTEGER :: left, right, i
                                                                                  27
CALL MPI_Cart_shift(...,left,right,...)
                                                                                  28
CALL MPI_Irecv(b_lefthalo ( 0), ..., left, ..., req(1), ...)
                                                                                  29
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
                                                                                  30
! b_lefthalo and b_righthalo is written asynchronously.
                                                                                  31
! There is no other concurrent access to b_lefthalo and b_righthalo.
                                                                                  32
CALL MPI_Isend(b_inner( 1), ..., left, ..., req(3), ...)
                                                                                  33
CALL MPI_Isend(b_inner(100), ..., right, ..., req(4), ...)
                                                                                  34
                                                                                  35
DO i=2,99 ! compute only elements for which halo data is not needed
                                                                                  36
  bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
                                                                                  37
  ! b_inner is read and sent at the same time.
                                                                                  38
  ! This is allowed based on the rules for ASYNCHRONOUS.
                                                                                  39
END DO
                                                                                  40
CALL MPI_Waitall(4,req,...)
                                                                                  41
END SUBROUTINE
                                                                                  42
                                                                                  43
```

1 2

4 5 6

8

9

```
1
\mathbf{2}
3
4
5
6
7
8
9
10
11
12
13
14
     Example 18.10 Protecting GPU optimizations with the ASYNCHRONOUS attribute.
15
     USE mpi_f08
16
     REAL :: buf(100,100)
17
     CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
18
     END
19
20
     SUBROUTINE separated_sections(buf_halo, buf_inner)
21
     REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
22
     REAL :: buf_inner(2:100,1:100)
23
     REAL :: local_buf(2:100,100)
^{24}
25
     CALL MPI_Irecv(buf_halo(1,1:100),...req,...)
26
     local_buf = buf_inner
27
     DO j=1,100
28
       DO i=2,100
29
          local_buf(i,j)=....
30
       END DO
31
     END DO
32
     buf_inner = local_buf ! buf_halo is not touched!!!
33
34
     CALL MPI_Wait(req,...)
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

# 18.2 Language Interoperability

### 18.2.1 Introduction

It is not uncommon for library developers to use one language to develop an application library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication used between a parallel client and a parallel server. It should be possible to code the server in one language and the clients in another language. To do so, communications should be possible between applications written in different languages.

There are several issues that need to be addressed in order to achieve interoperability.

**Initialization** We need to specify how the MPI environment is initialized for all languages.

- Interlanguage passing of MPI opaque objects We need to specify how MPI object handles are passed between languages. We also need to specify what happens when an MPI object is accessed in one language, to retrieve information (e.g., attributes) set in another language.
- **Interlanguage communication** We need to specify how messages sent in one language can be received in another language.

It is highly desirable that the solution for interlanguage interoperability be extensible to new languages, should MPI bindings be defined for such languages.

### 18.2.2 Assumptions

We assume that conventions exist for programs written in one language to call routines written in another language. These conventions specify how to link routines in different languages into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic data types in different languages. In general, these conventions will be implementation dependent. Furthermore, not every basic datatype may have a matching type in other languages. For example, C character strings may not be compatible with Fortran CHARACTER variables. However, we assume that a Fortran INTEGER, as well as a (sequence associated) Fortran array of INTEGERs, can be passed to a C program. We also assume that Fortran and C have addresssized integers. This does not mean that the default-size integers are the same size as default-sized pointers, but only that there is some way to hold (and pass) a C address in a Fortran integer. It is also assumed that INTEGER(KIND=MPI\_OFFSET\_KIND) can be passed from Fortran to C as MPI\_Offset.

### 18.2.3 Initialization

A call to MPI\_INIT or MPI\_INIT\_THREAD, from any language, initializes MPI for execution in all languages.

Advice to users. Certain implementations use the (inout) argc, argv arguments of the C version of MPI\_INIT in order to propagate values for argc and argv to all

### Unofficial Draft for Comment Only

1 2

3 4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15 16

17

18

19

20 21

22

23

 $^{24}$ 

25 26

27 28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41 42

43 44

45

 $46 \\ 47$ 

2 result in a loss of this ability. (End of advice to users.) 3 The function MPI\_INITIALIZED returns the same answer in all languages. 4 The function MPI\_FINALIZE finalizes the MPI environments for all languages. 5The function MPI\_FINALIZED returns the same answer in all languages. 6 The function MPI\_ABORT kills processes, irrespective of the language used by the 7 caller or by the processes killed. 8 9 The MPI environment is initialized in the same manner for all languages by 10 MPI\_INIT. E.g., MPI\_COMM\_WORLD carries the same information regardless of language: 11same processes, same environmental attributes, same error handlers. Information can be added to info objects in one language and retrieved in another. 1213 Advice to users. The use of several languages in one MPI program may require the 14use of special options at compile and/or link time. (End of advice to users.) 1516Advice to implementors. Implementations may selectively link language specific MPI 17 libraries only to codes that need them, so as not to increase the size of binaries for codes 18 that use only one language. The MPI initialization code need perform initialization for 19 a language only if that language library is loaded. (End of advice to implementors.) 202118.2.4 Transfer of Handles 22 23Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran  $^{24}$ handles to C handles. There is no direct access to C handles in Fortran. 25The type definition MPI\_Fint is provided in C for an integer of the size that matches a 26Fortran INTEGER; usually, MPI\_Fint will be equivalent to int. With the Fortran mpi module 27or the mpif.h include file, a Fortran handle is a Fortran INTEGER value that can be used in 28the following conversion functions. With the Fortran mpi\_f08 module, a Fortran handle is a 29 BIND(C) derived type that contains an INTEGER component named MPI\_VAL. This INTEGER 30 value can be used in the following conversion functions.  $^{31}$ The following functions are provided in C to convert from a Fortran communicator 32 handle (which is an integer) to a C communicator handle, and vice versa. See also Sec-33 tion 2.6.4. 34MPI\_Comm MPI\_Comm\_f2c(MPI\_Fint comm) 35 If comm is a valid Fortran handle to a communicator, then MPI\_Comm\_f2c returns a 36 valid C handle to that same communicator; if  $comm = MPI_COMM_NULL$  (Fortran value), 37 then MPI\_Comm\_f2c returns a null C handle; if comm is an invalid Fortran handle, then 38 39 MPI\_Comm\_f2c returns an invalid C handle. MPI\_Fint MPI\_Comm\_c2f(MPI\_Comm comm) 4041 The function MPI\_Comm\_c2f translates a C communicator handle into a Fortran handle 42to the same communicator; it maps a null handle into a null handle and an invalid handle 43 into an invalid handle. 44 Similar functions are provided for the other types of opaque objects. 45MPI\_Datatype MPI\_Type\_f2c(MPI\_Fint datatype) 4647MPI\_Fint MPI\_Type\_c2f(MPI\_Datatype datatype)

executing processes. Use of the Fortran version of MPI\_INIT to initialize MPI may

1

MPI_Group MPI_Group_f2c(MPI_Fint group)	1
MPI_Fint MPI_Group_c2f(MPI_Group group)	2 3
MPI_Request MPI_Request_f2c(MPI_Fint request)	4
MPI_Fint MPI_Request_c2f(MPI_Request request)	5 6
MPI_File MPI_File_f2c(MPI_Fint file)	7
MPI_Fint MPI_File_c2f(MPI_File file)	8
	9 10
MPI_Win MPI_Win_f2c(MPI_Fint win)	11
MPI_Fint MPI_Win_c2f(MPI_Win win)	12 13
MPI_Op MPI_Op_f2c(MPI_Fint op)	14
MPI_Fint MPI_Op_c2f(MPI_Op op)	15
MPI_Info MPI_Info_f2c(MPI_Fint info)	16 17
MPI_Fint MPI_Info_c2f(MPI_Info info)	18
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	19
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	20 21
MPI_Message MPI_Message_f2c(MPI_Fint message)	22
MPI_Fint MPI_Message_c2f(MPI_Message message)	23 24
m 1_Fint m 1_Message_021(m 1_Message message)	25
<b>Example 18.11</b> The example below illustrates how the Fortran MPI function	26
MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function	27 28
MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and	29
arguments are passed by addresses.	30
	31 32
! FORTRAN PROCEDURE SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)	33
INTEGER :: DATATYPE, IERR	34
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)	35 36
RETURN END	37
	38
/* C wrapper */	39 40
<pre>void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, MPI_Fint *ierr)</pre>	40
{	42
MPI_Datatype datatype;	43
datatura - MDI Tura fac( *f bandla).	44 45
<pre>datatype = MPI_Type_f2c( *f_handle); *ierr = (MPI_Fint)MPI_Type_commit( &amp;datatype);</pre>	46
<pre>*f_handle = MPI_Type_c2f(datatype);</pre>	47
return;	48

1

 $\mathbf{2}$ 

3

4

The same approach can be used for all other MPI functions. The call to MPI\_XXX\_f2c (resp. MPI\_XXX\_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

Rationale. The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.

Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple (e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (End of rationale.)

18.2.5 Status

19 The following two procedures are provided in C to convert from a Fortran (with the mpi 20module or mpif.h) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which 22 is hidden. That is, no status information is lost in the conversion. 23

 $^{24}$ 25

26

27

28

29

30

 $^{31}$ 

32

33

42

43

44

45

int MPI\_Status\_f2c(const MPI\_Fint \*f\_status, MPI\_Status \*c\_status)

If f\_status is a valid Fortran status, but not the Fortran value of MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE, then MPI\_Status\_f2c returns in c\_status a valid C status with the same content. If f\_status is the Fortran value of MPI\_STATUS\_IGNORE or

MPI\_STATUSES\_IGNORE, or if f\_status is not a valid Fortran status, then the call is erroneous. The C status has the same source, tag and error code values as the Fortran status, and returns the same answers when queried for count, elements, and cancellation. The conversion function may be called with a Fortran status argument that has an undefined error field, in which case the value of the error field in the C status argument is undefined.

Two global variables of type MPI\_Fint\*, MPI\_F\_STATUS\_IGNORE and 34MPI\_F\_STATUSES\_IGNORE are declared in mpi.h. They can be used to test, in C, whether 35 f\_status is the Fortran value of MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE defined in 36 the mpi module or mpif.h. These are global variables, not C constant expressions and 37 cannot be used in places where C requires constant expressions. Their value is defined only 38 between the calls to MPI\_INIT and MPI\_FINALIZE and should not be changed by user code. 39 To do the conversion in the other direction, we have the following: 40

int MPI\_Status\_c2f(const MPI\_Status \*c\_status, MPI\_Fint \*f\_status) 41

This call converts a C status into a Fortran status, and has a behavior similar to MPI\_Status\_f2c. That is, the value of c\_status must not be either MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE.

46 Advice to users. There exists no separate conversion function for arrays of statuses, 47 since one can simply loop through the array, converting each status with the routines 48 in Figure 18.1. (End of advice to users.)

# Unofficial Draft for Comment Only

}

56 7

8

9

10

11

12

13

14

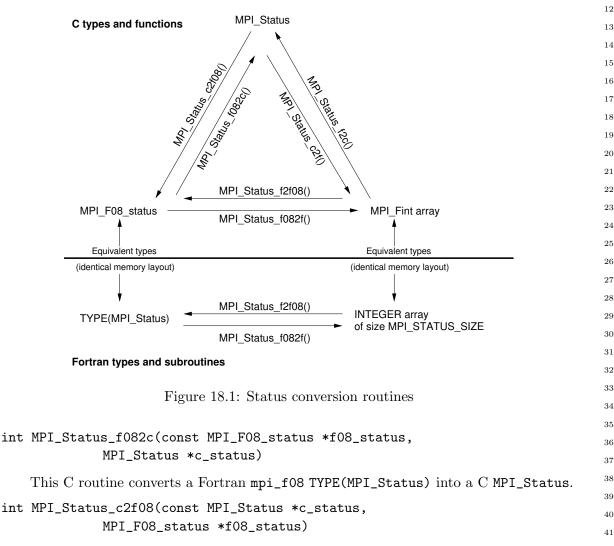
15

1617

18

Rationale. The handling of MPI\_STATUS\_IGNORE is required in order to layer libraries with only a C wrapper: if the Fortran call has passed MPI\_STATUS\_IGNORE, then the C wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and C. If MPI\_Status\_f2c were to handle MPI\_STATUS\_IGNORE, then the type of its result would have to be MPI\_Status\*\*, which was considered an inferior solution. (End of rationale.)

Using the mpi\_f08 Fortran module, a status is declared as TYPE(MPI\_Status). The C type MPI\_F08\_status can be used to pass a Fortran TYPE(MPI\_Status) argument into a C routine. Figure 18.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.



This C routine converts a C MPI\_Status into a Fortran mpi\_f08 TYPE(MPI\_Status). <sup>42</sup> Two global variables of type MPI\_F08\_status\*, MPI\_F08\_STATUS\_IGNORE and <sup>43</sup> MPI\_F08\_STATUSES\_IGNORE are declared in mpi.h. They can be used to test, in C, whether <sup>44</sup> f\_status is the Fortran value of MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE defined in <sup>45</sup> the mpi\_f08 module. These are global variables, not C constant expressions and cannot be <sup>46</sup> used in places where C requires constant expressions. Their value is defined only between <sup>47</sup> the calls to MPI\_INIT and MPI\_FINALIZE and should not be changed by user code. <sup>48</sup>

### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

```
1
          Conversion between the two Fortran versions of a status can be done with:
\mathbf{2}
3
     MPI_STATUS_F2F08(f_status, f08_status)
4
5
       IN
                 f_status
                                            status object declared as array
6
       OUT
                 f08_status
                                            status object declared as named type
7
8
     int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)
9
10
     MPI_Status_f2f08(f_status, f08_status, ierror)
11
          INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
12
          TYPE(MPI_Status), INTENT(OUT) :: f08_status
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
15
          INTEGER :: F_STATUS(MPI_STATUS_SIZE)
16
          TYPE(MPI_Status) :: F08_STATUS
17
          INTEGER IERROR
18
19
         This routine converts a Fortran INTEGER, DIMENSION (MPI_STATUS_SIZE) status array
20
     into a Fortran mpi_f08 TYPE(MPI_Status).
21
22
     MPI_STATUS_F082F(f08_status, f_status)
23
24
       IN
                 f08_status
                                            status object declared as named type
25
       OUT
                 f_status
                                            status object declared as array
26
27
     int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)
28
29
     MPI_Status_f082f(f08_status, f_status, ierror)
30
          TYPE(MPI_Status), INTENT(IN) :: f08_status
31
          INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
          TYPE(MPI_Status) :: F08_STATUS
35
          INTEGER :: F_STATUS(MPI_STATUS_SIZE)
36
          INTEGER IERROR
37
38
          This routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER,
39
     DIMENSION(MPI_STATUS_SIZE) status array.
40
41
            MPI Opaque Objects
     18.2.6
42
43
     Unless said otherwise, opaque objects are "the same" in all languages: they carry the same
     information, and have the same meaning in both languages. The mechanism described
44
45
     in the previous section can be used to pass references to MPI objects from language to
46
     language. An object created in one language can be accessed, modified or freed in another
47
     language.
```

<sup>48</sup> We examine below in more detail issues that arise for each type of MPI object.

### Unofficial Draft for Comment Only

### Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like MPI\_TYPE\_GET\_EXTENT will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function MPI\_GET\_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI\_BOTTOM have the same value in all languages (see Section 18.2.9).

### Example 18.12

```
! FORTRAN CODE
REAL :: R(5)
INTEGER :: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
! create an absolute datatype for array R
                                                                                   20
AOBLEN(1) = 5
                                                                                   21
CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
                                                                                   22
AOTYPE(1) = MPI_REAL
                                                                                   23
                                                                                   24
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
                                                                                   25
CALL C_ROUTINE(TYPE)
/* C code */
                                                                                   27
                                                                                   28
void C_ROUTINE(MPI_Fint *ftype)
                                                                                   29
{
                                                                                   30
   int count = 5;
   int lens[2] = \{1, 1\};
   MPI_Aint displs[2];
   MPI_Datatype types[2], newtype;
                                                                                   34
                                                                                   35
   /* create an absolute datatype for buffer that consists
                                                                */
                                                                                   36
   /* of count, followed by R(5)
                                                                */
                                                                                   37
                                                                                   38
   MPI_Get_address(&count, &displs[0]);
                                                                                   39
   displs[1] = 0;
   types[0] = MPI_INT;
   types[1] = MPI_Type_f2c(*ftype);
                                                                                   42
   MPI_Type_create_struct(2, lens, displs, types, &newtype);
                                                                                   43
   MPI_Type_commit(&newtype);
                                                                                   44
   MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   /* the message sent contains an int count of 5, followed
                                                                */
   /* by the 5 REAL entries of the Fortran array R.
                                                                */
```

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

1213

14

1516

17

18

19

26

31

32

33

40

41

45

46

47

Advice to implementors. The following implementation can be used: MPI addresses, as returned by MPI\_GET\_ADDRESS, will have the same value in all languages. One obvious choice is that MPI addresses be identical to regular addresses. The address is stored in the datatype, when datatypes with absolute addresses are constructed. When a send or receive operation is performed, then addresses stored in a datatype are interpreted as displacements that are all augmented by a base address. This base address is (the address of) buf, or zero, if buf = MPI\_BOTTOM. Thus, if MPI\_BOTTOM is zero then a send or receive call with buf = MPI\_BOTTOM is implemented exactly as a call with a regular buffer argument: in both cases the base address is buf. On the other hand, if MPI\_BOTTOM is not zero, then the implementation has to be slightly different. A test is performed to check whether buf = MPI\_BOTTOM. If true, then the base address is zero, otherwise it is buf. In particular, if MPI\_BOTTOM does not have the same value in Fortran and C, then an additional test for buf = MPI\_BOTTOM is needed in at least one of the languages.

It may be desirable to use a value other than zero for MPI\_BOTTOM even in C, so as to distinguish it from a NULL pointer. If MPI\_BOTTOM = c then one can still avoid the test buf = MPI\_BOTTOM, by using the displacement from MPI\_BOTTOM, i.e., the regular address - c, as the MPI address returned by MPI\_GET\_ADDRESS and stored in absolute datatypes. (*End of advice to implementors.*)

23 Callback Functions

<sup>24</sup> MPI calls may associate callback functions with MPI objects: error handlers are associ-<sup>25</sup> ated with communicators and files, attribute copy and delete functions are associated with <sup>27</sup> attribute keys, reduce operations are associated with operation objects, etc. In a multilan-<sup>28</sup> guage environment, a function passed in an MPI call in one language may be invoked by an <sup>29</sup> MPI call in another language. MPI implementations must make sure that such invocation <sup>20</sup> will use the calling convention of the language the function is bound to.

Advice to implementors. Callback functions need to have a language tag. This tag is set when the callback function is passed in by the library function (which is presumably different for each language and language support method), and is used to generate the right calling sequence when the callback function is invoked. (End of advice to implementors.)

Advice to users. If a subroutine written in one language or Fortran support method wants to pass a callback routine including the predefined Fortran functions (e.g., MPI\_COMM\_NULL\_COPY\_FN) to another application routine written in another language or Fortran support method, then it must be guaranteed that both routines use the callback interface definition that is defined for the argument when passing the callback to an MPI routine (e.g., MPI\_COMM\_CREATE\_KEYVAL); see also the advice to users on page 272. (End of advice to users.)

# Error Handlers

Advice to implementors. Error handlers, have, in C, a variable length argument list.
 It might be useful to provide to the handler information on the language environment
 where the error occurred. (End of advice to implementors.)

# Unofficial Draft for Comment Only

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

36

37

38

39

40

41

42

43 44

#### **Reduce Operations**

All predefined named and unnamed datatypes as listed in Section 5.9.2 can be used in the listed predefined operations independent of the programming language from which the MPI routine is called.

Advice to users. Reduce operations receive as one of their arguments the datatype of the operands. Thus, one can define "polymorphic" reduce operations that work for C and Fortran datatypes. (*End of advice to users.*)

### 18.2.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI\_TAG\_UB, MPI\_WTIME\_IS\_GLOBAL, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI\_{TYPE,COMM,WIN}\_CREATE\_KEYVAL call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

Advice to implementors. This requires that attributes be tagged either as "C" or "Fortran" and that the language tag be checked in order to use the right calling convention for the callback function. (*End of advice to implementors.*)

The attribute manipulation functions described in Section 6.7 defines attributes arguments to be of type void\* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C callee, or vice-versa.

MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs are smaller, then the (deprecated) Fortran function MPI\_ATTR\_GET will return the least significant part of the attribute word; the (deprecated) Fortran function MPI\_ATTR\_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized attributes, and have the same functionality as the old functions in C. These functions are described in Section 6.7. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integer-valued attributes. C attribute functions put and get address-valued attributes. Fortran attribute functions put and get integer-valued attributes. When an integer-valued attribute is accessed from C, then MPI\_XXX\_get\_attr will return the address of (a pointer to) the integer-valued attribute, which is a pointer to MPI\_Aint if the attribute was stored with Fortran MPI\_XXX\_SET\_ATTR, and a pointer to int if it was stored with the deprecated Fortran MPI\_ATTR\_PUT. When an address-valued attribute is accessed from Fortran, then MPI\_XXX\_GET\_ATTR will convert the address into an integer and return the result of this conversion. This conversion is lossless if new style attribute functions are used, and an integer of kind MPI\_ADDRESS\_KIND is returned. The conversion may cause truncation if 

#### **Unofficial Draft for Comment Only**

 $\overline{7}$ 

1 deprecated attribute functions are used. In C, the deprecated routines MPI\_Attr\_put and  $\mathbf{2}$ MPI\_Attr\_get behave identical to MPI\_Comm\_set\_attr and MPI\_Comm\_get\_attr. 3 Example 18.13 4 A. Setting an attribute value in C 56 int set\_val = 3; 7 struct foo set\_struct; 8 9 /\* Set a value that is a pointer to an int \*/ 10 11 MPI\_Comm\_set\_attr(MPI\_COMM\_WORLD, keyval1, &set\_val); 12/\* Set a value that is a pointer to a struct \*/ 13 MPI\_Comm\_set\_attr(MPI\_COMM\_WORLD, keyval2, &set\_struct); 14 /\* Set an integer value \*/ 15 MPI\_Comm\_set\_attr(MPI\_COMM\_WORLD, keyval3, (void \*) 17); 16 17B. Reading the attribute value in C 18 19 int flag, \*get\_val; 20struct foo \*get\_struct; 2122/\* Upon successful return, get\_val == &set\_val 23(and therefore \*get\_val == 3) \*/  $^{24}$ MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval1, &get\_val, &flag); 25/\* Upon successful return, get\_struct == &set\_struct \*/ 26MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval2, &get\_struct, &flag); 27/\* Upon successful return, get\_val == (void\*) 17 \*/ 28i.e., (MPI\_Aint) get\_val == 17 \*/ /\* 29MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval3, &get\_val, &flag); 30 C. Reading the attribute value with (deprecated) Fortran MPI-1 calls 3132 LOGICAL FLAG 33 INTEGER IERR, GET\_VAL, GET\_STRUCT 34 35 ! Upon successful return, GET\_VAL == &set\_val, possibly truncated 36 CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL1, GET\_VAL, FLAG, IERR) 37 ! Upon successful return, GET\_STRUCT == &set\_struct, possibly truncated 38 CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL2, GET\_STRUCT, FLAG, IERR) 39 ! Upon successful return, GET\_VAL == 17 40CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL3, GET\_VAL, FLAG, IERR) 41 42D. Reading the attribute value with Fortran MPI-2 calls 43 44 4546 47 48

```
1
LOGICAL FLAG
                                                                                       \mathbf{2}
INTEGER IERR
                                                                                       3
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
                                                                                       4
! Upon successful return, GET_VAL == &set_val
                                                                                       5
                                                                                       6
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
                                                                                       7
! Upon successful return, GET_STRUCT == &set_struct
                                                                                       8
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
! Upon successful return, GET_VAL == 17
                                                                                       9
                                                                                       10
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
                                                                                       11
                                                                                       12
Example 18.14 A. Setting an attribute value with the (deprecated) Fortran MPI-1 call
                                                                                      13
                                                                                      14
INTEGER IERR, VAL
                                                                                       15
VAL = 7
                                                                                       16
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
                                                                                       17
                                                                                       18
    B. Reading the attribute value in C
                                                                                       19
                                                                                      20
int flag;
                                                                                      21
int *value;
                                                                                      22
                                                                                      23
/* Upon successful return, value points to internal MPI storage and
                                                                                       ^{24}
   *value == (int) 7 */
                                                                                      25
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
                                                                                       26
                                                                                      27
    C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
                                                                                      28
                                                                                      29
LOGICAL FLAG
                                                                                       30
INTEGER IERR, VALUE
                                                                                       31
                                                                                       32
! Upon successful return, VALUE == 7
                                                                                      33
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                      34
    D. Reading the attribute value with Fortran MPI-2 calls
                                                                                      35
                                                                                      36
LOGICAL FLAG
                                                                                      37
INTEGER IERR
                                                                                       38
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE
                                                                                       39
                                                                                       40
! Upon successful return, VALUE == 7 (sign extended)
                                                                                      41
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                      42
                                                                                      43
                                                                                      44
Example 18.15 A. Setting an attribute value via a Fortran MPI-2 call
                                                                                       45
                                                                                       46
                                                                                       47
```

```
1
     INTEGER IERR
\mathbf{2}
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
3
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
4
     VALUE1 = 42
\mathbf{5}
     VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40
6
7
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
8
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
9
         B. Reading the attribute value in C
10
11
     int flag;
12
     MPI_Aint *value1, *value2;
13
14
     /* Upon successful return, value1 points to internal MPI storage and
15
        *value1 == 42 */
16
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
17
     /* Upon successful return, value2 points to internal MPI storage and
18
        *value2 == 2^40 */
19
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
20
21
         C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
22
23
     LOGICAL FLAG
24
     INTEGER IERR, VALUE1, VALUE2
25
26
     ! Upon successful return, VALUE1 == 42
27
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
28
     ! Upon successful return, VALUE2 == 2<sup>40</sup>, or 0 if truncation
29
     ! needed (i.e., the least significant part of the attribute word)
30
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
^{31}
32
         D. Reading the attribute value with Fortran MPI-2 calls
33
34
     LOGICAL FLAG
35
     INTEGER IERR
36
     INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2
37
38
     ! Upon successful return, VALUE1 == 42
39
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
40
     ! Upon successful return, VALUE2 == 2^40
41
     CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
42
43
         The predefined MPI attributes can be integer valued or address-valued. Predefined
^{44}
     integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a call to
45
     the deprecated Fortran routine MPI_ATTR_PUT, i.e., in Fortran,
46
     MPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_TAG_UB, val, flag, ierr) will return
47
     in val the upper bound for tag value; in C, MPI_Comm_get_attr(MPI_COMM_WORLD,
48
```

MPI\_TAG\_UB, &p, &flag) will return in p a pointer to an int containing the upper bound for tag value.

Address-valued predefined attributes, such as MPI\_WIN\_BASE behave as if they were put by a C call, i.e., in Fortran, MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_BASE, val, flag, ierror) will return in val the base address of the window, converted to an integer. In C, MPI\_Win\_get\_attr(win, MPI\_WIN\_BASE, &p, &flag) will return in p a pointer to the window base, cast to (void \*).

*Rationale.* The design is consistent with the behavior specified for predefined attributes, and ensures that no information is lost when attributes are passed from language to language. Because the language interoperability for predefined attributes was defined based on MPI\_ATTR\_PUT, this definition is kept for compatibility reasons although the routine itself is now deprecated. (*End of rationale.*)

Advice to implementors. Implementations should tag attributes either as (1) address attributes, (2) as INTEGER(KIND=MPI\_ADDRESS\_KIND) attributes or (3) as INTEGER attributes, according to whether they were set in (1) C (with MPI\_Attr\_put or MPI\_XXX\_set\_attr), (2) in Fortran with MPI\_XXX\_SET\_ATTR or (3) with the deprecated Fortran routine MPI\_ATTR\_PUT. Thus, the right choice can be made when the attribute is retrieved. (End of advice to implementors.)

# 18.2.8 Extra-State

Extra-state should not be modified by the copy or delete callback functions. (This is obvious from the C binding, but not obvious from the Fortran binding). However, these functions may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be a pointer to a data structure that is modified by the copy or callback functions; in Fortran, extra-state can be an index into an entry in a COMMON array that is modified by the copy or callback functions. In a multithreaded environment, users should be aware that distinct threads may invoke the same callback function concurrently: if this function modifies state associated with extra-state, then mutual exclusion code must be used to protect updates and accesses to the shared state.

# 18.2.9 Constants

MPI constants have the same value in all languages, unless specified otherwise. This does not apply to constant handles (MPI\_INT, MPI\_COMM\_WORLD, MPI\_ERRORS\_RETURN, MPI\_SUM, etc.) These handles need to be converted, as explained in Section 18.2.4. Constants that specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in Fortran than C since in C the length includes the null terminating character. Thus, these constants represent the amount of space which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

Advice to users. This definition means that it is safe in C to allocate a buffer to receive a string using a declaration like

char name [MPI\_MAX\_OBJECT\_NAME];

### Unofficial Draft for Comment Only

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $^{31}$ 

1

2

3

4

5 6

7

8

9

10

11

12

13

(End of advice to users.)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI\_BOTTOM or MPI\_STATUS\_IGNORE may have different values in different languages.

*Rationale.* The current MPI standard specifies that MPI\_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI\_BOTTOM in Fortran must be the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI\_BOTTOM = 0 (Caveat: Defining MPI\_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI\_BOTTOM; it may be that MPI\_BOTTOM = 1 is better. See the advice to implementors in the *Datatypes* subsection in Section 18.2.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (*End of rationale.*)

#### 14 15 16

17

# 18.2.10 Interlanguage Communication

The type matching rules for communication in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI\_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI\_BYTE or MPI\_PACKED. Interlanguage communication is allowed if it complies with these rules.

Example 18.16 In the example below, a Fortran array is sent from Fortran and received in C.

```
27
     ! FORTRAN CODE
28
     SUBROUTINE MYEXAMPLE()
29
     USE mpi_f08
30
     REAL :: R(5)
^{31}
     INTEGER :: IERR, MYRANK, AOBLEN(1)
32
     TYPE(MPI_Datatype) :: TYPE, AOTYPE(1)
33
     INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
34
35
     ! create an absolute datatype for array R
36
     AOBLEN(1) = 5
37
     CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
38
     AOTYPE(1) = MPI_REAL
39
     CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
40
     CALL MPI_TYPE_COMMIT(TYPE, IERR)
41
42
     CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
43
     IF (MYRANK.EQ.O) THEN
44
        CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
45
     ELSE
46
        CALL C_ROUTINE(TYPE%MPI_VAL)
47
     END IF
48
     END SUBROUTINE
```

```
/* C code */
void C_ROUTINE(MPI_Fint *fhandle)
{
    MPI_Datatype type;
    MPI_Status status;
    type = MPI_Type_f2c(*fhandle);
    MPI_Recv( MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
}
```

MPI implementors may weaken these type matching rules, and allow messages to be sent with Fortran types and received with C types, and vice versa, when those types match. I.e., if the Fortran type INTEGER is identical to the C type int, then an MPI implementation may allow data to be sent with datatype MPI\_INTEGER and be received with datatype MPI\_INT. However, such code is not portable.

## Annex A

# Language Bindings Summary

In this section we summarize the specific bindings for C and Fortran. First we present the constants, type definitions, info values and keys. Then we present the routine prototypes separately for each binding. Listings are alphabetical within chapter.

## A.1 Defined Values and Handles

#### A.1.1 Defined Constants

The C and Fortran names are listed below. Constants with the type const int may also be implemented as literal integer constants substituted by the preprocessor.

Error classes	
C type: const int (or unnamed enum)	
Fortran type: INTEGER	
MPI_SUCCESS	
MPI_ERR_BUFFER	
MPI_ERR_COUNT	
MPI_ERR_TYPE	
MPI_ERR_TAG	
MPI_ERR_COMM	
MPI_ERR_RANK	
MPI_ERR_REQUEST	
MPI_ERR_ROOT	
MPI_ERR_GROUP	
MPI_ERR_OP	
MPI_ERR_TOPOLOGY	
MPI_ERR_DIMS	
MPI_ERR_ARG	
MPI_ERR_UNKNOWN	
MPI_ERR_TRUNCATE	
MPI_ERR_OTHER	
MPI_ERR_INTERN	
MPI_ERR_PENDING	
(Continued on next page)	

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_NOKE1
19	MPI_ERR_INFO
20	MPI_ERR_IO
21	
22	
23	
	MPI_ERR_NAME
24	MPI_ERR_NO_MEM
25	MPI_ERR_NOT_SAME
26	MPI_ERR_NO_SPACE
27	MPI_ERR_NO_SUCH_FILE
28	MPI_ERR_PORT
29	MPI_ERR_QUOTA
30	MPI_ERR_READ_ONLY
31	MPI_ERR_RMA_ATTACH
32	MPI_ERR_RMA_CONFLICT
33	MPI_ERR_RMA_RANGE
34	MPI_ERR_RMA_SHARED
35	MPI_ERR_RMA_SYNC
36	MPI_ERR_RMA_FLAVOR
37	MPI_ERR_SERVICE
38	MPI_ERR_SIZE
39	MPI_ERR_SPAWN
40	MPI_ERR_UNSUPPORTED_DATAREP
41	MPI_ERR_UNSUPPORTED_OPERATION
42	MPI_ERR_WIN
43	MPI_ERR_PROC_FAILED
44	MPI_ERR_REVOKED
45	MPI_ERR_PROC_FAILED_PENDING
46	(Continued on next page)
47	(
48	

Error classes (continued)	1
C type: const int (or unnamed enum)	2
Fortran type: INTEGER	3
MPI_T_ERR_CANNOT_INIT	4
MPI_T_ERR_NOT_INITIALIZED	5
MPI_T_ERR_MEMORY	6
MPI_T_ERR_INVALID	7
MPI_T_ERR_INVALID_INDEX	8
MPI_T_ERR_INVALID_ITEM	9
MPI_T_ERR_INVALID_SESSION	10
MPI_T_ERR_INVALID_HANDLE	11
MPI_T_ERR_INVALID_NAME	12
MPI_T_ERR_OUT_OF_HANDLES	13
MPI_T_ERR_OUT_OF_SESSIONS	14
MPI_T_ERR_CVAR_SET_NOT_NOW	15
MPI_T_ERR_CVAR_SET_NEVER	16
MPI_T_ERR_PVAR_NO_WRITE	17
MPI_T_ERR_PVAR_NO_STARTSTOP	18
MPI_T_ERR_PVAR_NO_ATOMIC	19
MPI_ERR_LASTCODE	20
	21
Buffer Address Constants	22
C type: void * const	23
Fortran type: (predefined memory location) <sup>1</sup>	24
MPI_BOTTOM	25
MPI_IN_PLACE	26
$^1$ Note that in Fortran these constants are not usable for initialization	27
expressions or assignment. See Section $2.5.4$ .	28
	29
Assorted Constants	30
C type: const int (or unnamed enum)	31
Fortran type: INTEGER	32
MPI_PROC_NULL	33
MPI_ANY_SOURCE	34
MPI_ANY_TAG	35
MPI_UNDEFINED	36
MPI_BSEND_OVERHEAD	37
MPI_KEYVAL_INVALID	38
MPI_LOCK_EXCLUSIVE	39
MPI_LOCK_SHARED	40
MPI_ROOT	41
	42
No Process Message Handle	43
C type: MPI_Message	44
Fortran type: INTEGER or TYPE(MPI_Message)	45
MPI_MESSAGE_NO_PROC	46
	47
	48

Fortran Support Method Specific Constants
Fortran type: LOGICAL
MPI_SUBARRAYS_SUPPORTED (Fortran only)
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
Status size and reserved index values (Fortran only)
Fortran type: INTEGER
MPI_STATUS_SIZE
MPI_SOURCE
MPI_TAG
MPI_ERROR
Variable Address Size (Fortran only)
Fortran type: INTEGER
MPI_ADDRESS_KIND
MPI_COUNT_KIND
MPI_INTEGER_KIND
MPI_OFFSET_KIND
Error-handling specifiers
C type: MPI_Errhandler
Fortran type: INTEGER or TYPE(MPI_Errhandler)
MPI_ERRORS_ARE_FATAL
MPI_ERRORS_RETURN
Maximum Sizes for Strings
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_MAX_DATAREP_STRING
MPI_MAX_ERROR_STRING
MPI_MAX_INFO_KEY
MPI_MAX_INFO_VAL
MPI_MAX_LIBRARY_VERSION_STRING
MPI_MAX_OBJECT_NAME
MPI_MAX_PORT_NAME
MPI_MAX_PROCESSOR_NAME

Named Predefined Datatypes C type: MPI_Datatype	C types	
Fortran type: INTEGER		
or TYPE(MPI_Datatype)		
MPI_CHAR	char	
	(treated as printable character)	
MPI_SHORT	signed short int	
MPI_INT	signed int	
MPI_LONG	signed long	
MPI_LONG_LONG_INT	signed long long	
MPI_LONG_LONG (as a synonym)	signed long long	
MPI_SIGNED_CHAR	signed char	
	(treated as integral value)	
MPI_UNSIGNED_CHAR	unsigned char	
	(treated as integral value)	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long	
MPI_UNSIGNED_LONG_LONG	unsigned long long	
MPI_FLOAT	float	
MPI_DOUBLE	double	
MPI_LONG_DOUBLE	long double	
MPI_WCHAR	wchar_t	
	(defined in <stddef.h>)</stddef.h>	
	(treated as printable character)	
MPI_C_BOOL	_Bool	
MPI_INT8_T	int8_t	
MPI_INT16_T	int16_t	
MPI_INT32_T	int32_t	
MPI_INT64_T	int64_t	
MPI_UINT8_T	uint8_t	
MPI_UINT16_T	uint16_t	
MPI_UINT32_T	uint32_t	
MPI_UINT64_T	uint64_t	
MPI_AINT	MPI_Aint	
MPI_COUNT	MPI_Count	
MPI_OFFSET	MPI_Offset	
MPI_C_COMPLEX	float _Complex	
MPI_C_FLOAT_COMPLEX	float _Complex	
MPI_C_DOUBLE_COMPLEX	double _Complex	
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex	
MPI_BYTE	(any C type)	
MPI_PACKED	(any C type)	

 $45 \\ 46$ 

Image: Section of the sectin of the section of the section of the section of the section of th	1	Named Predefined Datatypes	For	tran	types
arr TYPE(MPI_Datatype)     INTEGER       6     MPI_INTEGER     INTEGER       6     MPI_REAL     REAL       7     MPI_OUBLE_PRECISION     DOUBLE PRECISION       8     MPI_COMPLEX     COMPLEX       9     MPI_LOGICAL     LOGICAL       10     MPI_CHARACTER     CHARACTER(1)       11     MPI_ARACTER     CHARACTER(1)       12     MPI_CONT     INTEGER (KIND=MPI_ADDRESS_KIND)       13     MPI_CONT     INTEGER (KIND=MPI_CONT_KIND)       14     MPI_CONT     INTEGER (KIND=MPI_CONT_KIND)       15     MPI_CONT     INTEGER (KIND=MPI_CONT_KIND)       16     MPI_CXXEND     (any Fortran type)       17     Named Predefined Datatypes1     C++ types       18     C type: NPI_Datatype     (any Fortran type)       19     Fortran type: INTEGER     std::complex <float>       21     MPI_CXX_DOUBLE_COMPLEX     std::complex<float>       22     MPI_CXX_DOUBLE_COMPLEX     std::complex<load< td="">       23     MPI_CXX_DOUBLE_COMPLEX     std::complex<load< td="">       24     MPI_CXX_DOUBLE_COMPLEX     std::complex<load< td="">       25     T ff an accompanying C++ compiler is missing, then the     MPI_ATTREGER1       26     MPI_CXX_DUBLE_COMPLEX     DOUBLE COMPLEX       27     Optional</load<></load<></load<></float></float>	2	C type: MPI_Datatype			
MPI_INTEGER     INTEGER       MPI_DOUBLE_PRECISION     DOUBLE PRECISION       MPI_COMPLEX     COMPLEX       MPI_COMPLEX     COMPLEX       MPI_COMPLEX     COMPLEX       MPI_COLOGICAL     LOGICAL       MPI_CONT     INTEGER (KIND=MPI_ADDRESS_KIND)       MPI_COUNT     INTEGER (KIND=MPI_ODTSET_KIND)       MPI_POFFSET     INTEGER (KIND=MPI_OFFSET_KIND)       MPI_BYTE     (any Fortran type)       MPI_CXX_BOOL     bool       MPI_CXX_BOOL     bool       MPI_CXX_BOOL     std::complex <float>       MPI_CXX_BOOL     std::complex<float>       MPI_CXX_BOOL     std::complex<float>       MPI_CXX_DOUBLE_COMPLEX     std::complex<float>       MPI_CXX_DOUBLE_COMPLEX     std::complex<long double="">       *     MPI_CXX_LONC_DOUBLE_COMPLEX     std::complex<long double="">       *     MPI_CXX_DOUBLE_COMPLEX     std::complex<long double="">       *     MPI_DUBLE_COMPLEX     DUBLE COMPLEX       MPI_DOUBLE_COMPLEX     MPI_NTEGER4     INTEGER*1       MPI_NTEGER1     INTEGER*8     INTEGER*8       MPI_INTEGER8     INTEGER*8     MPI_NTEGER*8       MPI_NTEGER16     INTEGER*8     MPI_REAL4       MPI_REAL4     REAL*4     REAL*8       MPI_COMPLEX4     COMPLEX*8       MPI_COMPL</long></long></long></float></float></float></float>	3	Fortran type: INTEGER			
MPI_REAL     REAL       MPI_DOUBLE_PRECISION     DUBLE PRECISION       MPI_LOGICAL     COMPLEX       MPI_LOGICAL     LOGICAL       MPI_CHARACTER     CHARACTER(1)       MPI_COUNT     INTEGER (KIND=MPI_ADDRESS_KIND)       MPI_COUNT     INTEGER (KIND=MPI_COUNT_KIND)       MPI_OFFSET     INTEGER (KIND=MPI_COUNT_KIND)       MPI_OFFSET     INTEGER (KIND=MPI_COUNT_KIND)       MPI_OFFSET     INTEGER (KIND=MPI_COUNT_KIND)       MPI_CXCKED     (any Fortran type)       MPI_CXX_BOOL     bool       MPI_CXX_BOOL     bool       MPI_CXX_DOUBLE_COMPLEX     std::complex <float>       MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       *     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       *     If an accompanying C++ compiler is missing, then the       MPI_CXX_LONG_DOUBLE_COMPLEX     potional datatypes       *     Optional datatypes (Fortran)     Fortran types       *     Or TYPE(MPI_Datatype)     MPI_DOUBLE_COMPLEX       *     MPI_DOUBLE_COMPLEX     DOUBLE COMPLEX       *     MPI_INTEGER     INTEGER*1       *     MPI_DOUBLE_COMPLEX     DOUBLE COMPLEX       *     MPI_INTEGER     INTEGER*1       *     MPI_INTEGER     INTEGER*1       *     MPI_INTEG</long></long></float>	4	or TYPE(MPI_Datatype)			
7     MPI_DOUBLE_PRECISION     DOUBLE PRECISION       8     MPI_COMPLEX     COMPLEX       9     MPI_LOGICAL     LOGICAL       10     MPI_CHARACTER     CHARACTER(1)       11     MPI_COUNT     INTEGER (KIND=MPI_ADDRESS_KIND)       12     MPI_COUNT     INTEGER (KIND=MPI_OUNT_KIND)       13     MPL_OFFSET     INTEGER (KIND=MPI_OUNT_KIND)       14     MPI_BYTE     (any Fortran type)       15     MPI_PACKED     (any Fortran type)       16     Type(MPI_Datatype     C++ types       17     MPI_CXX_FLOAT_COMPLEX     std::complex <float>       18     C type: MPI_Datatype)     bool     std::complex<float>       19     Fortran type: INTEGER     std::complex<float>       20     or TYPE(MPI_Datatype)     std::complex<float>       21     MPI_CXX_FLOAT_COMPLEX     std::complex<float>       22     MPI_CXX_DOUG_DOUBLE_COMPLEX     std::complex<long double="">       23     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       24     MPI_CXX_LONG_DOUBLE_COMPLEX     boulde       25     T ff an accompanying C++ compiler is missing, then the       26     C type: MPI_Datatype     fortran type:       27     Optional datatypes (Fortran)     Fortran types       28     C type: MPI_Data</long></long></float></float></float></float></float>	5	MPI_INTEGER	INT	EGER	
*     MPI_COMPLEX     COMPLEX       9     MPI_LOGICAL     LOGICAL       10     MPI_CHARACTER     CHARACTER(1)       11     MPI_COUNT     INTEGER (KIND=MPI_ADDRESS_KIND)       12     MPI_COUNT     INTEGER (KIND=MPI_OUNT_KIND)       13     MPI_OFFSET     INTEGER (KIND=MPI_OUNT_KIND)       14     MPI_BYTE     (any Fortran type)       15     MPI_PACKED     (any Fortran type)       16     MPI_CXX_BOOL     bool       17     Named Predefined Datatypes     C++ types       18     C type: MPI_Datatype     bool       19     Fortran type: INTEGER     std::complex <float>       20     or TYPE(MPI_Datatype)     bool     std::complex<float>       21     MPI_CXX_LOOG_DOUBLE_COMPLEX     std::complex<float>       22     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<loaded complex<="" td="">       23     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<loaded complex<="" td="">       24     MPI_CXX_DOUBLE_COMPLEX     std::complex<loaded complex<="" td="">       25     T f an accompanying C++ compiler is missing, then the     MPI_NPI_DATATYPE       26     C type: MPI_Datatype     Fortran type:     TTTEGER*1       27     MPI_DOUBLE_COMPLEX     DUUBLE COMPLEX       28     Optional datatypes (Fortran)     Fortran type:    &lt;</loaded></loaded></loaded></float></float></float>	6	MPI_REAL	REA	L	
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_CAKED     (any Fortran type)       16     MPI_CXX_FOOD     (any Fortran type)       17     Named Predefined Datatypes <sup>1</sup> C++ types       18     C type: MPI_Datatype     (any Fortran type)       19     Fortran type: INTEGER     or TYPE(MPI_Datatype)       20     MPI_CXX_FOONLEC     std::complex <float>       21     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<float>       23     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       24     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       25     <sup>1</sup> If an accompanying C++ compiler is missing, then the     MPI datatypes in this table are not defined.       26     Optional datatypes (Fortran)     Fortran types       27     Optional datatypes     INTEGER*4       38     MPI_INTEGER1     INTEGER*2       39     MPI_INTEGER1     INTEGER*4       30     MPI_INTEGER8     INTEGER*4       31     <td< td=""><td>7</td><td>MPI_DOUBLE_PRECISION</td><td>DOU</td><td>BLE F</td><td>PRECISION</td></td<></long></long></float></float>	7	MPI_DOUBLE_PRECISION	DOU	BLE F	PRECISION
Image: MPI_CHARACTER     CHARACTER(1)       IMPI_AINT     INTEGER (KIND=MPI_ADDRESS_KIND)       IMPI_OUNT     INTEGER (KIND=MPI_COUNT_KIND)       IMPI_OFFSET     INTEGER (KIND=MPI_OFFSET_KIND)       IMPI_SYTE     (any Fortran type)       IMPI_CXX_EDD     (any Fortran type)       IMPI_CXX_EDD     bool       IMPI_CXX_FLOAT_COMPLEX     std::complex <float>       MPI_CXX_FLOAT_COMPLEX     std::complex<float>       MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       ImPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       ImPI_datatypes in this table are not defined.     MPI datatypes       ImPI_CXX_IPE     DUUBLE COMPLEX       MPI datatypes in this table are not defined.     MPI datatypes       ImPI_CXX_IPE     INTEGER       ImPI_DOUBLE_COMPLEX     Std::complex<long double="">       ImPI_CXX_IPE     Fortran type:       ImPI_CXX_IPE     Fortran type:</long></long></long></float></float>	8	MPI_COMPLEX	COM	PLEX	
Image: Second	9	MPI_LOGICAL	LOG	ICAL	
12     MPI_COUNT     INTEGER (KIND=MPI_COUNT_KIND)       13     MPI_OFFSET     INTEGER (KIND=MPI_OFFSET_KIND)       14     MPI_BYTE     (any Fortran type)       15     MPI_CACKED     (any Fortran type)       16     Named Predefined Datatypes <sup>1</sup> C++ types       17     Named Predefined Datatypes <sup>1</sup> C++ types       18     C type: MPI_Datatype     (any Fortran type)       19     Fortran type: INTEGER     or TYPE(MPI_Datatype)       20     or TYPE(MPI_Datatype)     bool       21     MPI_CXX_FLOAT_COMPLEX     std::complex <float>       23     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<loaded ouble="">       24     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<loaded ouble="">       25     If an accompanying C++ compiler is missing, then the       26     MPI datatypes in this table are not defined.       27     Optional datatypes (Fortran)     Fortran types       28     C type: MPI_Datatype     DOUBLE COMPLEX       29     C type: MPI_Datatype     MPI_INTEGER       31     or TYPE(MPI_Datatype)     MPI_INTEGER       32     MPI_INTEGER1     INTEGER*16       33     MPI_INTEGER1     INTEGER*16       34     MPI_INTEGER8     INTEGER*8       35     MPI_INTEGER16     INTEGER*16</loaded></loaded></float>	10	MPI_CHARACTER	CHA	RACTE	ER(1)
MPI_OFFSET     INTEGER     (KIND=MPI_OFFSET_KIND)       MPI_BYTE     (any Fortran type)       MPI_PACKED     (any Fortran type)       MPI_PACKED     (any Fortran type)       MPI_CXX_ED     C++ types       C type: MPI_Datatype     C++ types       Fortran type: INTEGER     bool       MPI_CXX_FLOAT_COMPLEX     std::complex <float>       MPI_CXX_DOUBLE_COMPLEX     std::complex<long double="">       MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       MPI datatypes in this table are not defined.     MPI datatypes       C type: MPI_Datatype     fortran types       C type: MPI_Datatype     std::complex<long double="">       I ff an accompanying C++ compiler is missing, then the     MPI datatypes in this table are not defined.       C type: MPI_Datatype     or TYPE(MPI_Datatype)     DOUBLE_COMPLEX       MPI_ODUBLE_COMPLEX     DOUBLE_COMPLEX     MPI_NTEGER1       MPI_INTEGER1     INTEGER*1     INTEGER*2       MPI_INTEGER8     INTEGER*4     MPI_REAL2       MPI_REAL2     REAL*2     MPI_REAL4       MPI_COMPLEX4     COMPLEX*4     MPI_REAL4       MPI_COMPLEX4     COMPLEX*4     MPI_COMPLEX4       MPI_COMPLEX4     COMPLEX*4     MPI_COMPLEX4</long></long></long></float>	11	MPI_AINT	INT	EGER	(KIND=MPI_ADDRESS_KIND)
MPI_OFFSET     INTEGER     (KIND=MPI_OFFSET_KIND)       MPI_BYTE     (any Fortran type)       MPI_PACKED     (any Fortran type)       MPI_PACKED     (any Fortran type)       MPI_CXX_ED     C++ types       C type: MPI_Datatype     C++ types       Fortran type: INTEGER     bool       MPI_CXX_FLOAT_COMPLEX     std::complex <float>       MPI_CXX_DOUBLE_COMPLEX     std::complex<long double="">       MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       MPI datatypes in this table are not defined.     MPI datatypes       C type: MPI_Datatype     fortran types       C type: MPI_Datatype     std::complex<long double="">       I ff an accompanying C++ compiler is missing, then the     MPI datatypes in this table are not defined.       C type: MPI_Datatype     or TYPE(MPI_Datatype)     DOUBLE_COMPLEX       MPI_ODUBLE_COMPLEX     DOUBLE_COMPLEX     MPI_NTEGER1       MPI_INTEGER1     INTEGER*1     INTEGER*2       MPI_INTEGER8     INTEGER*4     MPI_REAL2       MPI_REAL2     REAL*2     MPI_REAL4       MPI_COMPLEX4     COMPLEX*4     MPI_REAL4       MPI_COMPLEX4     COMPLEX*4     MPI_COMPLEX4       MPI_COMPLEX4     COMPLEX*4     MPI_COMPLEX4</long></long></long></float>	12				
MPI_BYTE     (any Fortran type)       MPI_PACKED     (any Fortran type)       MPI_PACKED     (any Fortran type)       MPI_CXCKED     C++ types       C type: MPI_Datatype     C++ types       Portran type: INTEGER     or TYPE(MPI_Datatype)       MPI_CXX_BOOL     bool       MPI_CXX_FLOAT_COMPLEX     std::complex <float>       MPI_CXX_DOUBLE_COMPLEX     std::complex<load< td="">       MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<load< td="">       MPI datatypes in this table are not defined.     MPI datatype       C type: MPI_Datatype     Fortran types       C type: MPI_Datatype     Fortran types       Poptional datatypes (Fortran)     Fortran types       C type: MPI_Datatype     mPI_CXX_POUBLE_COMPLEX       MPI dotatypes in this table are not defined.     mPI dotatype       MPI_ODUBLE_COMPLEX     DOUBLE COMPLEX       MPI_DOUBLE_COMPLEX     MPI_CMI_Datatype)       MPI_INTEGER1     INTEGER*1       MPI_INTEGER8     INTEGER*2       MPI_INTEGER16     INTEGER*16       MPI_REAL2     REAL*3       MPI_REAL3     REAL*3       MPI_REAL4     REAL*3       MPI_COMPLEX4     COMPLEX*4       MPI_COMPLEX4     COMPLEX*4       MPI_COMPLEX4     COMPLEX*32  <td>13</td><td></td><td></td><td></td><td></td></load<></load<></float>	13				
MPI_PACKED     (any Fortran type)       Image: MPI_Datatype     C type: MPI_Datatype       Image: Grown type: INTEGER     Fortran type: INTEGER       Image: Grown type: INTEGER     bool       Image: Grown type: INTEGER     bool       Image: Grown type: INTEGER     std::complex <float>       Image: Grown type: INTEGER     std::complex<float>       Image: Grown type: INTEGER     std::complex       Image: Grown type: INTEGER     fortran type:       Image: Grown type: INTEGER     or TYPE(MPI_Datatype)       Image: Grown type: INTEGER1     INTEGER*1       Image: Grown type: INTEGER1     INTEGER*2       Image: Grown type: INTEGER3     INTEGER*8       Image: Grown type: INTEGER4     INTEGER*8       Image: Grown type: INTEGER4     INTEGER*8       Image: Grown type: INTEGER3     INTEGER*8       Image: Grown type: INTEGER4     INTEGER*8       Image: Grown type: INTEGER8     I</float></float></float></float></float></float></float></float></float>	14				
Image: Named Predefined Datatypes       C++ types         Image: Named Predefined Datatype       Description         Image: Named Predefined Datatype       Std::complex <float>         Image: Named Predefined Datatype       Std::complex<load< td="">         Image: Named Predefined Datatype       Std::complex<load< td="">         Image: Named Predefined Datatype       Std::complex<load< td="">         Image: Named Predefined Datatype       Fortran types         Image: Named Predefined Datatype       Named Predefined Datatype         Image: Named Predefined Datatype       Named Predefined Datatype</load<></load<></load<></float>	15		、 ·		0 <b>1</b> )
18       C type: MPI_Datatype         19       Fortran type: INTEGER         20       or TYPE(MPI_Datatype)         21       MPI_CXX_BOOL       bool         22       MPI_CXX_FLOAT_COMPLEX       std::complex <float>         23       MPI_CXX_DOUBLE_COMPLEX       std::complex<float>         24       MPI_CXX_LONG_DOUBLE_COMPLEX       std::complex<long double="">         25       <sup>1</sup> If an accompanying C++ compiler is missing, then the         26       MPI datatypes in this table are not defined.         27       C type: MPI_Datatype         28       Optional datatypes (Fortran)         29       C type: MPI_Datatype         31       or TYPE(MPI_Datatype)         32       MPI_NTEGER1         33       MPI_INTEGER2         34       MPI_INTEGER4         35       MPI_INTEGER8         36       MPI_REAL4         37       MPI_REAL2         38       MPI_REAL4         39       MPI_REAL4         40       MPI_REAL4         41       MPI_CER44         42       MPI_COMPLEX4         44       MPI_COMPLEX4         44       MPI_COMPLEX4         44       MPI_COMPLEX4</long></float></float>	16		()	) =	(J F )
19       Fortran type: INTEGER         20       or TYPE(MPI_Datatype)         21       MPI_CXX_BOOL       bool         22       MPI_CXX_DOUBLE_COMPLEX       std::complex <float>         23       MPI_CXX_DOUBLE_COMPLEX       std::complex<float>         24       MPI_CXX_LONG_DOUBLE_COMPLEX       std::complex<long double="">         24       MPI_CXX_LONG_DOUBLE_COMPLEX       std::complex<long double="">         25       1 If an accompanying C++ compiler is missing, then the         MPI datatypes in this table are not defined.         28       Optional datatype (Fortran)       Fortran types         29       C type: MPI_Datatype         30       Fortran type: INTEGER       or TYPE(MPI_Datatype)         31       or TYPE(MPI_Datatype)       DOUBLE COMPLEX         32       MPI_DOUBLE_COMPLEX       DOUBLE COMPLEX         33       MPI_INTEGER1       INTEGER*1         34       MPI_INTEGER4       INTEGER*8         35       MPI_INTEGER8       INTEGER*8         36       MPI_REAL2       REAL*2         38       MPI_REAL4       REAL*3         39       MPI_REAL4       REAL*4         40       MPI_REAL4       REAL*4         41       MPI_COMPLEX4</long></long></float></float>	17	Named Predefined Datatype	$\mathbf{es}^1$	C+-	+ types
20       or TYPE(MPI_Datatype)         21       MPI_CXX_BOOL       bool         22       MPI_CXX_FLOAT_COMPLEX       std::complex <float>         23       MPI_CXX_DOUBLE_COMPLEX       std::complex<long double="">         24       MPI_CXX_LONG_DOUBLE_COMPLEX       std::complex<long double="">         25       <sup>1</sup> If an accompanying C++ compiler is missing, then the         26       MPI datatypes in this table are not defined.         27       C type: MPI_Datatype         28       Optional datatypes (Fortran)       Fortran types         29       C type: MPI_Datatype         30       or TYPE(MPI_Datatype)         31       or TYPE(MPI_Datatype)         32       MPI_OUBLE_COMPLEX       DOUBLE COMPLEX         33       MPI_INTEGER1       INTEGER*1         34       MPI_INTEGER8       INTEGER*2         35       MPI_INTEGER8       INTEGER*8         36       MPI_REAL2       REAL*2         38       MPI_REAL4       REAL*4         44       MPI_REAL6       REAL*4         44       MPI_COMPLEX8       COMPLEX*32         44       MPI_COMPLEX16       COMPLEX*32</long></long></float>	18	C type: MPI_Datatype			
MPI_CXX_BOOL     bool       MPI_CXX_FLOAT_COMPLEX     std::complex <float>       MPI_CXX_DOUBLE_COMPLEX     std::complex<double>       MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       MPI_datatypes in this table are not defined.     C type: MPI_Datatype       C     type: MPI_Datatype)     Fortran types       Or TYPE(MPI_Datatype)     Fortran types     DOUBLE_COMPLEX       MPI_DOUBLE_COMPLEX     DOUBLE_COMPLEX     DOUBLE COMPLEX       MPI_INTEGER1     INTEGER*1     INTEGER*1       MPI_INTEGER8     INTEGER*8     INTEGER*8       MPI_INTEGER16     INTEGER*16     INTEGER*16       MPI_REAL2     REAL*2     REAL*2       MPI_REAL4     REAL*4     MPI_REAL4       MPI_REAL8     REAL*8     MPI_COMPLEX4       MPI_COMPLEX4     COMPLEX*16     COMPLEX*16       MPI_COMPLEX32     COMPLEX*32     COMPLEX*32</long></long></long></long></long></double></float>	19	Fortran type: INTEGER			
MPI_CXX_BOOL     bool       MPI_CXX_FLOAT_COMPLEX     std::complex <float>       MPI_CXX_DOUBLE_COMPLEX     std::complex<double>       MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       MPI_datatypes in this table are not defined.     C type: MPI_Datatype       C     type: MPI_Datatype)     Fortran types       Or TYPE(MPI_Datatype)     Fortran types     DOUBLE_COMPLEX       MPI_DOUBLE_COMPLEX     DOUBLE_COMPLEX     DOUBLE COMPLEX       MPI_INTEGER1     INTEGER*1     INTEGER*1       MPI_INTEGER8     INTEGER*8     INTEGER*8       MPI_INTEGER16     INTEGER*16     INTEGER*16       MPI_REAL2     REAL*2     REAL*2       MPI_REAL4     REAL*4     MPI_REAL4       MPI_REAL8     REAL*8     MPI_COMPLEX4       MPI_COMPLEX4     COMPLEX*16     COMPLEX*16       MPI_COMPLEX32     COMPLEX*32     COMPLEX*32</long></long></long></long></long></double></float>	20	• -			
23     MPI_CXX_DOUBLE_COMPLEX     std::complex <double>       24     MPI_CXX_LONG_DOUBLE_COMPLEX     std::complex<long double="">       25     1 If an accompanying C++ compiler is missing, then the     MPI datatypes in this table are not defined.       26     Optional datatypes (Fortran)     Fortran types       27     C type: MPI_Datatype     Fortran types       30     Fortran type: INTEGER     or TYPE(MPI_Datatype)       31     or TYPE(MPI_Datatype)     DOUBLE COMPLEX       33     MPI_DOUBLE_COMPLEX     DOUBLE COMPLEX       34     MPI_INTEGER1     INTEGER*1       35     MPI_INTEGER8     INTEGER*8       37     MPI_INTEGER8     INTEGER*16       38     MPI_REAL2     REAL*2       39     MPI_REAL2     REAL*2       34     MPI_REAL4     REAL*2       35     MPI_REAL4     REAL*8       41     MPI_REAL6     REAL*8       42     MPI_COMPLEX4     COMPLEX*4       43     MPI_COMPLEX16     COMPLEX*32       44     MPI_COMPLEX32     COMPLEX*32</long></double>	21	MPI_CXX_BOOL		bool	
MPI_CXX_DOUBLE_COMPLEX std::complex <double> std::complex<long double=""> std::complex</long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></long></double>	22	MPI_CXX_FLOAT_COMPLEX		std:	:complex <float></float>
MPI_CXX_LONG_DOUBLE_COMPLEX       std::complex <long double="">         If an accompanying C++ compiler is missing, then the         MPI datatypes in this table are not defined.         Optional datatypes (Fortran)       Fortran types         C type: MPI_Datatype         Fortran type: INTEGER         or TYPE(MPI_Datatype)         MPI_DOUBLE_COMPLEX       DOUBLE COMPLEX         MPI_DOUBLE_COMPLEX       DOUBLE COMPLEX         MPI_INTEGER1       INTEGER*1         MPI_INTEGER2       INTEGER*4         MPI_INTEGER8       INTEGER*8         MPI_INTEGER16       INTEGER*8         MPI_INTEGER8       INTEGER*8         MPI_INTEGER8       INTEGER*16         MPI_REAL2       REAL*2         MPI_REAL4       REAL*2         MPI_REAL4       REAL*2         MPI_REAL8       REAL*4         MPI_COMPLEX4       COMPLEX*4         MPI_COMPLEX4       COMPLEX*4         MPI_COMPLEX32       COMPLEX*32</long>	23				-
1       If an accompanying C++ compiler is missing, then the         MPI datatypes in this table are not defined.         Particular         Optional datatypes (Fortran)         Fortran type:         NPI_Datatype         Or TYPE(MPI_Datatype)         Or TYPE(MPI_Datatype)         MPI_DOUBLE_COMPLEX       DOUBLE COMPLEX         MPI_INTEGER1       INTEGER*1         MPI_INTEGER8       INTEGER*2         MPI_INTEGER8       INTEGER*4         MPI_INTEGER16       INTEGER*88         MPI_INTEGER8       INTEGER*16         MPI_REAL2       REAL*2         MPI_REAL4       REAL*2         MPI_REAL4       REAL*4         MPI_COMPLEX4       COMPLEX*4         MPI_COMPLEX8       COMPLEX*8         MPI_COMPLEX16       COMPLEX*16         MPI_COMPLEX32       COMPLEX*32	24		LEX		-
26       MPI datatypes in this table are not defined.         27         28       Optional datatypes (Fortran)       Fortran types         29       C type: MPI_Datatype       Fortran types INTEGER         30       Fortran type: INTEGER       DOUBLE COMPLEX         31       or TYPE(MPI_Datatype)       DOUBLE COMPLEX         32       MPI_DOUBLE_COMPLEX       DOUBLE COMPLEX         33       MPI_INTEGER1       INTEGER*1         34       MPI_INTEGER2       INTEGER*2         35       MPI_INTEGER8       INTEGER*8         37       MPI_INTEGER16       INTEGER*16         38       MPI_REAL2       REAL*2         39       MPI_REAL4       REAL*4         40       MPI_REAL4       REAL*8         41       MPI_REAL6       REAL*8         42       MPI_COMPLEX4       COMPLEX*4         43       MPI_COMPLEX8       COMPLEX*8         44       MPI_COMPLEX16       COMPLEX*16         45       MPI_COMPLEX32       COMPLEX*32	25	$^{-1}$ If an accompanying C++ comp	piler i		
2728Optional datatypes (Fortran)Fortran types29C type: MPI_Datatype30Fortran type: INTEGER31or TYPE(MPI_Datatype)32MPI_DOUBLE_COMPLEXDOUBLE COMPLEX33MPI_INTEGER1INTEGER*134MPI_INTEGER2INTEGER*235MPI_INTEGER8INTEGER*837MPI_REAL2REAL*239MPI_REAL2REAL*239MPI_REAL4REAL*440MPI_REAL4REAL*841MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*324747K	26				
29C type: MPI_Datatype30Fortran type: INTEGER31or TYPE(MPI_Datatype)32MPI_DOUBLE_COMPLEX33MPI_INTEGER134MPI_INTEGER235MPI_INTEGER436MPI_INTEGER837MPI_INTEGER1638MPI_REAL239MPI_REAL440MPI_REAL441MPI_REAL442MPI_COMPLEX443MPI_COMPLEX444MPI_COMPLEX445MPI_COMPLEX3247COMPLEX432	27	01			
30Fortran type: INTEGER31or TYPE (MPI_Datatype)32MPI_DOUBLE_COMPLEXDOUBLE COMPLEX33MPI_INTEGER1INTEGER*134MPI_INTEGER2INTEGER*235MPI_INTEGER4INTEGER*436MPI_INTEGER8INTEGER*837MPI_REAL2REAL*238MPI_REAL2REAL*239MPI_REAL4REAL*440MPI_REAL4REAL*441MPI_REAL6REAL*841MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*8844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32	28	Optional datatypes (I	Fortr	an)	Fortran types
31or TYPE(MPI_Datatype)32MPI_DOUBLE_COMPLEXDOUBLE_COMPLEX33MPI_INTEGER1INTEGER*134MPI_INTEGER2INTEGER*235MPI_INTEGER4INTEGER*436MPI_INTEGER8INTEGER*837MPI_REAL2REAL*238MPI_REAL2REAL*239MPI_REAL4REAL*440MPI_REAL4REAL*841MPI_REAL16REAL*842MPI_COMPLEX4COMPLEX*4443MPI_COMPLEX8COMPLEX*8844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32	29	C type: MPI_Datatype			
32MPI_DOUBLE_COMPLEXDOUBLE COMPLEX33MPI_INTEGER1INTEGER*134MPI_INTEGER2INTEGER*235MPI_INTEGER4INTEGER*436MPI_INTEGER8INTEGER*837MPI_INTEGER16INTEGER*1638MPI_REAL2REAL*239MPI_REAL4REAL*440MPI_REAL8REAL*841MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*844MPI_COMPLEX32COMPLEX*32	30	Fortran type: INTEGER			
33MPI_INTEGER1INTEGER*134MPI_INTEGER2INTEGER*235MPI_INTEGER4INTEGER*436MPI_INTEGER8INTEGER*837MPI_REAL2REAL*238MPI_REAL4REAL*239MPI_REAL4REAL*440MPI_REAL8REAL*841MPI_COMPLEX4COMPLEX*442MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32	31	or TYPE(MPI_Datatype)			
33MPI_INTEGER1INTEGER*134MPI_INTEGER2INTEGER*235MPI_INTEGER4INTEGER*436MPI_INTEGER8INTEGER*837MPI_REAL2REAL*238MPI_REAL4REAL*239MPI_REAL4REAL*440MPI_REAL8REAL*841MPI_COMPLEX4COMPLEX*442MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32	32	MPI_DOUBLE_COMPLEX			DOUBLE COMPLEX
MPI_INTEGER2INTEGER*235MPI_INTEGER4INTEGER*436MPI_INTEGER8INTEGER*837MPI_INTEGER16INTEGER*1638MPI_REAL2REAL*239MPI_REAL4REAL*440MPI_REAL8REAL*841MPI_REAL16REAL*1642MPI_COMPLEX4COMPLEX*4443MPI_COMPLEX8COMPLEX*8844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*324647A	33	MPI_INTEGER1			INTEGER*1
35MPI_INTEGER4INTEGER*436MPI_INTEGER8INTEGER*837MPI_INTEGER16INTEGER*1638MPI_REAL2REAL*239MPI_REAL4REAL*440MPI_REAL8REAL*841MPI_REAL16REAL*1642MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*8844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32	34				
36MPI_INTEGER8INTEGER*837MPI_INTEGER16INTEGER*1638MPI_REAL2REAL*239MPI_REAL4REAL*440MPI_REAL8REAL*841MPI_REAL16REAL*1642MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32	35	MPI_INTEGER4			INTEGER*4
MPI_INTEGENTO INTEGENTO NPI_GENTO NP	36	MPI_INTEGER8			INTEGER*8
MPI_REAL2REAL*239MPI_REAL4REAL*440MPI_REAL8REAL*841MPI_REAL16REAL*1642MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32464747	37	MPI_INTEGER16			INTEGER*16
40MPI_REAL8REAL*441MPI_REAL16REAL*1642MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32464747	38	MPI_REAL2			REAL*2
40MPI_REAL8REAL*841MPI_REAL16REAL*1642MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32464747	39	MPI_REAL4			REAL*4
41MPI_REAL16REAL*1642MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32464747	40	_			REAL*8
42MPI_COMPLEX4COMPLEX*443MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32464747	41	_			REAL*16
43MPI_COMPLEX8COMPLEX*844MPI_COMPLEX16COMPLEX*1645MPI_COMPLEX32COMPLEX*32464747	42	—			COMPLEX*4
44     MPI_COMPLEX16     COMPLEX*16       45     MPI_COMPLEX32     COMPLEX*32       46     47     46	43	_			
45 MPI_COMPLEX32 COMPLEX*32 46	44	—			
46 47	45	_			
	46				
48	47				
	48				

Γ	Datatypes for reduction functions (C)	1
	type: MPI_Datatype	2
	ortran type: INTEGER or TYPE(MPI_Datatype)	3
N	IPI_FLOAT_INT	4
M	IPI_DOUBLE_INT	5
Ν	IPI_LONG_INT	6
N	IPI_2INT	7
N	IPI_SHORT_INT	8
Ν	IPI_LONG_DOUBLE_INT	9
Data	atypes for reduction functions (Fortran)	10
	pe: MPI_Datatype	- 12
	an type: INTEGER or TYPE(MPI_Datatype)	13
	2REAL	- 14
	2DOUBLE_PRECISION	15
_	2INTEGER	16
		- 17
_	Reserved communicators	18
	C type: MPI_Comm	19
_	Fortran type: INTEGER or TYPE(MPI_Comm)	20
	MPI_COMM_WORLD	21
	MPI_COMM_SELF	22
_		23 24
-	Communicator split type constants	24
	C type: const int (or unnamed enum)	25
_	Fortran type: INTEGER	20
_	MPI_COMM_TYPE_SHARED	27
Result	s of communicator and group comparison	
C type:	const int (or unnamed enum)	30
Fortran	type: INTEGER	31
MPI_IDE	ENT	32
MPI_CO	NGRUENT	33
MPI_SIN	/ILAR	34
MPI_UN	EQUAL	35
	Environmental inquiry info key	36
-	Environmental inquiry info key	38
	C type: MPI_Info	39
-	Fortran type: INTEGER or TYPE(MPI_Info)	40
-	MPI_INFO_ENV	41
	Environmental inquiry keys	42
	C type: const int (or unnamed enum)	43
	Fortran type: INTEGER	44
	MPI_TAG_UB	45
	MPI_IO	46
	MPI_HOST	47
	MPI_WTIME_IS_GLOBAL	48

Collective Operations
C type: MPI_Op
Fortran type: INTEGER or TYPE(MPI_Op)
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_MAXLOC
MPI_MINLOC
MPI_BAND
MPI_BOR
MPI_BXOR
MPI_LAND
MPI_LOR
MPI_LXOR
MPI_REPLACE
MPI_NO_OP
Null Handles
C/Fortran name
C type / Fortran type
MPI_GROUP_NULL
MPI_Group / INTEGER or TYPE(MPI_Group)
MPI_COMM_NULL
MPI_Comm / INTEGER or TYPE(MPI_Comm)
MPI_DATATYPE_NULL
MPI_Datatype / INTEGER or TYPE(MPI_Datatype)
MPI_REQUEST_NULL
MPI_Request / INTEGER or TYPE(MPI_Request) MPI_OP_NULL
MPI_Op / INTEGER or TYPE(MPI_Op)
MPI_ERRHANDLER_NULL
MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)
MPI_FILE_NULL
MPI_File / INTEGER or TYPE(MPI_File)
MPI_INFO_NULL
MPI_Info / INTEGER or TYPE(MPI_Info)
MPI_WIN_NULL
MPI_Win / INTEGER or TYPE(MPI_Win)
MPI_WIN / INTEGER OF TIPE(MFI_WIN) MPI_MESSAGE_NULL
<pre>MPI_Message / INTEGER or TYPE(MPI_Message)</pre>
Empty group
C type: MPI_Group
·
Fortran type: INTEGER or TYPE(MP1_Group)
Fortran type: INTEGER or TYPE(MPI_Group) MPI_GROUP_EMPTY

	Topologies
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_GRAPH
	MPI_CART
	MPI_DIST_GRAPH
	Predefined functions
Fortran name	
C type	
/ Fortran type with mp	i module / Fortran type with mpi_f08 module
PI_COMM_NULL_CO	PY_FN
MPI_Comm_copy_attr_:	function
/ COMM_COPY_ATTR_FUN	$\label{eq:comm_copy_attr_function} \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \$
PI_COMM_DUP_FN	
MPI_Comm_copy_attr_:	
/ COMM_COPY_ATTR_FUN	
PI_COMM_NULL_DEI	
MPI_Comm_delete_att	-
/ COMM_DELETE_ATTR_F	· · · · · · · · · · · · · · · · · · ·
PI_WIN_NULL_COPY	
<pre>MPI_Win_copy_attr_f / WIN_COPY_ATTR_FUNC</pre>	
PI_WIN_DUP_FN	SILON / FROCEDORE(MF1_WIIL_COPY_attr_runction) )
MPI_Win_copy_attr_f	unction
/ WIN_COPY_ATTR_FUNC	
PI_WIN_NULL_DELE	, ,
MPI_Win_delete_attr	
/ WIN_DELETE_ATTR_FU	
PI_TYPE_NULL_COP	
MPI_Type_copy_attr_:	—
/ TYPE_COPY_ATTR_FUN	_
PI_TYPE_DUP_FN	, JI _ IJ , ,
MPI_Type_copy_attr_	function
/ TYPE_COPY_ATTR_FUN	_
PI_TYPE_NULL_DELI	ETE_FN
MPI_Type_delete_att:	r_function
/ TYPE_DELETE_ATTR_F	FUNCTION       / PROCEDURE(MPI_Type_delete_attr_function)       1)
PI_CONVERSION_FN	
MPI_Datarep_convers	
/ DATAREP_CONVERSION	
	ementors (on page 272) and advice to users (on page 272)
on the predefined For	rtran functions MPI_COMM_NULL_COPY_FN, in
-	
Section $6.7.2$ .	
Section 6.7.2.	

1	Deprecated predefined functions
2	C/Fortran name
3	C type / Fortran type with mpi module
4	MPI_NULL_COPY_FN
5	MPI_Copy_function / COPY_FUNCTION
6	MPI_DUP_FN
7	MPI_Copy_function / COPY_FUNCTION
8	MPI_NULL_DELETE_FN
9	MPI_Delete_function / DELETE_FUNCTION
10	
11	Predefined Attribute Keys
12	C type: const int (or unnamed enum)
13	Fortran type: INTEGER
14	MPI_APPNUM
15	MPI_LASTUSEDCODE
16	MPI_UNIVERSE_SIZE
17	MPI_WIN_BASE
18	MPI_WIN_DISP_UNIT
19	MPI_WIN_SIZE
20	MPI_WIN_CREATE_FLAVOR
21	MPI_WIN_MODEL
22	
23	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	<b>MPI</b> 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	
44	
45	
40	
47	
40	

Mode Constants	1
C type: const int (or unnamed enum)	2
Fortran type: INTEGER	3
MPI_MODE_APPEND	4
MPI_MODE_CREATE	5
MPI_MODE_DELETE_ON_CLOSE	6
MPI_MODE_EXCL	7
MPI_MODE_NOCHECK	8
MPI_MODE_NOPRECEDE	9
MPI_MODE_NOPUT	10
MPI_MODE_NOSTORE	11
MPI_MODE_NOSUCCEED	12
MPI_MODE_RDONLY	13
MPI_MODE_RDWR	14
MPI_MODE_SEQUENTIAL	15
MPI_MODE_UNIQUE_OPEN	16
MPI_MODE_WRONLY	17
	18
Datatype Decoding Constants	19
C type: const int (or unnamed enum)	20
Fortran type: INTEGER	21
MPI_COMBINER_CONTIGUOUS	22
MPI_COMBINER_DARRAY	23
MPI_COMBINER_DUP	24
MPI_COMBINER_F90_COMPLEX	25
MPI_COMBINER_F90_INTEGER	26
MPI_COMBINER_F90_REAL	27
MPI_COMBINER_HINDEXED	28
MPI_COMBINER_HVECTOR	29 30
MPI_COMBINER_INDEXED_BLOCK	
MPI_COMBINER_HINDEXED_BLOCK	31 32
MPI_COMBINER_INDEXED	32
MPI_COMBINER_NAMED	34
MPI_COMBINER_RESIZED	35
MPI_COMBINER_STRUCT	36
MPI_COMBINER_SUBARRAY	37
MPI_COMBINER_VECTOR	38
Threads Constants	39
C type: const int (or unnamed enum)	40
Fortran type: INTEGER	41
MPI_THREAD_FUNNELED	42
MPI_THREAD_MULTIPLE	43
MPI_THREAD_SERIALIZED	44
MPI_THREAD_SINGLE	45
	46
	47

C type: const MPL_Offset (or unnamed enum)         Fortran type: INTEGER (KIND=MPL_OFFSET_KIND)         MPL_DISPLACEMENT_CURRENT         File Operation Constants, Part 2         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPL_DISTRIBUTE_EQULC         MPL_DISTRIBUTE_NONE         MPL_DISTRIBUTE_NONE         MPL_ORDER_C         MPL_ORDER_C         MPL_SEEK_END         MPL_SEEK_SET         Fortran type: INTEGER         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPL_ARGV_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPL_STATUS_S.GNORE         MPL_STATUS_S.GNORE         MPL_Status* / INTEGER, DIMENSION (MPI_STATUS_SIZE, +)         or TYPE(MPI_Status), DIMENSION (MPI_STATUS_SIZE)         or TYPE(MPI_Status), DIMENSION (MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_STATUSE, INTEGER array <th>1</th> <th>File Operation Constants, Part 1</th>	1	File Operation Constants, Part 1
Fortran type: INTEGER (KIND=MPI_OFFSET_KIND)         MPI_DISFLACEMENT_CURRENT         File Operation Constants, Part 2         C type: const int (or unnamed enum)         Portran type: INTEGER         MPI_DISTRIBUTE_BLOCK         MPI_DISTRIBUTE_OPLI_DARG         MPI_DISTRIBUTE_OPLI_DARG         MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_NONE         MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_END         MPI_SEEK_SET         P90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_INTEGER         MPI_ARGV_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_STATUS_S_IGNORE         MPI_STATUS_CONCR         MPI_STATUS_CONCRE         MPI_STATUS_CONCRE         MPI_STATUS_CONCRE         MPI_STATUS_CONCRE         MPI_STATUS_CO		
MPI_DISPLACEMENT_CURRENT         File Operation Constants, Part 2         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_DISTRIBUTE_BLOCK         MPI_DISTRIBUTE_CYCLIC         MPI_ORDER_C         MPI_ORDER_C         MPI_ORDER_FORTRAN         MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_SET         F00 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_NITEGER         MPI_ARGVS_NULL         char*** / array of CHARACTER*(*)         MPI_ARGVS_NULL         char*** / array of CHARACTER*(*)         MPI_RERCODES_IGNORE         int* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status)         MPI_STATUS_ICHORE         MPI_VUNUEIGHTED		· - / /
File Operation Constants, Part 2         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPL_DISTRIBUTE_BLOCK         MPL_DISTRIBUTE_OPLT_DARG         MPL_ORDER_C         MPL_ORDER_CONR         MPL_SEEK_CUR         MPL_SEEK_CUR         MPL_SEEK_CUR         MPL_SEEK_CUR         MPLSEEK_SET         See Constants Specifying Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_TYPECLASS_COMPLEX         MPL_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPL_ARGV_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPL_RRCODES_IGNORE         int* / INTEGER array         MPL_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status)         MPI_UNWEIGHTED<		
File Operation Constants, Part 2         C type: const int (or unmamed enum)         Fortran type: INTEGER         MPL_DISTRIBUTE_BLOCK         MPL_DISTRIBUTE_OPLI_DARG         MPL_ORDER_C         MPL_ORDER_C         MPL_SEEK_CUR         MPL_SEEK_CUR         MPL_SEEK_CUR         MPL_SEEK_END         MPL_SEEK_SET         B         F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPL_TYPECLASS_COMPLEX         MPL_AGVS_NULL         C/Fortran name         C type / Fortran type1         MPL_ARGVS_NULL         char*** / array of CHARACTER*(*)         MPL_SERCODES_IGNORE         int* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE),*)         or TYPE(MPI_Status)         MPI_UNWEIGHTED		
7       C type: const int (or unnamed enum)         8       Fortran type: INTEGER         9       MPI_DISTRIBUTE_BLOCK         10       MPI_DISTRIBUTE_CVCLIC         11       MPI_DISTRIBUTE_NONE         12       MPI_ORDER_FORTRAN         13       MPI_SEEK_END         14       MPI_SEEK_END         15       MPI_SEEK_END         16       MPI_SEEK_END         17       MPI_SEEK_SET         18       Totran type: INTEGER         19       F90 Datatype Matching Constants         20       C type: const int (or unnamed enum)         17       MPI_SEEK_SET         18       MPI_TYPECLASS_COMPLEX         29       MPI_TYPECLASS_COMPLEX         20       C/Fortran type:         21       TYPECLASS_COMPLEX         22       MPI_TYPECLASS_COMPLEX         23       MPI_TYPECLASS_COMPLEX         24       MPI_TYPECLASS_COMPLEX         25       MPI_CONCE         26       Constants Specifying Empty or Ignored Input         27       C/Fortran name         28       C type / Fortran type <sup>1</sup> 29       MPI_ARGV_NULL         char** / 2-dim. array of CHARACTER*(*)		File Operation Constants Part 2
Fortran type: INTEGER         9       MPI_DISTRIBUTE_BLOCK         10       MPI_DISTRIBUTE_OPLT_DARG         11       MPI_DISTRIBUTE_NONE         12       MPI_ORDER_C         13       MPI_ORDER_CORTRAN         14       MPI_SEEK_CUR         15       MPI_SEEK_CUR         16       MPI_SEEK_END         17       MPI_SEEK_SET         18       F90 Datatype Matching Constants         19       Fortran type: INTEGER         19       Fortran type: INTEGER         19       Fortran type: INTEGER         10       MPI_TYPECLASS_COMPLEX         11       MPI_TYPECLASS_INTEGER         11       MPI_ARGVS_NULL         11       char*** / 2-dim. array of CHARACTER*(*)         11       MPI_ARGV_NULL         11       char** / array of CHARACTER*(*)         11       MPI_ESTATUSES_IGNORE         11       mPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         10       rTYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         11       MPI_STATUSES_IGNORE         11       MPI_STATUSES_IGNORE         11       MPI_STATUSES_IGNORE         11       MPI_STATUSES_IGNORE         11 <t< td=""><td></td><td></td></t<>		
9       MPI_DISTRIBUTE_BLOCK         11       MPI_DISTRIBUTE_CYCLIC         11       MPI_DISTRIBUTE_DLT_DARG         12       MPI_ORDER_C         13       MPI_ORDER_FORTRAN         14       MPI_SEEK_END         15       MPI_SEEK_END         16       MPI_SEEK_END         17       MPI_SEEK_SET         18       F90 Datatype Matching Constants         20       C type: const int (or unnamed enum)         8       Fortran type: INTEGER         24       MPI_TYPECLASS_COMPLEX         25       MPI_TYPECLASS_COMPLEX         26       Constants Specifying Empty or Ignored Input         27       C/Fortran name         28       C type / Fortran type1         29       MPI_ARGVS_NULL         20       char** / 2-dim. array of CHARACTER*(*)         31       MPI_STATUSES_IGNORE         32       int* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         33       MPI_STATUSES_IGNORE         34       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         35       MPI_STATUSES_IGNORE         36       MPI_STATUSES_IGNORE         37       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         38       M		
MPI_DISTRIBUTE_CYCLIC         MPI_DISTRIBUTE_DFLT_DARG         MPI_DISTRIBUTE_DFLT_DARG         MPI_ORDER_C         MPI_ORDER_FORTRAN         MPI_SEEK_END         MPI_SEEK_END         MPI_SEEK_END         MPI_SEEK_END         MPI_SEEK_SET         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         C/Fortran name         C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL         char*** / aray of CHARACTER*(*)         MPI_ERRCODES_IGNORE         int* / INTEGER array         MPI_STATUS_IGNORE         MPI_NUWEIGHTED         int* / INTEGER array         MPI_VE(MPI_Status)         MPI_UNUEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         MPI_WEIGHTS_EMPTY         MPI_Note that in Fortran these constants are not usable for initializa		* -
MPI_DISTRIBUTE_DFLT_DARG         MPI_DISTRIBUTE_NONE         MPI_ORDER_C         MPI_ORDER_FORTRAN         MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_SET         MPI_SEEK_SET         PODatatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type1         MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_Status* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UWEIGHTED         int* / INTEGER array         MPI_UWEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         'Note that in Fortran t		
MPI_DISTRIBUTE_NONE       MPI_ORDER_C       MPI_ORDER_FORTRAN       MPI_SEEK_CUR       MPI_SEEK_END       MPI_SEEK_END       MPI_SEEK_SET       Potran type: INTEGER       MPI_TYPECLASS_COMPLEX       MPI_TYPECLASS_UPUL       char*** / array of CHARACTER*(*)       MPI_SEATUS       MPI_STATUSES_IGNORE       MPI_STATUSES_IGNORE       MPI_STATUSES_IGNORE       MPI_STATUSES_IGNORE       MPI_STATUS_SIGNORE       MPI_STATUS_IGNORE       MPI_STATUS_SIGNORE       MPI_STATUS_SIGNORE       MPI_ST		
MPI_ORDER_C         MPI_ORDER_FORTRAN         MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_SET         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type1         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_UWEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY </td <td></td> <td></td>		
MPI_ORDER_FORTRAN         MPI_SEEK_CUR         MPI_SEEK_END         MPI_SEEK_SET         MPI_SEEK_SET         P         PO Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         C/Fortran name         C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char** / array of CHARACTER*(*)         MPI_Status* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         'Note that in Fortran these constants are not usable for initializatior         expressions or assignment. See Section 2.5.4.		
<sup>15</sup> MPI_SEEK_CUR <sup>16</sup> MPI_SEEK_END <sup>17</sup> MPI_SEEK_SET <sup>18</sup> <b>F90 Datatype Matching Constants</b> <sup>19</sup> C type: const int (or unnamed enum) <sup>21</sup> Fortran type: INTEGER <sup>22</sup> MPI_TYPECLASS_COMPLEX <sup>23</sup> MPI_TYPECLASS_REAL <sup>24</sup> Constants Specifying Empty or Ignored Input <sup>25</sup> C/Fortran name <sup>26</sup> Constants Specifying Empty or Ignored Input <sup>27</sup> C/Fortran name <sup>28</sup> C type / Fortran type <sup>1</sup> <sup>29</sup> MPI_ARGVS_NULL <sup>20</sup> char*** / 2-dim. array of CHARACTER*(*) <sup>31</sup> MPI_ERCODES_ICNORE <sup>32</sup> char** / array of CHARACTER*(*) <sup>33</sup> MPI_ERCODES_ICNORE <sup>34</sup> int* / INTEGER array <sup>35</sup> MPI_STATUSES_IGNORE <sup>36</sup> MPI_STATUS_SIZE, NIMENSION(MPI_STATUS_SIZE, *) <sup>37</sup> or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE) <sup>38</sup> MPI_UNWEIGHTED <sup>34</sup> MPI_UNWEIGHTED <sup>35</sup> MPI_UNWEIGHTED <sup>36</sup> MPI_WEIGER array		
MPI_SEEK_END         MPI_SEEK_SET         Portantype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type¹         MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_ERCODES_IGNORE         int* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         'Note that in Fortran these constants are not usable for initializatior         expressions or a		
Impl_SEEK_SET         Impl_Seek_SEEk_SET         Impl_Seek_SEK         Impl_Seek_SEK <thimpl_seek_sek_sek< th=""></thimpl_seek_sek_sek<>		
INTEGER         F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Portran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C/Fortran name         C/Fortran name         C/Fortran name         MPI_ARGVS_NULL         char** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char** / array of CHARACTER*(*)         MPI_Status* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE),*)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         MPI_UWWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         MPI_WEIGHTS_EMPTY         MIT         MPI weight in Fortran these constants are not usable for initialization         expressions or assignment. See Section 2.5.4.		
F90 Datatype Matching Constants         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type1         MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_STATUSES_IGNORE         int* / INTEGER array         MPI_STATUSE_SIGNORE         MPI_STATUSE_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_SIGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_GRORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE         MPI_STATUS_IGNORE		
20       C type: const int (or unnamed enum)         21       Fortran type: INTEGER         22       MPI_TYPECLASS_COMPLEX         23       MPI_TYPECLASS_INTEGER         24       MPI_TYPECLASS_REAL         25       Constants Specifying Empty or Ignored Input         27       C/Fortran name         28       C type / Fortran type <sup>1</sup> 29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSE_IGNORE         34       int* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         36       MPI_STATUS_IGNORE         37       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         31       or TYPE(MPI_Status)         32       or TYPE(MPI_Status)         33       MPI_UNWEIGHTED         34       int* / INTEGER array         35       MPI_WEIGHTS_EMPTY         36       int* / INTEGER array         37<		
21       Fortran type: INTEGER         22       MPI_TYPECLASS_COMPLEX         23       MPI_TYPECLASS_INTEGER         24       MPI_TYPECLASS_REAL         25       Constants Specifying Empty or Ignored Input         27       C/Fortran name         28       C type / Fortran type <sup>1</sup> 29       MPI_ARGVS_NULL         30       char** / 2-dim. array of CHARACTER* (*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER* (*)         33       MPI_STATUSES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         34       int* / INTEGER, DIMENSION (MPI_STATUS_SIZE, *)         35       MPI_Status* / INTEGER, DIMENSION (MPI_STATUS_SIZE, *)         36       MPI_STATUS_IGNORE         37       or TYPE(MPI_Status), DIMENSION (MPI_STATUS_SIZE)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION (MPI_STATUS_SIZE)         31       or TYPE(MPI_Status)         32       model array         34       MPI_UNWEIGHTED         35       MPI_UNWEIGHTS_EMPTY         36       int* / INTEGER array         37       Note that in Fortran these constants are		
Pottal type: Integer         MPI_TYPECLASS_COMPLEX         MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char*** / array of CHARACTER*(*)         MPI_RECODES_IGNORE         int* / INTEGER array         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUSES_IGNORE         MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         INote that in Fortran these constants are not usable for initialization         expressions or assignment. See Section 2.5.4.		
23       MPI_TYPECLASS_INTEGER         24       MPI_TYPECLASS_REAL         25       Constants Specifying Empty or Ignored Input         27       C/Fortran name         28       C type / Fortran type1         29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_STATUSES_IGNORE         37       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         38       MPI_STATUS_IGNORE         39       MPI_STATUS_IGNORE         39       MPI_STATUS_IGNORE         39       MPI_STATUS_IGNORE         30       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         31       OFTYPE(MPI_Status), MPI_STATUS_SIZE)         32       or TYPE(MPI_Status)         34       MPI_UNWEIGHTED         35       MPI_WEIGHTS_EMPTY         36       MPI_WEIGHTS_EMPTY         37       int* / INTEGER array         38       MPI_WEIGHTS_EMPTY         39       int* / INTEGER array <td></td> <td></td>		
MPI_TYPECLASS_INTEGER         MPI_TYPECLASS_REAL         MPI_TYPECLASS_REAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type1         MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char** / array of CHARACTER*(*)         MPI_ERRCODES_IGNORE         int* / INTEGER array         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(*)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         Note that in Fortran these constants are not usable for initialization         expressions or assignment. See Section 2.5.4.		
MPI_TTPECLASS_KEAL         MPI_TTPECLASS_KEAL         Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type <sup>1</sup> MPI_ARGVS_NULL       char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL       char*** / array of CHARACTER*(*)         MPI_ERCODES_IGNORE       MPI_ERCODES_IGNORE         MPI_Status* / INTEGER array       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or       TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)       or         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)       or       TYPE(MPI_Status)         MPI_UNWEIGHTED       int* / INTEGER array       MPI_UNWEIGHTED         int* / INTEGER array       MPI_WEIGHTS_EMPTY       int* / INTEGER array         MPI_WEIGHTS_EMPTY       int* / INTEGER array       Int* / INTEGER array		
Constants Specifying Empty or Ignored Input         C/Fortran name         C type / Fortran type1         MPI_ARGVS_NULL         char*** / 2-dim. array of CHARACTER*(*)         MPI_ARGV_NULL         char** / array of CHARACTER*(*)         MPI_ERRCODES_IGNORE         int* / INTEGER array         MPI_STATUSES_IGNORE         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         or TYPE(MPI_Status), DIMENSION(*)         MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         or TYPE(MPI_Status)         MPI_UNWEIGHTED         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         MPI_WEIGHTS_EMPTY         int* / INTEGER array         I Note that in Fortran these constants are not usable for initialization         expressions or assignment. See Section 2.5.4.		MPI_TYPECLASS_REAL
27       C/Fortran name         28       C type / Fortran type <sup>1</sup> 29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 <sup>1</sup> Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.		Constants Specifying Empty or Ignored Input
28       C type / Fortran type <sup>1</sup> 29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 <sup>1</sup> Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	27	
29       MPI_ARGVS_NULL         30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45       1 Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	28	
30       char*** / 2-dim. array of CHARACTER*(*)         31       MPI_ARGV_NULL         32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_STATUSES_IGNORE         37       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE,*)         38       MPI_STATUS_IGNORE         39       MPI_STATUS_IGNORE         34       or TYPE(MPI_Status), DIMENSION(MPI_STATUS_SIZE)         36       MPI_STATUS_IGNORE         37       or TYPE(MPI_Status)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 <sup>1</sup> Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	29	*- , *-
<ul> <li>MPI_ARGV_NULL</li> <li>char** / array of CHARACTER*(*)</li> <li>MPI_ERRCODES_IGNORE</li> <li>int* / INTEGER array</li> <li>MPI_STATUSES_IGNORE</li> <li>MPI_STATUSES_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)</li> <li>or TYPE(MPI_Status), DIMENSION(*)</li> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	30	
32       char** / array of CHARACTER*(*)         33       MPI_ERRCODES_IGNORE         34       int* / INTEGER array         35       MPI_STATUSES_IGNORE         36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 <sup>1</sup> Note that in Fortran these constants are not usable for initialization         46       expressions or assignment. See Section 2.5.4.	31	
<ul> <li>MPI_ERRCODES_IGNORE</li> <li>int* / INTEGER array</li> <li>MPI_STATUSES_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)</li> <li>or TYPE(MPI_Status), DIMENSION(*)</li> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	32	
<ul> <li>int* / INTEGER array</li> <li>MPI_STATUSES_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)</li> <li>or TYPE(MPI_Status), DIMENSION(*)</li> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	33	, .
<ul> <li>MPI_STATUSES_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)</li> <li>or TYPE(MPI_Status), DIMENSION(*)</li> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	34	
36       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)         37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 <sup>1</sup> Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.	35	,
37       or TYPE(MPI_Status), DIMENSION(*)         38       MPI_STATUS_IGNORE         39       MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)         40       or TYPE(MPI_Status)         41       MPI_UNWEIGHTED         42       int* / INTEGER array         43       MPI_WEIGHTS_EMPTY         44       int* / INTEGER array         45 <sup>1</sup> Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.	36	
<ul> <li>MPI_STATUS_IGNORE</li> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li>Note that in Fortran these constants are not usable for initialization</li> <li>expressions or assignment. See Section 2.5.4.</li> </ul>	37	,
<ul> <li>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)</li> <li>or TYPE(MPI_Status)</li> <li>MPI_UNWEIGHTED</li> <li>int* / INTEGER array</li> <li>MPI_WEIGHTS_EMPTY</li> <li>int* / INTEGER array</li> <li><sup>1</sup> Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.</li> </ul>	38	
40       or TYPE(MPI_Status)         41       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 initialization expressions or assignment. See Section 2.5.4.	39	
<ul> <li><sup>41</sup> MPI_UNWEIGHTED</li> <li><sup>42</sup> int* / INTEGER array</li> <li><sup>43</sup> MPI_WEIGHTS_EMPTY</li> <li><sup>44</sup> int* / INTEGER array</li> <li><sup>45</sup> <sup>1</sup> Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.</li> </ul>	40	
<ul> <li><sup>42</sup> int* / INTEGER array</li> <li><sup>43</sup> MPI_WEIGHTS_EMPTY</li> <li><sup>44</sup> int* / INTEGER array</li> <li><sup>45</sup> <sup>1</sup> Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.</li> </ul>	41	
<ul> <li><sup>43</sup> MPI_WEIGHTS_EMPTY</li> <li><sup>44</sup> int* / INTEGER array</li> <li><sup>45</sup> Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.</li> </ul>	42	
<sup>44</sup> <sup>45</sup> <sup>46</sup> <sup>47</sup> <sup>1</sup> Note that in Fortran these constants are not usable for initialization <sup>46</sup> expressions or assignment. See Section 2.5.4.	43	,
<sup>45</sup> <sup>1</sup> Note that in Fortran these constants are not usable for initialization <sup>46</sup> expressions or assignment. See Section 2.5.4.	44	
46 expressions or assignment. See Section 2.5.4.	45	
47	46	
48	47	on problems of abbighment. See Section 2.0.1.
	48	

C Constants Specify	ing Ignored Input (no Fortran)
type: MPI_Fint*	equivalent to Fortran
I_F_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi $/$ mpif.
PI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h
ype: MPI_F08_status*	equivalent to Fortran
PI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE $in mpi_f08$
I_F08_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi_f08
C preprocessor Con	stants and Fortran Parameters
	nacro that expands to an int value
Fortran type: INTEGER	1
MPI_SUBVERSION	
MPI_VERSION	
Null handles used in t	he MPI tool information interface
MPI_T_ENUM_NULL	
MPI_T_enum	
MPI_T_CVAR_HANDLE_NU	LL
MPI_T_cvar_handle	
MPI_T_PVAR_HANDLE_NU	LL
MPI_T_pvar_handle	
MPI_T_PVAR_SESSION_NU	ILL
MPI_T_pvar_session	
<b>T</b> T <b>1 1 1 1 1</b>	
	e MPI tool information interface
C type: const int (or unn	,
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_USER	-
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPID	
MPI_T_VERBOSITY_MPIC	
MPI_T_VERBOSITY_MPIC	

	C type: const int (or unnamed enum)
	MPI_T_BIND_NO_OBJECT
	MPI_T_BIND_MPI_COMM
	MPI_T_BIND_MPI_DATATYPE
	MPI_T_BIND_MPI_ERRHANDLER
	MPI_T_BIND_MPI_FILE
	MPI_T_BIND_MPI_GROUP
	MPI_T_BIND_MPI_OP
	MPI_T_BIND_MPI_REQUEST
	MPI_T_BIND_MPI_WIN
	MPI_T_BIND_MPI_MESSAGE
	MPI_T_BIND_MPI_INFO
	Constants describing the scope of a control variation
	in the MPI tool information interface
_	C type: const int (or unnamed enum)
	MPI_T_SCOPE_CONSTANT
	MPI_T_SCOPE_READONLY
	MPI_T_SCOPE_LOCAL
	MPI_T_SCOPE_GROUP
	MPI_T_SCOPE_GROUP_EQ
	MPI_T_SCOPE_ALL
_	MPI_T_SCOPE_ALL_EQ
	Additional constants used
	by the MPI tool information interface
	C type: MPI_T_pvar_handle
	MPI_T_PVAR_ALL_HANDLES
	Daufannan ar siniahlar alarara waad ha tha
	Performance variables classes used by the MPI tool information interface
	C type: const int (or unnamed enum)
	MPI_T_PVAR_CLASS_STATE MPI_T_PVAR_CLASS_LEVEL
	MPI_T_PVAR_CLASS_SIZE
	MPI_T_PVAR_CLASS_PERCENTAGE
	MPI_T_PVAR_CLASS_HIGHWATERMARK
	MPI_T_PVAR_CLASS_LOWWATERMARK
	MPI_T_PVAR_CLASS_COUNTER
	MPI_T_PVAR_CLASS_AGGREGATE
	MPI_T_PVAR_CLASS_AGGREGATE MPI_T_PVAR_CLASS_TIMER MPI_T_PVAR_CLASS_GENERIC

## A.1.2 Types

47

 $_{48}$  The following are defined C type definitions, included in the file mpi.h.

/* C opaque types */	1
MPI_Aint	2
MPI_Count	3
MPI_Fint	4
MPI_Offset	5
MPI_Status	6
MPI_F08_status	7
	8
/* C handles to assorted structures */	9
MPI_Comm	10
MPI_Datatype	11
MPI_Errhandler	12
MPI_File	13
MPI_Group	14
MPI_Info	15
MPI_Message	16
MPI_Op	17
MPI_Request	18
MPI_Win	19
	20
/* Types for the MPI_T interface */	21
MPI_T_enum	22
MPI_T_cvar_handle	23
MPI_T_pvar_handle	24
MPI_T_pvar_session	25
	26
	27
The following are defined Fortran type definitions, included in the mpi_f08 and mpi	28
modules.	29
modules.	30
! Fortran opaque types in the mpi_f08 and mpi modules	31
TYPE(MPI_Status)	32
	33
! Fortran handles in the mpi_f08 and mpi modules	34
TYPE(MPI_Comm)	35
TYPE(MPI_Datatype)	36
TYPE(MPI_Errhandler)	37
TYPE(MPI_File)	38
TYPE(MPI_Group)	39
TYPE(MPI_Info)	40
TYPE(MPI_Message)	41
TYPE(MPI_Op)	42
TYPE(MPI_Request)	43
TYPE(MPI_Win)	43
	45
	46
	40
	48
	-

1 A.1.3 Prototype Definitions  $\mathbf{2}$ C Bindings 3 4 The following are defined C typedefs for user-defined functions, also included in the file 5mpi.h. 6  $\overline{7}$ /\* prototypes for user-defined functions \*/ typedef void MPI\_User\_function(void \*invec, void \*inoutvec, int \*len, 8 9 MPI\_Datatype \*datatype); 10 11 typedef int MPI\_Comm\_copy\_attr\_function(MPI\_Comm oldcomm, int comm\_keyval, void \*extra\_state, void \*attribute\_val\_in, 12void \*attribute\_val\_out, int \*flag); 13 14typedef int MPI\_Comm\_delete\_attr\_function(MPI\_Comm comm, int comm\_keyval, void \*attribute\_val, void \*extra\_state); 151617typedef int MPI\_Win\_copy\_attr\_function(MPI\_Win oldwin, int win\_keyval, 18 void \*extra\_state, void \*attribute\_val\_in, 19 void \*attribute\_val\_out, int \*flag); typedef int MPI\_Win\_delete\_attr\_function(MPI\_Win win, int win\_keyval, 20void \*attribute\_val, void \*extra\_state); 212223typedef int MPI\_Type\_copy\_attr\_function(MPI\_Datatype oldtype,  $^{24}$ int type\_keyval, void \*extra\_state, void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag); 2526typedef int MPI\_Type\_delete\_attr\_function(MPI\_Datatype datatype, int type\_keyval, void \*attribute\_val, void \*extra\_state); 2728typedef void MPI\_Comm\_errhandler\_function(MPI\_Comm \*, int \*, ...); 29typedef void MPI\_Win\_errhandler\_function(MPI\_Win \*, int \*, ...); 30  $^{31}$ typedef void MPI\_File\_errhandler\_function(MPI\_File \*, int \*, ...); 3233 typedef int MPI\_Grequest\_query\_function(void \*extra\_state, 34 MPI\_Status \*status); typedef int MPI\_Grequest\_free\_function(void \*extra\_state); 35typedef int MPI\_Grequest\_cancel\_function(void \*extra\_state, int complete); 36 37 typedef int MPI\_Datarep\_extent\_function(MPI\_Datatype datatype, 3839 MPI\_Aint \*file\_extent, void \*extra\_state); typedef int MPI\_Datarep\_conversion\_function(void \*userbuf, 4041 MPI\_Datatype datatype, int count, void \*filebuf, 42MPI\_Offset position, void \*extra\_state); 43 44Fortran 2008 Bindings with the mpi\_f08 Module 45The callback prototypes when using the Fortran mpi\_f08 module are shown below: 46The user-function argument to MPI\_Op\_create should be declared according to: 47ABSTRACT INTERFACE 48

```
1
  SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
                                                                                   2
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                    3
      TYPE(C_PTR), VALUE :: invec, inoutvec
      INTEGER :: len
                                                                                   4
      TYPE(MPI_Datatype) :: datatype
                                                                                   5
                                                                                   6
    The copy and delete function arguments to MPI_Comm_create_keyval should be de-
                                                                                    7
clared according to:
                                                                                    8
ABSTRACT INTERFACE
                                                                                   9
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
                                                                                   10
  attribute_val_in, attribute_val_out, flag, ierror)
                                                                                   11
      TYPE(MPI_Comm) :: oldcomm
                                                                                   12
      INTEGER :: comm_keyval, ierror
                                                                                   13
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                   14
      attribute_val_out
                                                                                   15
      LOGICAL :: flag
                                                                                   16
                                                                                   17
ABSTRACT INTERFACE
                                                                                   18
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
                                                                                   19
  attribute_val, extra_state, ierror)
      TYPE(MPI_Comm) :: comm
                                                                                   20
                                                                                   21
      INTEGER :: comm_keyval, ierror
                                                                                   22
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                   23
   The copy and delete function arguments to MPI_Win_create_keyval should be declared
                                                                                   24
according to:
                                                                                   25
ABSTRACT INTERFACE
                                                                                   26
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
                                                                                   27
  attribute_val_in, attribute_val_out, flag, ierror)
                                                                                   28
      TYPE(MPI_Win) :: oldwin
                                                                                   29
      INTEGER :: win_keyval, ierror
                                                                                   30
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                   31
      attribute_val_out
                                                                                   32
      LOGICAL :: flag
                                                                                   33
                                                                                   34
ABSTRACT INTERFACE
                                                                                   35
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
                                                                                   36
  extra_state, ierror)
                                                                                   37
      TYPE(MPI_Win) :: win
                                                                                   38
      INTEGER :: win_keyval, ierror
                                                                                   39
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                   40
   The copy and delete function arguments to MPI_Type_create_keyval should be declared
                                                                                   41
according to:
                                                                                   42
ABSTRACT INTERFACE
                                                                                   43
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
                                                                                   44
  attribute_val_in, attribute_val_out, flag, ierror)
                                                                                   45
      TYPE(MPI_Datatype) :: oldtype
                                                                                   46
      INTEGER :: type_keyval, ierror
                                                                                   47
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                   48
```

1attribute\_val\_out  $\mathbf{2}$ LOGICAL :: flag 3 ABSTRACT INTERFACE 4 SUBROUTINE MPI\_Type\_delete\_attr\_function(datatype, type\_keyval, 5attribute\_val, extra\_state, ierror) 6 TYPE(MPI\_Datatype) :: datatype 7 INTEGER :: type\_keyval, ierror 8 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: attribute\_val, extra\_state 9 10 The handler-function argument to MPI\_Comm\_create\_errhandler should be declared 11 like this: 12ABSTRACT INTERFACE 13SUBROUTINE MPI\_Comm\_errhandler\_function(comm, error\_code) 14TYPE(MPI\_Comm) :: comm 15INTEGER :: error\_code 16The handler-function argument to MPI\_Win\_create\_errhandler should be declared like 17this: 18 ABSTRACT INTERFACE 19 SUBROUTINE MPI\_Win\_errhandler\_function(win, error\_code) 20TYPE(MPI\_Win) :: win 21INTEGER :: error\_code 22 23The handler-function argument to MPI\_File\_create\_errhandler should be declared like  $^{24}$ this: 25ABSTRACT INTERFACE 26SUBROUTINE MPI\_File\_errhandler\_function(file, error\_code) 27TYPE(MPI\_File) :: file 28INTEGER :: error\_code 29 The query, free, and cancel function arguments to MPI\_Grequest\_start should be de-30 clared according to:  $^{31}$ ABSTRACT INTERFACE 32 SUBROUTINE MPI\_Grequest\_query\_function(extra\_state, status, ierror) 33 TYPE(MPI\_Status) :: status 34 INTEGER :: ierror 35 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state 36 37 ABSTRACT INTERFACE 38 SUBROUTINE MPI\_Grequest\_free\_function(extra\_state, ierror) 39 INTEGER :: ierror 40INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state 41 ABSTRACT INTERFACE 42SUBROUTINE MPI\_Grequest\_cancel\_function(extra\_state, complete, ierror) 43 INTEGER :: ierror 44 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state 45 LOGICAL :: complete 4647 The extent and conversion function arguments to MPI\_Register\_datarep should be de-48

```
1
clared according to:
                                                                                     2
ABSTRACT INTERFACE
                                                                                     3
  SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
  ierror)
                                                                                     4
      TYPE(MPI_Datatype) :: datatype
                                                                                     5
                                                                                     6
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
      INTEGER :: ierror
                                                                                     7
ABSTRACT INTERFACE
                                                                                     9
  SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
                                                                                     10
  filebuf, position, extra_state, ierror)
                                                                                     11
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                    12
      TYPE(C_PTR), VALUE :: userbuf, filebuf
                                                                                    13
      TYPE(MPI_Datatype) :: datatype
                                                                                    14
      INTEGER :: count, ierror
                                                                                     15
      INTEGER(KIND=MPI_OFFSET_KIND) :: position
                                                                                     16
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                     17
                                                                                    18
                                                                                    19
Fortran Bindings with mpif.h or the mpi Module
                                                                                    20
With the Fortran mpi module or mpif.h, here are examples of how each of the user-defined
                                                                                    21
subroutines should be declared.
                                                                                    22
    The user-function argument to MPI_OP_CREATE should be declared like this:
                                                                                    23
                                                                                    24
SUBROUTINE USER_FUNCTION (INVEC, INOUTVEC, LEN, DATATYPE)
                                                                                    25
   <type> INVEC(LEN), INOUTVEC(LEN)
                                                                                     26
   INTEGER LEN, DATATYPE
                                                                                    27
    The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be
                                                                                    28
                                                                                    29
declared like these:
                                                                                    30
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
                                                                                    31
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                    32
   INTEGER OLDCOMM, COMM_KEYVAL, IERROR
                                                                                    33
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                    34
             ATTRIBUTE_VAL_OUT
                                                                                    35
   LOGICAL FLAG
                                                                                    36
                                                                                    37
SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
                                                                                    38
             EXTRA_STATE, IERROR)
                                                                                    39
   INTEGER COMM, COMM_KEYVAL, IERROR
                                                                                     40
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                    41
                                                                                    42
    The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be
                                                                                    43
declared like these:
                                                                                    44
                                                                                    45
SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
                                                                                     46
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                     47
   INTEGER OLDWIN, WIN_KEYVAL, IERROR
                                                                                     48
```

LOGICAL FLAG LUGICAL FLAG LUGIC	1	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
<pre>SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER WIN, WIN_KEYVAL, IERROR INTEGER (KIND=MP1_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE The copy and delete function arguments to MP1_TYPE_CREATE_KEYVAL should be declared like these: SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER (KIND=MP1_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER COMM_ERRHANDLER_FUNCTION(COMM_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE The bandler-function argument to MP1_VIN_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE The handler-function argument to MP1_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER WIN, ERROR_CODE The handler-function argument to MP1_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE The handler-function argument to MP1_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE The query, free, and cancel function arguments to MP1_GREQUEST_START should be declared like these: SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER (KIND=MP1_ADDRESS_KIND) EXTRA_STATE</pre>		ATTRIBUTE_VAL_OUT
EXTRA_STATE, IERROR)         INTEGER WIN, WIM_KEYVAL, IERROR         INTEGER(KIND-MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE         The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be         declared like these:         SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,         ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)         INTEGER OLDTYPE, TYPE_KEYVAL, IERROR         INTEGER (KIND-MPI_ADDRESS_KIND) EXTRA_STATE,         ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT         LOGICAL FLAG         SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,         EXTRA_STATE, IERROR)         INTEGER (KIND-MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE         INTEGER (KIND-MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE         INTEGER COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)         INTEGER COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)         INTEGER COMM, ERROR_CODE         The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-         clared like this:         SUBROUTINE WIN_ERRHANDLER_FUNCTION(VIN, ERROR_CODE)         INTEGER WIN, ERROR_CODE         The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-         clared like this:         SUBROUTINE FILE_ERRHANDLER_FUNCTION(VIN, ERROR_CODE)         INTEGER FILE, ERROR_CODE	4	
INTEGER WIN, WIN_KEYVAL, IERROR         INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE         INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE         INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE,         ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR         INTEGER (LIND=MPI_ADDRESS_KIND) EXTRA_STATE,         ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR         INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE,         ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT         LOGICAL FLAG         SUBROUTINE TYPE_DELETE_ATTR_FUNCTION (DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,         EXTRA_STATE, IERROR)         INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE         INTEGER COMM_ERRANDLER_FUNCTION (COMM_CREATE_ERRHANDLER should be de- clared like this:         SUBROUTINE COMM_ERRHANDLER_FUNCTION (COMM, ERROR_CODE)         INTEGER COMM, ERROR_CODE         The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this:         SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)         INTEGER WIN, ERROR_CODE         The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:         SUBROUTINE FILE_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)         INTEGER FILE, ERRON_CODE         The handler-function argument to MPI_GREQUEST_START should be declared like these:         SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE		
The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be declared like these: SUBROUTINE TYPE_COPY_ATTE_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER OLDTYPE, TYPE_KEYVAL, IERROR SUBROUTINE TYPE_DELETE_ATTE_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR INTEGER OLTATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER OLTATYPE, TYPE_KEYVAL, IERROR SUBROUTINE TYPE_DELETE_ATTE_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE The handler-function argument to MPI_GREQUEST_START should be declared like this: SUBROUTINE WIN_ERROR_CODE The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these: SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER FILE, ERROR_STATUS_SIZE), IERROR INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE	7	
The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be declared like these: SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER (KIND-MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT LOGICAL FLAG SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERROR_CODE The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERROR_CODE The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERROR_CODE The handler-function argument to MPI_GREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERROR_CODE The handler-function argument to MPI_GREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERROR_CODE The handler-function argument to MPI_GREATE_ERRHANDLER should be de- clared like this: SUBROUTINE FILE_ERHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these: SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER KIND=MPI_ADDRESS_KIND) EXTRA_STATE	8	INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
<ul> <li>declared like these:</li> <li>SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)</li> <li>INTEGER OLDTYPE, TYPE_KEYVAL, IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT</li> <li>LOGICAL FLAG</li> <li>SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)</li> <li>INTEGER DATATYPE, TYPE_KEYVAL, IERROR</li> <li>INTEGER ATATYPE, TYPE_KEYVAL, IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE</li> <li>The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)</li> <li>INTEGER COMM, ERROR_CODE</li> <li>The handler-function argument to MPI_VIN_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>	9	
<ul> <li>SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT</li> <li>LOGICAL FLAG</li> <li>SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE</li> <li>The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE</li> <li>The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER VIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER TIM, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERROR_CODE</li> <li>The handler-function argument to MPI_GREQUEST_START should be de- clared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE</li> <li>The handler-function argument to MPI_GREQUEST_START should be clared like this:</li> </ul>		
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,         ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR         INTEGER OLDTYPE, TYPE_KEYVAL, IERROR         INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE,         ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT         INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE,         INTEGER (ATATYPE, TYPE_KEYVAL, IERROR)         INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE         INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE         The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de-         clared like this:         SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)         INTEGER COMM, ERROR_CODE         The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-         clared like this:         SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)         INTEGER WIN, ERROR_CODE         The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-         clared like this:         SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)         INTEGER FILE, ERROR_CODE         The handler-function argument to MPI_GREQUEST_START should be de-         clared like this:         SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)         INTEGER FILE, ERROR_CODE         The query, free, and cancel function arguments to MPI_G		declared like these:
<ul> <li>INTEGER OLDTYPE, TYPE_KEYVAL, IERROR</li> <li>INTEGER(KIND-MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT</li> <li>LOGICAL FLAG</li> <li>SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)</li> <li>INTEGER DATATYPE, TYPE_KEYVAL, IERROR</li> <li>INTEGER CMM_PFI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE</li> <li>The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)</li> <li>INTEGER COMM, ERROR_CODE</li> <li>The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		
<ul> <li>INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT</li> <li>LOGICAL FLAG</li> <li>SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)</li> <li>INTEGER DATATYPE, TYPE_KEYVAL, IERROR</li> <li>INTEGER CKIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE</li> <li>The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE</li> <li>The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER WIN, ERROR_CODE</li> <li>The handler-function arguments to MPI_GREQUEST_START should be de- clared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		
17ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT18LOGICAL FLAG29SUBROUTINE TYPE_DELETE_ATTR_FUNCTION (DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)21INTEGER DATATYPE, TYPE_KEYVAL, IERROR23INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE24The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this:27SUBROUTINE COMM_ERRHANDLER_FUNCTION (COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE28SUBROUTINE WIN_ERRHANDLER_FUNCTION (COMM, ERROR_CODE) INTEGER WIN, ERROR_CODE30The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this:33SUBROUTINE WIN_ERRHANDLER_FUNCTION (WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE34The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:35SUBROUTINE FILE_ERRHANDLER_FUNCTION (FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE36The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:37SUBROUTINE FILE_ERRHANDLER_FUNCTION (FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE36The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:37SUBROUTINE GREQUEST_QUERY_FUNCTION (EXTRA_STATE, STATUS, IERROR) INTEGER STATUS (MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE		
LOGICAL FLAG SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER CATATYPE, TYPE_KEYVAL, IERROR The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these: SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE		•
20SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)21INTEGER DATATYPE, TYPE_KEYVAL, IERROR23INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE24The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this:27SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE30The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this:31SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE34SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE35SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE36SUBROUTINE FILE_ERHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE37SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE36SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE37SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE36SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE37SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE	18	
EXTRA_STATE, IERROR) EXTRA_STATE, IERROR INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these: SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	19	
<ul> <li>INTEGER DATATYPE, TYPE_KEYVAL, IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE</li> <li>The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be declared like this:</li> <li>SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)</li> <li>INTEGER COMM, ERROR_CODE</li> <li>The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be declared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>	20	SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
23       INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE         24       The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be declared like this:         26       SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)         29       INTEGER COMM, ERROR_CODE         30       The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be declared like this:         31       SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)         34       The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be declared like this:         35       SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)         36       The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be declared like this:         37       SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)         38       SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)         47       The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:         48       SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)         44       INTEGER STATUS(MPI_STATUS_SIZE), IERROR         45       SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)         46       INTEGER STATUS(MPI_STATUS_SIZE), IERROR         47       INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	21	
<ul> <li>The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be declared like this:</li> <li>SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE</li> <li>The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be declared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be declared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>	22	
25The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de- clared like this:27SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE30The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this:33SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE34SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE36The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:38SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE40The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:41SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE		INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
<ul> <li>clared like this:</li> <li>SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)         <ul> <li>INTEGER COMM, ERROR_CODE</li> <li>The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be declared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be declared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul> </li> </ul>		The handler-function argument to MPL COMM_CREATE_ERRHANDLER should be de-
<ul> <li>SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE) INTEGER COMM, ERROR_CODE</li> <li>The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		
<ul> <li>INTEGER COMM, ERROR_CODE</li> <li>The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be declared like this:</li> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be declared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		
The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these: SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	28	•
The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE) INTEGER WIN, ERROR_CODE The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this: SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these: SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	29	INTEGER COMM, ERROR_CODE
<ul> <li>clared like this:</li> <li>subROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>subROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>subROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>	30	The handler function argument to MPL WIN CREATE ERRHANDLER should be de-
<ul> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		8
<ul> <li>SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)</li> <li>INTEGER WIN, ERROR_CODE</li> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		
35       INTEGER WIN, ERROR_CODE         36       The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-         37       clared like this:         39       SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)         40       INTEGER FILE, ERROR_CODE         41       The query, free, and cancel function arguments to MPI_GREQUEST_START should be         42       The query, free, and cancel function arguments to MPI_GREQUEST_START should be         43       declared like these:         44       SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)         46       INTEGER STATUS(MPI_STATUS_SIZE), IERROR         47       INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE		SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
<ul> <li>The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:</li> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		INTEGER WIN, ERROR_CODE
<ul> <li><sup>37</sup> clared like this:</li> <li><sup>38</sup></li> <li><sup>39</sup> SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)</li> <li><sup>40</sup> INTEGER FILE, ERROR_CODE</li> <li><sup>41</sup></li> <li><sup>42</sup> The query, free, and cancel function arguments to MPI_GREQUEST_START should be</li> <li><sup>43</sup> declared like these:</li> <li><sup>44</sup></li> <li><sup>45</sup> SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li><sup>46</sup> INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li><sup>47</sup> INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		The handler function argument to MDI FILE CREATE EPPHANDLER should be de
<ul> <li>SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)         <ul> <li>INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> </ul> </li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)         <ul> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul> </li> </ul>	37	
<ul> <li>INTEGER FILE_EAUMANDLER_FONCTION(FILE, EAUOR_CODE)</li> <li>INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>	38	
<ul> <li>INTEGER FILE, ERROR_CODE</li> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be</li> <li>declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>	39	SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
<ul> <li>The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:</li> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>	40	INTEGER FILE, ERROR_CODE
<ul> <li>declared like these:</li> <li>44</li> <li>45 SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>46 INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>47 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		
<ul> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		
<ul> <li>SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)</li> <li>INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		declared like these:
<ul> <li>46 INTEGER STATUS(MPI_STATUS_SIZE), IERROR</li> <li>47 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</li> </ul>		SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA STATE, STATUS, IERROR)
47 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE		
48	47	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
	48	

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)	1
INTEGER IERROR	2
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	3 4
	4 5
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)	6
INTEGER IERROR	7
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	8
LOGICAL COMPLETE	9
The extent and conversion function arguments to MPI_REGISTER_DATAREP should	10
be declared like these:	11
	12
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)	13
INTEGER DATATYPE, IERROR	14
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE	15
	16
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,	17
POSITION, EXTRA_STATE, IERROR)	18
<type> USERBUF(*), FILEBUF(*)</type>	19
INTEGER COUNT, DATATYPE, IERROR	20
INTEGER(KIND=MPI_OFFSET_KIND) POSITION	21
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	22
	23
A.1.4 Deprecated Prototype Definitions	24
The following are defined C typedefs for deprecated user-defined functions, also included in	25
the file mpi.h.	26
	27
<pre>/* prototypes for user-defined functions */</pre>	28
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,	29
void *extra_state, void *attribute_val_in,	30
<pre>void *attribute_val_out, int *flag);</pre>	31
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,	32
<pre>void *attribute_val, void *extra_state);</pre>	33
	34
The following are deprecated Fortran user-defined callback subroutine prototypes. The	35 36
deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be de-	37
clared like these:	38
	39
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,	40
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR) INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	41
ATTRIBUTE_VAL_OUT, IERR	42
LOGICAL FLAG	43
	44
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)	45
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR	46
	47
	48

#### A.1.5 Info Keys 1

- $\mathbf{2}$ The following info keys are reserved. They are strings. 3
- access\_style 4
- accumulate\_ops  $\mathbf{5}$
- accumulate\_ordering 6
- alloc\_shared\_noncontig  $\overline{7}$
- appnum 8
- arch 9
- cb\_block\_size 10
- cb\_buffer\_size 11
- cb\_nodes 12
- chunked\_item 13
- chunked\_size 14
- chunked 15
- collective\_buffering 16
- file\_perm 17
- filename 18
- file 19
- host 20
- io\_node\_list 21
- ip\_address 22
- ip\_port 23
- nb\_proc 24
- no\_locks 25
- num\_io\_nodes 26
- path 27
- same\_disp\_unit 28
- same\_size 29
- soft 30
- striping\_factor 31
- striping\_unit 32wdir
- 33
- 34
- 35
- A.1.6 Info Values 36
- 37The following info values are reserved. They are strings. 38
- false 39
- random 40
- rar  $^{41}$
- raw 42
- read\_mostly 43
- read\_once 44
- reverse\_sequential 45
- same\_op 46
- same\_op\_no\_op 47
- sequential 48

true	1
war	2
waw	3
write_mostly	4
write_once	5
	6

## A.2 C Bindings

- 3 4

- 11 12

 $1 \\ 2$ 

<i>A.3</i> .	FORTRAN 2008 BINDINGS WITH THE MPI_F08 MODULE	713
A.3	Fortran 2008 Bindings with the mpi_f08 Module	1
		2
		3
		4
		5
		6
		7
		8
		9
		10
		11
		12
		14
		15
		16
		17
		18
		19
		20
		21
		22
		23
		24 25
		25 26
		23
		28
		29
		30
		31
		32
		33
		34
		35
		36
		37 38
		38
		40
		41
		42
		43
		44
		45
		46
		47
		48
	Unofficial Draft for Comment Only	

	714	ANNEX A. LANGUAGE BINDINGS SUMMARY
1	A.4	Fortran Bindings with mpif.h or the mpi Module
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		
42		
43 44		
45		
46		
47		
48		

## Annex B

# Change-Log

Annex B.1 summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown. If not otherwise noted, the section and page references refer to the locations of the change or new functionality in this version of the standard. Changes in Annexes B.2–B.5 were already introduced in the corresponding sections in previous versions of this standard.

B.1	Changes from Version 3.1 to Version 4.0	23 24
		24
B.1.1	Changes from Version 3.1 to Version 4.0	26
1	Section 2.8 on page 20, and MPI-3.1 Section 2.8 on page 20.	27
1.	Added a reference to Chapter 15 about process failures.	28
2	Section 8.1.2 on page 336, and MPI-3.1 Section 8.1.2 on page 335.	29
۷.	Added the MPI_FT predefined attribute.	30
	Added the MFI_I I predemied attribute.	31
3.	Section 8.3 on page 342, and MPI-3.1 Sections 8.3 on page 340. Section 8.4 on page 350,	32
	and MPI-3.1 Section 8.4 on page 347.	33
	Listed the additional error classes for process failure handling.	34
		35 36
4.	Section 8.7 on page 357, and MPI-3.1 Section 8.7 on page 359. Section 10.5.4 on	30
	page 399, and MPI-3.1 Section 10.5.4 on page 398.	38
	Clarified the semantic of MPI_FINALIZE with respect to process failures.	39
5.	Additional Chapter 15 added on page 601.	40
	Added functions and semantics to handle process failures.	41
		42
B.2	Changes from Version 3.0 to Version 3.1	43
D.Z	Changes from version 5.0 to version 5.1	44
B.2.1	Fixes to Errata in Previous Versions of MPI	45
		46
1.	Chapters 3–18, Annex A.3 on page 713, and Example 5.21 on page 189, and MPI-3.0	47
	Chapters 3-17, Annex A.3 on page 707, and Example 5.21 on page 187.	48

	(10	ANNEX B. CHANGE-LOG
1 2 2		Within the mpi_f08 Fortran support method, BIND(C) was removed from all SUBROUTINE, FUNCTION, and ABSTRACT INTERFACE definitions.
3 4 5 6	2.	Section 3.2.5 on page 32, and MPI-3.0 Section 3.2.5 on page 30. The three public fields MPI_SOURCE, MPI_TAG, and MPI_ERROR of the Fortran derived type TYPE(MPI_Status) must be of type INTEGER.
7 8 9 10	3.	Section 3.8.2 on page 69, and MPI-3.0 Section 3.8.2 on page 67. The flag arguments of the Fortran interfaces of MPI_IMPROBE were originally incorrectly defined as INTEGER (instead as LOGICAL).
11 12 13	4.	Section 6.4.2 on page 239, and MPI-3.0 Section 6.4.2 on page 237. In the mpi_f08 binding of MPI_COMM_IDUP, the output argument newcomm is declared as ASYNCHRONOUS.
14 15 16 17	5.	Section 6.4.4 on page 250, 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.
18 19 20 21 22 23	6.	Section 7.6 on page 316, 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).
24 25 26	7.	Section 8.1.1 on page 335, 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.
27 28 29 30 31 32 33	8.	Sections 8.2 (MPI_ALLOC_MEM and MPI_ALLOC_MEM_CPTR), 11.2.2 (MPI_WIN_ALLOCATE and MPI_WIN_ALLOCATE_CPTR), 11.2.3 (MPI_WIN_ALLOCATE_SHARED and MPI_WIN_ALLOCATE_SHARED_CPTR), 11.2.3 (MPI_WIN_SHARED_QUERY and MPI_WIN_SHARED_QUERY_CPTR), 14.2.1 and 14.2.7 (Profiling interface), and corresponding sections in MPI-3.0. The linker name concept was substituted by defining specific procedure names.
34 35 36 37	9.	Section 11.2.1 on page 405, 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.
38 39 40 41	10.	Section 11.3.4 on page 425, 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.
42 43 44 45	11.	Section 11.3.4 on page 425, 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.
46 47 48	12.	Section 14.3 on page 569, 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.

ANNEX B. CHANGE-LOG

13.	Section 14.3.3 on page 571, and MPI-3.0 Section 14.3.3 on page 563. New paragraph clarifying variable name equivalence in the tools information interface.	1 2
14.	Sections 14.3.6, 14.3.7, and 14.3.8 on pages 575, 582, and 594, 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.	3 4 5 6 7 8 9
15.	Section 14.3.7 on page 582, and MPI-3.0 Section 14.3.7 on page 573. Clarify return code of MPI_T_PVAR_{START,STOP,RESET} routines.	10 11 12
16.	Section 14.3.7 on page 582, and MPI-3.0 Section 14.3.7 on page 579, line 7. Clarify the return code when bad handle is passed to an MPI_T_PVAR_* routine.	13 14
17.	Section 18.1.4 on page 631, and MPI-3.0 Section 17.1.4 on page 603. The advice to implementors at the end of the section was rewritten and moved into the following section.	15 16 17 18
18.	Section 18.1.5 on page 632, and MPI-3.0 Section 17.1.5 on page 605. The section was fully rewritten. The linker name concept was substituted by defining specific procedure names.	19 20 21 22
19.	Section 18.1.6 on page 637, and MPI-3.0 Section 17.1.6 on page 611. The requirements on BIND(C) procedure interfaces were removed.	23 24
20.	Annexes A.2, A.3, and A.4 on pages 712, 713, and 714, and MPI-3.0 Annexes A.2, A.3, and A.4 on pages 685, 707, and 756. The predefined callback MPI_CONVERSION_FN_NULL was added to all three annexes.	25 26 27 28 29
21.	Annex ?? on page ??, and MPI-3.0 Annex A.3.4 on page 724. In the mpi_f08 binding of MPI_{COMM TYPE WIN}_{DUP NULL_COPY NULL_DELETE}_FN, all INTENT() information was removed.	30 31 32 33 34
B.2.2	2 Changes in MPI-3.1	35 36
1.	Sections 2.6.4 and 4.1.5 on pages 20 and 103. The use of the intrinsic operators "+" and "-" for absolute addresses is substituted by MPI_AINT_ADD and MPI_AINT_DIFF. In C, they can be implemented as macros.	37 38 39
2.	Sections 8.1.1, 8.7, and 12.4 on pages 335, 357, and 486. The routines MPI_INITIALIZED, MPI_FINALIZED, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN, MPI_GET_VERSION, and MPI_GET_LIBRARY_VERSION are callable from threads without restriction (in the sense of MPI_THREAD_MULTIPLE), irrespective of the actual level of thread support provided, in the case where the im- plementation supports threads.	40 41 42 43 44 45 46
3.	Section 11.2.1 on page 405. The same_disp_unit info key was added for use in RMA window creation routines.	47 48

	Sections 13.4.2 and 13.4.3 on pages 511 and 516. Added MPI_FILE_IREAD_AT_ALL, MPI_FILE_IWRITE_AT_ALL, MPI_FILE_IREAD_ALL, and MPI_FILE_IWRITE_ALL
(	Sections 14.3.6, 14.3.7, and 14.3.8 on pages 575, 582, and 594. Clarified that NULL parameters can be provided in MPI_T_{CVAR PVAR CATEGORY}_GET_INFO routines.
]   ;	Sections 14.3.6, 14.3.7, 14.3.8, and 14.3.9 on pages 575, 582, 594, and 598. New routines MPI_T_CVAR_GET_INDEX, MPI_T_PVAR_GET_INDEX, MPI_T_CATEGORY_GET_INDEX, were added to support retrieving indices of vari- ables and categories. The error codes MPI_T_ERR_INVALID and MPI_T_ERR_INVALID_NAME were added to indicate invalid uses of the interface.
B.3	Changes from Version 2.2 to Version 3.0
B.3.1	Fixes to Errata in Previous Versions of MPI
]	Sections 2.6.2 and 2.6.3 on pages 19 and 19, and MPI-2.2 Section 2.6.2 on page 17, lines 41-42, Section 2.6.3 on page 18, lines 15-16, and Section 2.6.4 on page 18, lines 40-41.
r	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.
	Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, and Annex A.1.1 on pages 27, 178, 542, and 689, and MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, and Annex A.1.1 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 corresponding to the C++ types bool, std::complex <float>, std::complex<double>, and std::complex<long double="">. These datatypes also correspond to the deprecated C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, and MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non- standard C++ types Complex&lt;&gt; were substituted by the standard types</long></double></float>
i	<pre>std::complex&lt;&gt;.</pre>

- This is an MPI-2.2 erratum: MPI\_C\_COMPLEX was added to the "Complex" reduction group.
- 4. Section 7.5.5 on page 304, and MPI-2.2, Section 7.5.5 on page 257, C++ interface on page 264, line 3.
  - This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree are now defined as int& indegree and int& outdegree in the C++ interface of MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT.
- 46 5. Section 13.5.2, Table 13.2 on page 542, and MPI-2.2, Section 13.5.3, Table 13.2 on page 433.

39 40

41

42

43

44

45

This was an MPI-2.2 erratum: The MPI\_C\_BOOL "external32" representation is cor-1 2 rected to a 1-byte size. 3 6. MPI-2.2 Section 16.1.16 on page 471, line 45. 4 This is an MPI-2.2 erratum: The constant MPI::\_LONG\_LONG should be 5 MPI::LONG\_LONG. 6 7. Annex A.1.1 on page 689, Table "Optional datatypes (Fortran)," and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37-41. 9 This is an MPI-2.2 erratum: The C++ datatype handles MPI::INTEGER16, 10 MPI::REAL16, MPI::F\_COMPLEX4, MPI::F\_COMPLEX8, MPI::F\_COMPLEX16, 11 MPI::F\_COMPLEX32 were added to the table. 1213 Changes in MPI-3.0 B.3.2 141. Section 2.6.1 on page 17, Section 17.2 on page 624 and all other chapters. 15The C++ bindings were removed from the standard. See errata in Section B.3.1 on 16page 718 for the latest changes to the MPI C++ binding defined in MPI-2.2. 17This change may affect backward compatibility. 18 19 2. Section 2.6.1 on page 17, Section 16.1 on page 619 and Section 17.1 on page 623. 20The deprecated functions MPI\_TYPE\_HVECTOR, MPI\_TYPE\_HINDEXED, 21MPI\_TYPE\_STRUCT, MPI\_ADDRESS, MPI\_TYPE\_EXTENT, MPI\_TYPE\_LB, 22MPI\_TYPE\_UB, MPI\_ERRHANDLER\_CREATE (and its callback function prototype 23MPI\_Handler\_function), MPI\_ERRHANDLER\_SET, MPI\_ERRHANDLER\_GET, the dep-24recated special datatype handles MPI\_LB, MPI\_UB, and the constants 25MPI\_COMBINER\_HINDEXED\_INTEGER, MPI\_COMBINER\_HVECTOR\_INTEGER, 26MPI\_COMBINER\_STRUCT\_INTEGER were removed from the standard. 27This change may affect backward compatibility. 28 293. Section 2.3 on page 10. 30 Clarified parameter usage for IN parameters. C bindings are now const-correct where 31backward compatibility is preserved. 32 4. Section 2.5.4 on page 15 and Section 7.5.4 on page 298. 33 The recommended C implementation value for MPI\_UNWEIGHTED changed from NULL 34 to non-NULL. An additional weight array constant (MPI\_WEIGHTS\_EMPTY) was in-35troduced. 36 37 5. Section 2.5.4 on page 15 and Section 8.1.1 on page 335. 38 Added the new routine MPI\_GET\_LIBRARY\_VERSION to query library specific ver-39 sions, and the new constant MPI\_MAX\_LIBRARY\_VERSION\_STRING. 40 41 6. Sections 2.5.8, 3.2.2, 3.3, 5.9.2, on pages 17, 27, 29, 178, Sections 4.1, 4.1.7, 4.1.8, 424.1.11, 12.3 on pages 85, 108, 110, 113, 484, and Annex A.1.1 on page 689. New inquiry functions, MPI\_TYPE\_SIZE\_X, MPI\_TYPE\_GET\_EXTENT\_X, 43 44MPI\_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 ele-45ment counts in memory, file views, etc. A new function, 4647MPI\_STATUS\_SET\_ELEMENTS\_X, modifies the opaque part of an MPI\_Status object 48 so that a call to MPI\_GET\_ELEMENTS\_X returns the provided MPI\_Count value (in

1 2		Fortran, INTEGER (KIND=MPI_COUNT_KIND)). The corresponding predefined datatype is MPI_COUNT.
3 4 5 6	7.	Chapter 3 on page 25 until Chapter 18 on page 625. In the C language bindings, the array-arguments' interfaces were modified to consistently use use [] instead of *.
7 8 9		Exceptions are MPI_INIT, which continues to use char <b>***argv</b> (correct because of subtle rules regarding the use of the & operator with char <b>*argv</b> []), and MPI_INIT_THREAD, which is changed to be consistent with MPI_INIT.
10 11 12 13 14 15 16 17	8.	Sections 3.2.5, 4.1.5, 4.1.11, 4.2 on pages 32, 103, 113, 134. The functions MPI_GET_COUNT and MPI_GET_ELEMENTS were defined to set the count argument to MPI_UNDEFINED when that argument would overflow. The functions MPI_PACK_SIZE and MPI_TYPE_SIZE were defined to set the size argument to MPI_UNDEFINED when that argument would overflow. In all other MPI-2.2 routines, the type and semantics of the count arguments remain unchanged, i.e., int or INTEGER.
18 19 20 21	9.	Section 3.2.6 on page 34, and Section 3.8 on page 66. MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE, and MPI_MPROBE.
22 23 24 25	10.	Section 3.8 on page 66 and Section 3.11 on page 82. 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.
26 27 28 29 30 31 32 33 34	11.	Sections 3.8.2, 3.8.3, 18.2.4, A.1.1 on pages 69, 71, 674, 689. 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.
35 36 37 38	12.	Section 4.1.2 on page 87 and Section 4.1.13 on page 118. The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant MPI_COMBINER_HINDEXED_BLOCK were added.
39 40	13.	Chapter 5 on page 143 and Section 5.12 on page 198. Added nonblocking interfaces to all collective operations.
41 42 43 44 45	14.	Sections 6.4.2, 6.4.4, 11.2.7, on pages 239, 250, 417. The new routines MPI_COMM_DUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were added. The routine MPI_COMM_DUP must also duplicate info hints.
46 47 48	15.	Section 6.4.2 on page 239. Added MPI_COMM_IDUP.

16.	Section 6.4.2 on page 239. 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.	1 2 3 4
17.	Section 6.4.2 on page 239. Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type con- stant MPI_COMM_TYPE_SHARED.	5 6 7 8
18.	Section 6.6.2 on page 262. In MPI-2.2, communication involved in an MPI_INTERCOMM_CREATE operation could interfere with point-to-point communication on the parent communicator with the same tag or MPI_ANY_TAG. This interference has been removed in MPI-3.0.	9 10 11 12 13
19.	Section 6.8 on page 283. Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type and window names.	13 14 15 16
20.	Section 7.5.8 on page 314. MPI_CART_MAP can also be used for a zero-dimensional topologies.	17 18 19
21.	Section 7.6 on page 316 and Section 7.7 on page 325. 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_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. The displacement arguments in MPI_NEIGHBOR_ALLTOALLW and MPI_INEIGHBOR_ALLTOALLW were defined as address size integers. In MPI_DIST_GRAPH_NEIGHBORS, an ordering rule was added for communicators created with MPI_DIST_GRAPH_CREATE_ADJACENT.	20 21 22 23 24 25 26 27 28 29 30 31
22.	Section 8.7 on page 357 and Section 12.4.3 on page 489. 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.	31 32 33 34 35
23.	Section 8.7 on page 357. Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.	36 37
24.	Chapter 11 on page 403. Substantial revision of the entire One-sided chapter, with new routines for window creation, additional synchronization methods in passive target communication, new one-sided communication routines, a new memory model, and other changes.	38 39 40 41 42
25.	Section 14.3 on page 569. A new MPI Tool Information Interface was added.	43 44 45
	The following changes are related to the Fortran language support.	46
26.	Section 2.3 on page 10, and Sections 18.1.1, 18.1.2, 18.1.7 on pages 625, 626, and 641. The new mpi_08 Fortran module was introduced.	47 48

1 2 2 3 4 5 6	Handles to opaque objects were defined as nodule. The operators .EQ., .NE., ==, and	2, 18.1.3, 18.1.7 on pages 626, 629, and 641. s named types within the mpi_08 Fortran d /= were overloaded to allow the compari- the overloaded operators are also available
		ns 18.1.1, 18.1.10, 18.1.11, 18.1.12, 18.1.13 ons 18.1.2, 18.1.3, 18.1.7 on pages 626, 629,
10 11 12 13 14 15	Within the mpi_08 Fortran module, choice assumed-rank according to Fortran 2008 TS MPI_SUBARRAYS_SUPPORTED was set to .T can be used in nonblocking MPI operation	buffers were defined as assumed-type and 5 29113 [41], and the compile-time constant <b>CRUE</b> With this, Fortran subscript triplets as; vector subscripts are not supported in does not support this Fortran TR 29113
16 17 18	Section $2.6.2$ on page 19, Section $18.1.2$ on The ierror dummy arguments are OPTIONAL	page 626, and Section 18.1.7 on page 641. L within the mpi_08 Fortran module.
20 21 22 23 24 25 26 27	Section 18.2.5 on page 676. Within the mpi_08 Fortran module, the s Additionally, within both the mpi and the MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG lefined. New conversion routines were as	, MPI_ERROR, and TYPE(MPI_Status) are dded: MPI_STATUS_F2F08, and MPI_Status_f082c, In mpi.h, the new
28 29 30 31		n, the type of the buffer_addr argument of ned and the argument is therefore unused.
32 33 34 35 36 37 38 39	nentation dependent; therefore it is recon- derived types in MPI communication buffer or derived types (in Fortran) is to be used	s within Fortran derived types are imple- mended to use the BIND(C) attribute for ers. If an array of structures (in $C/C++$ ) d in MPI communication buffers, it is rec- e datatype handle and additionally applies
40 3 41 42 43 44 45 46 47 48	185, 191, 277, 283, 344, 346, 347, 619, an gument names were changed because the TYPE and FUNCTION. The new dummy ar	PI_TYPE_SET_ATTR,

	MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, the callback prototype defini- tion MPI_Type_delete_attr_function, and the predefined callback function MPI_TYPE_NULL_DELETE_FN; function was changed in MPI_OP_CREATE, MPI_COMM_CREATE_ERRHANDLER, MPI_WIN_CREATE_ERRHANDLER, MPI_FILE_CREATE_ERRHANDLER, and MPI_ERRHANDLER_CREATE. For consis- tency reasons, INOUBUF was changed to INOUTBUF in MPI_REDUCE_LOCAL, and intracomm to newintracomm in MPI_INTERCOMM_MERGE.	1 2 3 4 5 6 7 8
34.	Section 6.7.2 on page 269. It was clarified that in Fortran, the flag values returned by a comm_copy_attr_fn callback, including MPI_COMM_NULL_COPY_FN and MPI_COMM_DUP_FN, are .FALSE. and .TRUE.; see MPI_COMM_CREATE_KEYVAL.	9 10 11 12
35.	Section 8.2 on page 339. With the mpi and mpi_f08 Fortran modules, MPI_ALLOC_MEM now also supports TYPE(C_PTR) C-pointers instead of only returning an address-sized integer that may be usable together with a non-standard Cray-pointer.	13 14 15 16 17
36.	Section 18.1.15 on page 657, and Section 18.1.7 on page 641. Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers in MPI operations.	18 19 20 21
37.	Section 18.1.16 on page 659 to Section 18.1.19 on page 668, Section 18.1.7 on page 641, and Section 18.1.8 on page 642. The sections about Fortran optimization problems and their solutions were partially rewritten and new methods are added, e.g., the use of the ASYNCHRONOUS attribute. The constant MPI_ASYNC_PROTECTS_NONBLOCKING tells whether the semantics of the ASYNCHRONOUS attribute is extended to protect nonblocking operations. The Fortran routine MPI_F_SYNC_REG is added. MPI-3.0 compliance for an MPI library together with a Fortran compiler is defined in Section 18.1.7.	21 22 23 24 25 26 27 28 29
38.	Section 18.1.2 on page 626. Within the mpi_08 Fortran module, dummy arguments are now declared with INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces.	30 31 32 33
39.	Section 18.1.3 on page 629, and Section 18.1.7 on page 641. The existing mpi Fortran module must implement compile-time argument checking.	34 35
40.	Section 18.1.4 on page 631. The use of the mpif.h Fortran include file is now strongly discouraged.	36 37 38
41.	Section A.1.1, Table " <i>Predefined functions</i> " on page 697, Section A.1.3 on page 704, and Section ?? on page ??. 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.	39 40 41 42 43 44
42.	Section A.1.3 on page 704. In some routines, the Fortran callback prototype names were changed from $\ldots$ _FN to $\ldots$ _FUNCTION to be consistent with the other language bindings.	45 46 47 48

1	B.4	Changes from Version 2.1 to Version 2.2
2 3 4 5 6	1.	Section 2.5.4 on page 15. 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.
7 8 9 10	2.	Section 2.6 on page 17, and Section 17.2 on page 624. The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.
11 12 13 14 15 16 17	3.	Section 3.2.2 on page 27. 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.
18 19 20 21	4.	Section 3.2.2 on page 27. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are now valid predefined MPI datatypes.
22 23 24 25 26 27	5.	Section 3.4 on page 39, Section 3.7.2 on page 50, Section 3.9 on page 75, and Section 5.1 on page 143. The read access restriction on the send buffer for blocking, non blocking and collective API has been lifted. It is permitted to access for read the send buffer while the operation is in progress.
28 29	6.	Section 3.7 on page 49. The Advice to users for IBSEND and IRSEND was slightly changed.
30 31 32 33	7.	Section 3.7.3 on page 54. The advice to free an active request was removed in the Advice to users for MPI_REQUEST_FREE.
34 35 36	8.	Section 3.7.6 on page 65. MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.
37 38 39 40	9.	Section 5.8 on page 170. "In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and MPI_ALLTOALLW for intracommunicators.
40 41 42 43 44	10.	<ul> <li>Section 5.9.2 on page 178.</li> <li>Predefined parameterized datatypes (e.g., returned by</li> <li>MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g.</li> <li>MPI_REAL8) have been added to the list of valid datatypes in reduction operations.</li> </ul>
45 46 47 48	11.	Section 5.9.2 on page 178. $MPI_(U)INT\{8,16,32,64\}_T$ are all considered C integer types for the purposes of the predefined reduction operators. $MPI_AINT$ and $MPI_OFFSET$ are considered Fortran

1 integer types. MPI\_C\_BOOL is considered a Logical type.  $\mathbf{2}$ MPI\_C\_COMPLEX, MPI\_C\_FLOAT\_COMPLEX, MPI\_C\_DOUBLE\_COMPLEX, and 3 MPI\_C\_LONG\_DOUBLE\_COMPLEX are considered Complex types. 4 12. Section 5.9.7 on page 191. 5The local routines MPI\_REDUCE\_LOCAL and MPI\_OP\_COMMUTATIVE have been 6 added. 7 8 13. Section 5.10.1 on page 192. 9 The collective function MPI\_REDUCE\_SCATTER\_BLOCK is added to the MPI stan-10 dard. 11 14. Section 5.11.2 on page 196. 12Added in place argument to MPI\_EXSCAN. 13 1415. Section 6.4.2 on page 239, and Section 6.6 on page 259. 15Implementations that did not implement MPI\_COMM\_CREATE on intercommuni-16cators will need to add that functionality. As the standard described the behav-17 ior of this operation on intercommunicators, it is believed that most implementa-18tions already provide this functionality. Note also that the C++ binding for both 19 MPI\_COMM\_CREATE and MPI\_COMM\_SPLIT explicitly allow Intercomms. 2016. Section 6.4.2 on page 239. 21MPI\_COMM\_CREATE is extended to allow several disjoint subgroups as input if comm 22 is an intracommunicator. If comm is an intercommunicator it was clarified that all 23processes in the same local group of comm must specify the same value for group. 242517. Section 7.5.4 on page 298. 26New functions for a scalable distributed graph topology interface has been added. 27In this section, the functions MPI\_DIST\_GRAPH\_CREATE\_ADJACENT and 28 MPI\_DIST\_GRAPH\_CREATE, the constants MPI\_UNWEIGHTED, and the derived C++ 29 class Distgraphcomm were added. 30 18. Section 7.5.5 on page 304. 31For the scalable distributed graph topology interface, the functions 32 MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT and MPI\_DIST\_GRAPH\_NEIGHBORS and 33 the constant MPI\_DIST\_GRAPH were added. 34 35 19. Section 7.5.5 on page 304. 36 Remove ambiguity regarding duplicated neighbors with MPI\_GRAPH\_NEIGHBORS 37 and MPI\_GRAPH\_NEIGHBORS\_COUNT. 38 20. Section 8.1.1 on page 335. 39 The subversion number changed from 1 to 2. 40 41 21. Section 8.3 on page 342, Section 16.2 on page 622, and Annex A.1.3 on page 704. 42Changed function pointer typedef names MPI\_{Comm,File,Win}\_errhandler\_fn to 43 MPI\_{Comm,File,Win}\_errhandler\_function. Deprecated old "\_fn" names. 4422. Section 8.7.1 on page 363. 4546Attribute deletion callbacks on MPI\_COMM\_SELF are now called in LIFO order. Imple-47mentors must now also register all implementation-internal attribute deletion callbacks 48 on MPI\_COMM\_SELF before returning from MPI\_INIT/MPI\_INIT\_THREAD.

### Unofficial Draft for Comment Only

1 2	23.	Section 11.3.4 on page 425. The restriction added in MPI 2.1 that the operation MPI_REPLACE in
3		MPI_ACCUMULATE can be used only with predefined datatypes has been removed.
4		MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0.
5 6		Also, a clarification has been made that MPI_REPLACE can be used only in
7		MPI_ACCUMULATE, not in collective operations that do reductions, such as
8		MPI_REDUCE and others.
9	24.	Section 12.2 on page 477.
10		Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for
11		MPI::Grequest::Start() for consistency with the rest of MPI functions that take function
12		pointer arguments.
13	25	Section $13.5.2$ on page 540, and Table 13.2 on page 542.
14	20.	MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX,
15		MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX,
16		MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes
17		in the external32 representation.
18	0.0	-
19 20	26.	Section 18.2.7 on page 681. The description was modified that it only describes how an MPI implementation be-
20		haves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example
22		16.17 was replaced with three new examples 18.13, 18.14, and 18.15 on pages 682-683
23		explicitly detailing cross-language attribute behavior. Implementations that matched
24		the behavior of the old example will need to be updated.
25	~ -	· ·
26	27.	Annex A.1.1 on page 689.
27		Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 702).
28	28.	Annex A.1.1 on page 689. Table Named Predefined Datatypes.
29		Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL,
30 31		$MPI\_C\_FLOAT\_COMPLEX, \ MPI\_C\_COMPLEX, \ MPI\_C\_DOUBLE\_COMPLEX, \ \mathrm{and}$
32		MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes.
33		
34	B.5	Changes from Version 2.0 to Version 2.1
35	-	
36	1.	Section 3.2.2 on page 27, and Annex A.1 on page 689.
37		In addition, the MPI_LONG_LONG should be added as an optional type; it is a syn-
38		onym for MPI_LONG_LONG_INT.
39	2.	Section 3.2.2 on page 27, and Annex A.1 on page 689.
40 41		MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym),
42		MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved
43		from optional to official and they are therefore defined for all three language bindings.
44	3.	Section 3.2.5 on page 32.
45		MPI_GET_COUNT with zero-length datatypes: The value returned as the
46		count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes
47		have been transferred is zero. If the number of bytes transferred is greater than zero,
48		MPI_UNDEFINED is returned.

### Unofficial Draft for Comment Only

4.	Section 4.1 on page 85. General rule about derived datatypes: Most datatype constructors have replication	1 $2$	
	count or block length arguments. Allowed values are non-negative integers. If the	3	
	value is zero, no elements are generated in the type map and there is no effect on	4	
	datatype bounds or extent.	5	
		6	
5.	Section $4.3$ on page $140$ .	7	
	MPI_BYTE should be used to send and receive data that is packed using	8	
	MPI_PACK_EXTERNAL.	9	
0		10	
6.	Section 5.9.6 on page 189.	11	
	If comm is an intercommunicator in MPI_ALLREDUCE, then both groups should provide count and datatype arguments that specify the same type signature (i.e., it is not		
	necessary that both groups provide the same <b>count</b> value).	14	
7	Section $6.3.1$ on page $230$ .	15	
1.	MPI_GROUP_TRANSLATE_RANKS and MPI_PROC_NULL: MPI_PROC_NULL is a valid	16	
	rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL	17	
	as the translated rank.	18	
	as the translated fairk.	19	
8.	Section 6.7 on page 267.	20	
	About the attribute caching functions:	21	
		22	
	Advice to implementors. High-quality implementations should raise an er-	23	
	ror when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL	24	
	is used with an object of the wrong type with a call to	25	
	MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or	26	
	MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each key-	27	
	val, information on the type of the associated user function. (End of advice to	28	
	implementors.)	29	
0	Castion 6.9 on name 202	30	
9.	Section 6.8 on page 283.	31	
	In MPI_COMM_GET_NAME: In C, a null character is additionally stored at		
	name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For-	32 33	
	tran, name is padded on the right with blank characters. resultlen cannot be larger	34	
	then MPI_MAX_OBJECT_NAME.	34	
10.	Section 7.4 on page $292$ .	36	
	About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must	30 37	
	have identical values on all processes of the group of comm_old.	38	
		39	
11.	Section $7.5.1$ on page 294.	39 40	
	In MPI_CART_CREATE: If ndims is zero then a zero-dimensional Cartesian topology	40 41	
	is created. The call is erroneous if it specifies a grid that is larger than the group size		
	or if <b>ndims</b> is negative.	42	
10		43 44	
12.	Section 7.5.3 on page 296.		
	In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$ , then		
	MPI_COMM_NULL is returned in all processes.	46	
		47	
		48	

1	13.	Section $7.5.3$ on page 296.
2		In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times
3		in the list of neighbors of a process (i.e., there may be multiple edges between two
4		processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the
5		graph). The adjacency matrix is allowed to be non-symmetric.
6		graph). The adjacency matrix is anowed to be non-symmetric.
		Advice to users. Performance implications of using multiple edges or a non-
7		symmetric adjacency matrix are not defined. The definition of a node-neighbor
8		
9		edge does not imply a direction of the communication. (End of advice to users.)
10	14	Section $7.5.5$ on page $304$ .
11	11.	In MPI_CARTDIM_GET and MPI_CART_GET: If comm is associated with a zero-
12		
13		dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and
14		MPI_CART_GET will keep all output arguments unchanged.
15	15	Section 7.5.5 on page 204
16	10.	Section 7.5.5 on page 304.
		In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol-
17		ogy, coord is not significant and 0 is returned in rank.
18	16	Castion 7.5.5 on name 204
19	10.	Section 7.5.5 on page 304.
20		In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian
21		topology, <b>coords</b> will be unchanged.
22	17	Section 7.5.6 on page 212
23	11.	Section 7.5.6 on page 312.
24		In MPI_CART_SHIFT: It is erroneous to call MPI_CART_SHIFT with a direction that
25		is either negative or greater than or equal to the number of dimensions in the Cartesian
26		communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a
27		comm that is associated with a zero-dimensional Cartesian topology.
28	10	
	18.	Section 7.5.7 on page 313.
29		In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associ-
30		ated with a zero-dimensional Cartesian topology then newcomm is associated with a
31		zero-dimensional Cartesian topology.
32	101	
33	18.1.	Section 8.1.1 on page 335.
34		The subversion number changed from 0 to 1.
35	10	Casting 0.1.0 and as 226
36	19.	Section 8.1.2 on page 336.
37		In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at
38		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
39		then MPI_MAX_PROCESSOR_NAME.
40		
41	20.	Section $8.3$ on page $342$ .
42		MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object
43		is created. That is, once the error handler is no longer needed,
44		MPI_ERRHANDLER_FREE should be called with the error handler returned from
45		MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark
46		the error handler for deallocation. This provides behavior similar to that of
47		MPI_COMM_GROUP and MPI_GROUP_FREE.
48		
-		

#### B.5. CHANGES FROM VERSION 2.0 TO VERSION 2.1

- 21. Section 8.7 on page 357, see explanations to MPI\_FINALIZE. MPI\_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI\_COMM\_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4 on page 399.
- 22. Section 8.7 on page 357. About MPI\_ABORT:

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

23. Section 9 on page 367.

An implementation must support info objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI\_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI\_INFO\_GET\_NKEYS, MPI\_INFO\_GET\_NTHKEY, MPI\_INFO\_GET\_VALUELEN, and MPI\_INFO\_GET must retain all (key,value) pairs so that layered functionality can also use the Info object.

- 24. Section 11.3 on page 419. MPI\_PROC\_NULL is a valid target rank in the MPI RMA calls MPI\_ACCUMULATE, MPI\_GET, and MPI\_PUT. The effect is the same as for MPI\_PROC\_NULL in MPI pointto-point communication. See also item 25 in this list.
- 25. Section 11.3 on page 419. After any RMA operation with rank MPI\_PROC\_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch. See also item 24 in this list.
- 26. Section 11.3.4 on page 425. MPI\_REPLACE in MPI\_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.
- 27. Section 13.2.8 on page 502. About MPI\_FILE\_SET\_VIEW and MPI\_FILE\_SET\_INFO: When an info object that specifies a subset of valid hints is passed to MPI\_FILE\_SET\_VIEW or MPI\_FILE\_SET\_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.
- 28. Section 13.2.8 on page 502. 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.

**Unofficial Draft for Comment Only** 

 $\mathbf{2}$ 

 $^{24}$ 

1 2 3	29.	Section 13.3 on page 505. If a file does not have the mode MPI_MODE_SEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.
4 5 6	30.	Section 13.5.2 on page 540. The bias of 16 byte doubles was defined with 10383. The correct value is 16383.
7 8 9	31.	MPI-2.2, Section 16.1.4 (Section was removed in MPI-3.0). In the example in this section, the buffer should be declared as const void* buf.
10 11	32.	Section 18.1.9 on page 643. About MPI_TYPE_CREATE_F90_XXX:
12 13 14 15 16 17 18 19 20 21 22		Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, the MPI implementation should return the same datatype handle for the same ( REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination ( p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash- table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (End of advice to implementors.)
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	33.	Section A.1.1 on page 689. MPI_BOTTOM is defined as void * const MPI::BOTTOM.
47 48		

# Bibliography

- V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2
- [2] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2
- [3] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994. 6.1
- [4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993.
   1.2
- [5] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990. 1.2
- [6] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. IJHPCN, 1(1/2/3):91–99, 2004. 11.7
- [7] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993. 13.1
- [8] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992. 1.2
- [9] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. 1.2
- [10] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, April 1994. 1.2
- [11] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormholerouted networks for interprocessor communication in multicomputers. In *Proceedings* of the 1990 Supercomputing Conference, pages 647–656, 1990. 7.1

9 10

11

12

13 14

15

16

17 18

19

20

21

22

23 24

25

26

27 28

29

30

 $^{31}$ 

32

33 34

35

36 37

38

39

40

41

42

43

 $\frac{44}{45}$ 

46

47

1 2 3	[12]	S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communica- tion performance of large multicomputers. In <i>Proceedings of the 1991 International</i> <i>Conference on Parallel Processing, vol. II (Software)</i> , pages II–1 – II–4, 1991. 7.1
4 5 6	[13]	Parasoft Corporation. Express version 1.0: A communication environment for parallel computers, 1988. $1.2$
7 8 9 10 11	[14]	Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors. Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings, volume 6960 of Lecture Notes in Computer Science. Springer, 2011. 16, 36
12 13 14 15	[15]	Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In <i>IPPS '93 Workshop on Input/Output in Parallel Computer Systems</i> , pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38. 13.1
16 17 18 19	[16]	James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective communicator creation in MPI. In Cotronis et al. [14], pages 282–291. 6.4.2
20 21 22	[17]	J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. <i>Computers in Physics</i> , 7(2):166–75, April 1993. 1.2
23 24 25 26	[18]	J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user- level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993. 1.2
27 28 29	[19]	Edinburgh Parallel Computing Centre, University of Edinburgh. <i>CHIMP Concepts</i> , June 1991. 1.2
30 31	[20]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992. 1.2
32 33 34 35	[21]	D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991. 6.1.2
36 37 38 39	[22]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard. The In- ternational Journal of Supercomputer Applications and High Performance Computing, 8, 1994. 1.3
40 41	[23]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org. 1.3
42 43 44 45	[24]	Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. <i>PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing.</i> MIT Press, 1994. 10.1
46 47 48	[25]	G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable in- strumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990. 1.2

### Unofficial Draft for Comment Only

- [26] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009. 3.8.2
- [27] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. 1.2
- [28] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Available via world wide web from http://www.uni-karlsruhe.de/~Michael.Hennecke/Publications/#MPI\_F90. 18.1.3
- [29] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. 3.8.1, 3.8.2
- [30] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 5.12
- [31] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum mechanical calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 15th European PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. 7.6
- [32] T. Hoefler and A. Lumsdaine. Message progression in parallel computing to thread or not to thread? In Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society, Oct. 2008. 5.12
- [33] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07.* IEEE Computer Society/ACM, Nov. 2007. 5.12
- [34] T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimization for medical image reconstruction algorithms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 5.12
- [35] T. Hoefler and J. L. Traeff. Sparse collective operations for MPI. In Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop, May 2009. 7.6
- [36] Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI common practice, issues, and extensions. In Cotronis et al. [14], pages 345–355. 6.4.2
- [37] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. 13.5.2
- [38] International Organization for Standardization, Geneva, ISO 8859-1:1987. Information 46 processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet 47 No. 1, 1987. 13.5.2

#### Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 

3

4

5 6

 $\overline{7}$ 

8

9

10

11

12

13

14

15 16

17

18

19

20

21

22

23 24

25

26

27 28

29

30

 $^{31}$ 

32

33

34

35

36 37

38

39

40

41

42 43

44

1 2 3	[39]	International Organization for Standardization, Geneva, ISO/IEC 9945-1:1996(E). In- formation technology — Portable Operating System Interface (POSIX) — Part 1: Sys- tem Application Program Interface (API) [C Language], December 1996. 12.4, 13.2.1
4 5 6 7	[40]	International Organization for Standardization, Geneva, ISO/IEC 1539-1:2010. In- formation technology – Programming languages – Fortran – Part 1: Base language, November 2010. 18.1.1, 18.1.2
8 9 10 11 12 13	[41]	International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TS 29113. TS on further interoperability with C, 2012. http://www.nag.co.uk/sc22wg5/, successfully balloted DTS at ftp://ftp.nag.co.uk/sc22wg5/N1901-N1950/N1917.pdf. 18.1.1, 18.1.1, 18.1.2, 18.1.7, 28
14 15	[42]	Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. <i>The High Performance Fortran Handbook</i> . MIT Press, 1993. 4.1.4
16 17 18 19	[43]	David Kotz. Disk-directed I/O for MIMD multiprocessors. In <i>Proceedings of the 1994 Symposium on Operating Systems Design and Implementation</i> , pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 13.1
20 21 22	[44]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. 7.1
23 24 25 26 27	[45]	S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Also available at http://www.netbsd.org/Documentation/lite2/psd/. 10.5.5
28 29	[46]	nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2
30 31 32	[47]	Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 13.1
33 34	[48]	William J. Nitzberg. <i>Collective Parallel I/O</i> . PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 13.1
35 36 37	[49]	4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. 10.5.5
38 39 40 41	[50]	Paul Pierce. The NX/2 operating system. In <i>Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications</i> , pages 384–390. ACM Press, 1988. 1.2
42 43 44	[51]	Martin Schulz and Bronis R. de Supinski. $P^N$ MPI tools: A whole lot greater than the sum of their parts. In ACM/IEEE Supercomputing Conference (SC), pages 1–10. ACM, 2007. 14.2.8
45 46 47 48	[52]	K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In <i>Proceedings of Supercomputing '95</i> , December 1995. 13.1

- [53] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990. 1.2, 6.1.2
- [54] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992. 1.2
- [55] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993. 6.1
- [56] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University — Dept. of Computer Science, April 1994. http://www.erc.msstate.edu/mpi/mpix.html. 5.2.2
- [57] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994. 6.1.2, 6.5.6
- [58] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996. 13.1
- [59] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. 13.5.2
- [60] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2

 $^{24}$ 

General Index

10

11

This index lists mainly terms of the MPI specification. The underlined page numbers refer to the definitions or parts of the definition of the terms. Bold face numbers mark section titles.

12absolute addresses, 16, 103, 661 access epoch, 439 13 action - in function names, 1014active, <u>54</u>, 288 15active target communication, 43816addresses, 117 17absolute, 16, 103, 661 18 correct use, 117 19relative displacement, 16, 103 all-reduce, 189 20nonblocking, 212 21all-to-all. 170 22 nonblocking, 208 23array arguments, 14  $^{24}$ assertions, 452 25ASYNCHRONOUS - Fortran attribute, 663 26attribute, 227, 267, 681 27caching, 226 28barrier synchronization, 149 29 nonblocking, 200 30 blocking, 11, 39, 42, 509  $^{31}$ I/O, **510** 32 bounds of datatypes, 108 33 broadcast, 150 34 nonblocking, 201 buffer allocation, **46** 35 buffered, 39, 49, 50 36 nonblocking, 49 37 buffered send, 4238 39 C – language binding, 19 caching, 225, 226, 267 40callback functions 41 language interoperability, 680 42prototype definitions, 704 43 deprecated, 709 44cancel, 73 45canonical pack and unpack, 140 46Cartesian - topology, 292, 294 47change-log, 715 choice, 16 48

class - in function names, 10clock synchronization, 338 collective, 11, 509 collective communication, 143 correctness, **216** file data access operations, 527 neighborhood, 316 nonblocking, 198 commit, 111COMMON blocks, 666 communication, 403collective. 143 modes, 39 one-sided, 403 point-to-point, 25 RMA, 403 communicator, 29, 225, 226 completes - operation, 11completion, 54 multiple, 59 connected, 399 constants, 15, 685, 689 context, 225, 226, 228 control variables – tools interface, 575 conversion, 37 counts, 17 create - in function names, 10 data, 27 data conversion, 37 datatypes, 85, 679 delete - in function names, <u>10</u>

deprecated functions, 619

disconnected, 399

empty, 54

displacement, 493, 506

derived datatype, <u>12</u>, **85**, **657** 

elementary datatype, <u>493</u>, 507

deprecated names and functions, 17

distributed graph – topology, 292, 298

dynamically attached memory, 412

end of file, 495envelope, <u>25</u>, **29** environmental inquiries, 336 equivalent datatypes,  $\underline{12}$ error handling, 20, 342 error codes and classes, 350 error handlers, 344, 350, 680 I/O, 557 one-sided communication, 454 resource error, 21 establishing communication, 387 etype, <u>493</u>, 507 exception, 342exclusive scan, 196 nonblocking, 216 explicit offsets, 509, 511 exposure epoch, 439extent of datatypes, <u>86</u>, <u>107</u>, **108** true extent, 110 external32 – file data representation, <u>538</u> extra-state, 685 fairness, 44, <u>64</u> fault tolerance, 343, **601** ack, 609, 609 agree, <u>610</u>, 611 communicator, 603, 607 dynamic process, 604 error classes, 613 finalize, 361, 400, 602 I/O, **606**, **612** inquiry, 338, 602 mitigation, 607 notification, **602**, 613 one-sided, 605, 611 process failure, 20, 400, 601 revoke, <u>607</u>, 611, 612 shrink, 608startup, 602 file, <u>493</u> data access, 508 collective operations, 527 explicit offsets, 511 individual file pointers, 516 seek, **528** shared file pointers, 524 split collective, 529

end of file, 495

interoperability, **536** manipulation, **495** offset, **16**,  $\underline{494}$  pointer,  $\underline{495}$ 

filetype,  $\underline{494}$  handle,  $\underline{495}$ 

size, $\underline{495}$	1
view, 493, <u>494</u> , <b>505</b>	2
file size, <b>552</b>	3
finished, <b>363</b>	4
Fortran – language binding, <b>19</b> , <b>625</b>	5
Fortran support, <b>625</b>	6
gather, <b>151</b>	7
nonblocking, <b>202</b>	8
gather-to-all, 167	9
nonblocking, <b>206</b>	10
general datatype, $\underline{85}$	11
generalized requests, $477$ , $477$	12
get – in function names, $\underline{10}$	13
graph – topology, <u>292</u> , <b>296</b>	
group, <b>225</b> , <u>226</u> , <u>228</u> , 260	14
group objects, $\underline{228}$	15
	16
handles, $12, 674$	17
host rank, <b>337</b>	18
immediate, $50$	19
inactive, <u>54</u>	20
inclusive scan, <b>195</b>	21
nonblocking, <b>215</b>	
independent, <u>399</u>	22
individual file pointers, <u>509</u> , <b>516</b>	23
info object, $367$	24
file info, <b>502</b>	25
keys, <b>710</b>	26
values, <b>710</b>	27
initiation, <b>50</b>	28
inter-communication, $\underline{227}$ , $\underline{259}$	29
inter-communicator, 146, <u>227</u> , <u>259</u>	30
collective operations, 147, 148	
interlanguage communication, 686	31
internal – file data representation, $538$	32
interoperability, <b>536</b>	33
intra-communication, <u>227</u> , <u>259</u>	34
intra-communicator, $1\overline{46}$ , $2\overline{26}$ , $259$	35
collective operations, 146	36
intra-communicator objects, $\underline{229}$	37
I/O, <b>493</b>	38
IO rank, <b>337</b>	39
is – in function names, $\underline{10}$	
	40
language binding, 17, 625	41
interoperability, 673	42
summary, 689	43
lb_marker, 97, 98, 101, <u>107</u> , 107, 111	44
erased, $\underline{110}$	45
local, $\underline{11}$ , 39	46
local group, 240	40
loosely synchronous model, 287	
lower bound, <u>107</u>	48

738
-----

```
1
      lower-bound markers, 106
\mathbf{2}
      macros, 20
3
      main thread, 487
4
      matched receives, 71
5
      matching
6
           type, 35, 114, 551
7
      matching probe, 69
8
      memory
           allocation, 339
9
           system, 12
10
      memory model, 404, 437
11
           separate, 404, 410
12
           unified, 404, 410
13
      message, 25
14
           data, 27
15
           envelope, 29
16
      modes, 39
      module variables, 666
17
      mpi module – Fortran support, 629
18
      mpi_f08 module - Fortran support, 626
19
      mpiexec, 358, 363, 365
20
      mpif.h include file – Fortran support, 631
21
      mpirun, 364
22
      multiple completions, 59
23
      named datatype, 12
24
      names, 387
25
           name publishing, 392
26
      naming objects, 283
27
      native – file data representation, 537
28
      neighborhood collective communication, 316
29
           nonblocking, 326
      non-local, <u>11</u>, <u>39</u>, <u>40</u>
30
      nonblocking, 11, 49, 419, 509
^{31}
           communication, \underline{49}
32
           completion, 54
33
           Fortran problems, 660
34
           I/O, 510
35
           initiation, 50
36
           request objects, 50
      null handle, 54
37
      null processes, 82
38
39
      offset, 16, 494
40
      one-sided communication, 403
41
           Fortran problems, 661
42
      opaque objects, 12, 678
43
      operation completes, 11
      origin, 404
44
45
      pack, 134
46
           canonical, 140
47
      packing unit, 136
48
      parallel procedure, 288
```

passive target communication, 438performance variables – tools interface, 582 persistent communication requests, 75 Fortran problems, 661 PMPI\_, 563 point-to-point communication, 25 portable datatype, 12 ports, **387** predefined datatype, 12 predefined reduction operations, 178 private window copy, 437probe, 66 probe, matching, 69 process creation, 373 process failure fault tolerance, 601 process group, 29 processes, 20 processor name, 338 profiling interface, 563 prototype definitions, 704 deprecated, 709 public window copy, 437rank, 228 ready, <u>40</u>, 49, <u>50</u> nonblocking, 49 ready send, 42receive, 25, 26, 30 buffer, 26 complete,  $\underline{49}$ context, 260start call, 49 reduce, **176** nonblocking, 211 reduce-scatter, 192 nonblocking, 213, 214 reduction operations, 175, 681 predefined, 178 process-local, 191 scan, 195 user-defined. 185 related, 136 relative displacement, 16, 103 remote group, 240 Remote Memory Access, see RMA removed interfaces, 623 removed names and functions, 17 request complete I/O, 510 request objects, 50 resource error,  $\underline{21}$ RMA, 403 communication calls, 419 request-based, 432

memory model, 437 synchronization calls, 438 scan, 195 exclusive, 196 inclusive, **195** scatter, 161 nonblocking, 204 seek, 528 semantics file consistency, 546 nonblocking communications, 58 point-to-point communication, 42 semantics and correctness one-sided communication, 455 send, <u>25</u>, **26** buffer, 25complete, 49context, 260start,  $\underline{49}$ send-receive, 80 separate memory model, 404, 410, 437 sequential storage,  $\underline{117}$ Set - in function names, 10 shared file pointers, <u>509</u>, **524** shared memory allocation, 409 signals, 22 singleton init, 398 size changing I/O, 552 source, 260split collective, 509, 529 standard, <u>39</u>, <u>49</u> nonblocking, 49standard send, 42starting processes, 374, 376 startup, **357** portable, 364 state, 14 status, 32, 676 associating information, 484 ignore, 34 test, 65 strong synchronization, 441synchronization, 403, 419 synchronization calls – RMA, 438 synchronous, <u>39</u>, 49, <u>50</u>, <u>54</u> nonblocking, 49 synchronous send, 42system memory, <u>12</u> tag values, 337 target, 404thread compliant,  $\underline{486}$ ,  $\underline{490}$ 

threads, <b>486</b>	1
timers and synchronization, <b>356</b>	2
tool information interface, <b>569</b>	3
tool support, <b>563</b>	4
topologies, <b>291</b>	5
topology	6
Cartesian, <u>292</u> , <b>294</b>	
distributed graph, $\underline{292}$ , $298$	7
graph, <u>292</u> , <b>296</b>	8
virtual, <b>292</b>	9
true extent of datatypes, <b>110</b>	10
type map, $\frac{86}{10}$	11
type matching, $35$ , $114$	12
type signature, $\underline{86}$	13
types, <b>702</b>	14
ub_marker, 97, 98, 101, 102, <u>107</u> , 107, 111	15
erased, $\underline{110}$	16
unified memory model, $404$ , $410$ , $437$	17
universe size, <b>397</b>	18
unnamed datatype, $\underline{12}$	19
unpack, 134	
canonical, 140	20
upper bound, $\underline{107}$	21
upper-bound markers, <b>106</b>	22
user functions at process termination, $363$	23
user-defined data representations, $541$	24
user-defined reduction operations, 185	25
	26
verbosity levels – tools interface, 570	27
version inquiries, <b>335</b>	28
view, 493, <u>494</u> , <b>505</b> virtual topology, 226, 227, <b>292</b>	29
Virtual topology, $\underline{220}, \underline{221}, \underline{292}$	30
weak synchronization, $441$	
window	31
allocation, <b>407</b>	32
creation, $405$	33
dynamically attached memory, 412	34
shared memory allocation, <b>409</b>	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

## **Examples Index**

10

11 12 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.

13	ASYNCHRONOUS, 471, 669, 671
14	Attributes between languages, $682$
15	Basic tool using performance variables in the
16	MPI tool information interface, 592
17	
18	C/Fortran handle conversion, $675$
19	Cartesian virtual topologies, 331
20	Client-server code, 64, 65
21	with blocking probe, 68
22	with blocking probe, wrong, 68
23	Comm_failure_allget example, 614 Comm_failure_allget2 example, 614
24	Comm_nanure_anget2 example, 014
24	Datatype
	3D array, 125
26	absolute addresses, $130$
27	array of structures, $127$
28	elaborate example, 137, 139
29	matching type, 114
30	matrix transpose, 126
31	union, 131
32	Datatypes matching 25
33	matching, 35 not matching, 36
34	untyped, 36
35	Deadlock
36	if not buffered, 45
37	with MPI_Bcast, 216, 217
38	wrong message exchange, 45
39	
40	False matching of collective operations, 220 Fault-tolerant iterative refinement with shrink
41	and agreement, 616
42	Fortran 90 copying and sequence problem, 653
43	655, 656
44	Fortran 90 derived types, 657
45	Fortran 90 heterogeneous communication, 650
45	651
	Fortran 90 invalid KIND, 646
47	Fortran 90 MPI_TYPE_MATCH_SIZE
48	implementation, 649

Fortran 90 overlapping communication and computation, 669, 670, 672 Fortran 90 register optimization, 660, 662 Independence of nonblocking operations, 223 Intercommunicator, 243, 247 Interlanguage communication, 686 Intertwined matching pairs, 43 Master example, 615Message exchange, 44 Mixing blocking and nonblocking collective operations, 219 Mixing collective and point-to-point requests, 222MPI\_ACCUMULATE, 427 MPI\_Accumulate, 469, 472 MPI\_Aint, 127 MPI\_Aint\_add, 472 MPI\_Allgather, 169 MPI\_ALLOC\_MEM, 341 MPI\_Alloc\_mem, 342, 472 MPI\_ALLREDUCE, 190  $\mathrm{MPI\_Alltoall},\, \underline{221}$ MPI\_ASYNC\_PROTECTS\_NONBLOCKING, 471 MPI\_Barrier, 361, 459-462, 468-470 MPI\_Bcast, 151, 216-220 MPI\_BSEND, 43 MPI\_Buffer\_attach, 47, 360 MPI\_Buffer\_detach, 47 MPI\_BYTE, 36 MPI\_Cancel, 361 MPI\_CART\_COORDS, 313 MPI\_CART\_GET, 331 MPI\_CART\_RANK, 313 MPI\_CART\_SHIFT, 313, 331 MPI\_CART\_SUB, 314 MPI\_CHARACTER, 37 MPI\_COMM\_AGREE, 613, 614, 616 MPI\_Comm\_create, 243, 253, 254, 257

MPI\_Comm\_create\_keyval, 281 MPI\_Comm\_dup, 256 MPI\_COMM\_FAILURE\_ACK, 614, 615 MPI\_COMM\_FAILURE\_GET\_ACKED, 614, 615 MPI\_COMM\_FREE, 613, 614, 616 MPI\_Comm\_get\_attr, 281 MPI\_COMM\_GROUP, 614 MPI\_Comm\_group, 243, 257, 281 MPI\_Comm\_remote\_size, 247 MPI\_COMM\_REVOKE, 616 MPI\_Comm\_set\_attr, 281 MPI\_COMM\_SHRINK, 614, 616 MPI\_COMM\_SPAWN, 378 MPI\_Comm\_spawn, 378 MPI\_COMM\_SPAWN\_MULTIPLE, 384 MPI\_Comm\_spawn\_multiple, 384 MPI\_COMM\_SPLIT, 613 MPI\_Comm\_split, 247, 265, 266 MPI\_Compare\_and\_swap, 470, 472 MPI\_DIMS\_CREATE, 295, 331 MPI\_DIST\_GRAPH\_CREATE, 302 MPI\_Dist\_graph\_create, 303 MPI\_DIST\_GRAPH\_CREATE\_ADJACENT, 302 MPI\_F\_sync\_reg, 471 MPI\_FILE\_CLOSE, 517, 520 MPI\_FILE\_GET\_AMODE, 501 MPI\_FILE\_IREAD, 520 MPI\_FILE\_OPEN, 517, 520 MPI\_FILE\_READ, 517 MPI\_FILE\_SET\_ATOMICITY, 553 MPI\_FILE\_SET\_VIEW, 517, 520 MPI\_FILE\_SYNC, 553 MPI\_Finalize, 360, 361 MPI\_FREE\_MEM, 341 MPI\_Free\_mem, 472 MPI\_Gather, 139, 154, 155, 159 MPI\_Gatherv, 139, 156–159 MPI\_GET, 423, 424 MPI\_Get, 459, 461, 467, 468 MPI\_Get\_accumulate, 469, 472 MPI\_GET\_ADDRESS, 104, 657, 658, 679 MPI\_Get\_address, 127, 130, 131, 137 MPI\_GET\_COUNT, 116 MPI\_GET\_ELEMENTS, 116 MPI\_GRAPH\_CREATE, 296, 309 MPI\_GRAPH\_NEIGHBORS, 309 MPI\_GRAPH\_NEIGHBORS\_COUNT, 309 MPI\_Grequest\_complete, 482 MPI\_Grequest\_start, 482 MPI\_GROUP\_DIFFERENCE, 614 MPI\_Group\_excl, 253 MPI\_GROUP\_FREE, 614

MPI_Group_free, 243, 253, 254	1
MPI_Group_incl, 243, 254, 257	$^{2}$
MPI_Iallreduce, 222	3
MPI_Ialltoall, 221	4
MPI_Ibarrier, 219–222	5
MPI_Ibcast, 201, 222, 223	
MPI_INFO_ENV, 359	6
MPI_Intercomm_create, 265, 266	7
MPI_Iprobe, 361	8
MPI_IRECV, 56–58, 64, 65	9
MPI_Irecv, 222	10
MPI_ISEND, 56–58, 64, 65	11
MPI_Op_create, 188, 189, 197	
MPI_Pack, 137, 139	12
MPI_Pack_size, 139	13
MPI_PROBE, 68	14
MPI_Put, 444, 450, 460, 462, 466, 467	15
MPI_RECV, 35–37, 43–45, 58, 68, 114	16
	17
MPI_Recv, 221	
MPI_REDUCE, 179, 180, 183	18
MPI_Reduce, 182, 184, 188, 189	19
MPI_REQUEST_FREE, 57	20
MPI_Request_free, 360	21
MPI_Rget, 471	22
MPI_Rput, 471	23
MPI_Scan, 197	
MPI_Scatter, 164	24
MPI_Scatterv, 164, 165	25
MPI_SEND, 35–37, 44, 45, 58, 68, 114	26
MPI_Send, 127, 130, 131, 137, 221, 222	27
MPI_SENDRECV, 125, 126	28
MPI_SENDRECV_REPLACE, 313	29
$MPI\_SSEND, 43, 58$	
MPI_Test_cancelled, 361	30
MPI_TYPE_COMMIT, 112, 125, 126, 423,	31
657,658	32
MPI_Type_commit, 127, 130, 131, 137,	33
155-159, 165, 197	34
MPI_TYPE_CONTIGUOUS, 88, 107, 114, 116	35
MPI_Type_contiguous, 155	
MPI_TYPE_CREATE_DARRAY, 103	36
MPI_TYPE_CREATE_HVECTOR, 125, 126	37
MPI_Type_create_hvector, 127, 130	38
MPI_TYPE_CREATE_INDEXED_BLOCK,	39
423	40
MPI_TYPE_CREATE_RESIZED, 657, 658	41
MPI_TYPE_CREATE_STRUCT, 95, 107,	
126, 657, 658	42
MPI_Type_create_struct, 127, 130, 131, 137,	43
158, 159, 197	44
MPI_TYPE_CREATE_SUBARRAY, 561	45
MPI_TYPE_EXTENT, 423	46
MPI_TYPE_FREE, 423	47
MPI_Type_get_contents, 132	
$1 \times 1 \times$	48

742MPI\_Type\_get\_envelope, 132  $\mathbf{2}$ MPI\_TYPE\_GET\_EXTENT, 125, 126, 424, 4273 MPI\_Type\_get\_extent, 127 MPI\_TYPE\_INDEXED, 91, 125 MPI\_Type\_indexed, 127, 130 6 MPI\_TYPE\_VECTOR, 88, 89, 125, 126  $\overline{7}$ MPI\_Type\_vector, 156, 157, 159, 165 8 MPI\_Unpack, 137, 139 9 MPI\_User\_function, 189 MPI\_WAIT, 56-58, 64, 65, 520 10 MPI\_Wait, 219-222 11 MPI\_Waitall, 222, 471 12MPI\_WAITANY, 64 13MPI\_Waitany, 471 14MPI\_WAITSOME, 65 15MPI\_Win\_attach, 472 16MPI\_Win\_complete, 444, 461, 462, 467, 468 17MPI\_WIN\_CREATE, 423, 424, 427

1

4

5

- MPI\_Win\_create\_dynamic, 472 18
- MPI\_Win\_detach, 472 19
- MPI\_WIN\_FENCE, 423, 424, 427 20
- MPI\_Win\_fence, 466, 467 21
- MPI\_Win\_flush, 460, 469, 470, 472 22
- MPI\_Win\_flush\_all, 469 23
- MPI\_Win\_flush\_local, 459
- $^{24}$ MPI\_WIN\_FREE, 424, 427
- MPI\_Win\_lock, 450, 459-462 25
- MPI\_Win\_lock\_all, 471, 472 26
- MPI\_Win\_post, 461, 462, 467, 468 27
- MPI\_Win\_start, 444, 461, 462, 467, 468 28
- MPI\_Win\_sync, 459, 460, 469, 470 29
- shared memory windows, 471 30
- MPI Win unlock, 450, 459–462  $^{31}$
- MPI\_Win\_unlock\_all, 471, 472
- 32 MPI\_Win\_wait, 461, 462, 467, 468
- mpiexec, 359, 366 33
- 34Neighborhood collective communication, 331
- 35 No Matching of Blocking and Nonblocking
- 36 collective operations, 221
- 37 Non-deterministic program with MPI\_Bcast, 21838
- Non-overtaking messages, 43 39
- Nonblocking operations, 56, 57 40
- message ordering, 58 41
- progress, 58 42
- 43 Overlapping Communicators, 222
- 44Pipelining nonblocking collective operations, 45
- 22246
- Profiling interface, 566 47
- Progression of nonblocking collective 48
- operations, 221

- Shared memory windows MPI\_Win\_sync, 471
  - Threads and MPI, 487 Topologies, 331 Typemap, 87–89, 91, 95, 103
  - Using MPI\_T\_CVAR\_GET\_INFO to list all names of control variables., 578

Reading the value of a control variable in the

MPI tool information interface, 581

Virtual topologies, 331

# MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

MPI::\_LONG\_LONG, 719 MPI::BOOL, 718 MPI::COMPLEX, 718 MPI::DOUBLE\_COMPLEX, 718 MPI::F\_COMPLEX16, 719 MPI::F\_COMPLEX32, 719 MPI::F\_COMPLEX4, 719 MPI::F\_COMPLEX8, 719 MPI::INTEGER16, 719 MPI::LONG\_DOUBLE\_COMPLEX, 718 MPI::LONG\_LONG, 719 MPI::REAL16, 719 MPI\_2DOUBLE\_PRECISION, 182, 695 MPI\_2INT, 182, 695 MPI\_2INTEGER, 182, 695 MPI\_2REAL, 182, 695 MPI\_ADDRESS\_KIND, 15, 16, 16, 28, 268, 652, 681, 692 MPI\_AINT, 27, 29, 179, 413, 693, 694, 724, 726 MPI\_ANY\_SOURCE, 30, 31, 43, 53, 54, 66, 67, 69-71, 78, 81, 82, 289, 337, 602, 603, 609, 615, 691 MPI\_ANY\_TAG, 15, 30, 31, 33, 53, 54, 66, 67, 69-73, 78, 81-83, 691, 721 MPI\_APPNUM, 398, 399, 698 MPI\_ARGV\_NULL, 16, 378, 379, 652, 700 MPI\_ARGVS\_NULL, 16, 383, 652, 700 MPI\_ASYNC\_PROTECTS\_NONBLOCKING, 15, 471, 626, 627, 629, 631, 634, 641, 643, 663, 692, 723 MPI\_BAND, 178, 179, 696 MPI\_BOR, 178, 179, 696 MPI\_BOTTOM, 10, 15, 16, 34, 103, 117, 118, 146, 300, 302, 380, 413, 417, 628, 630, 637, 652, 657, 659, 661, 662, 664-666, 668, 679, 680, 686, 691, 730 MPI\_BSEND\_OVERHEAD, 48, 691 MPI\_BXOR, 178, 179, 696 MPI\_BYTE, 27, 28, 35, 36, 38, 140, 179, 494,

538, 551, 686, 693, 694, 727	13
MPI_C_BOOL, 28, 179, 693, 719, 724–726	14
MPI_C_COMPLEX, 28, 179, 693, 718,	15
724–726	16
MPI_C_DOUBLE_COMPLEX, 28, 179, 693,	17
724-726	18
MPI_C_FLOAT_COMPLEX, 179, 693,	19
724-726	
MPI_C_LONG_DOUBLE_COMPLEX, 28,	20
179, 693, 724-726	21
MPI_CART, 304, 697	22
MPI_CHAR, 28, 38, 95, 180, 181, 573, 574,	23
693, 724	24
MPI_CHARACTER, 27, 36–38, 180, 181, 694	25
MPI_COMBINER_CONTIGUOUS, 119, 122,	26
699	27
MPI_COMBINER_DARRAY, 119, 124, 699	28
MPI_COMBINER_DUP, 119, 122, 699	
MPI_COMBINER_F90_COMPLEX, 119, 124,	29
699	30
MPI_COMBINER_F90_INTEGER, 119, 124,	31
699	32
MPI_COMBINER_F90_REAL, 119, 124, 699	33
MPI_COMBINER_HINDEXED, 18, 119, 123,	34
699	35
MPI_COMBINER_HINDEXED_BLOCK, 119,	36
123, 699, 720 MPI_COMBINER_HINDEXED_INTEGER,	37
18, 624, 719	38
MPI_COMBINER_HVECTOR, 18, 119, 123,	39
699	40
MPI_COMBINER_HVECTOR_INTEGER,	
18, 624, 719	41
MPI_COMBINER_INDEXED, 119, 123, 699	42
MPI_COMBINER_INDEXED_BLOCK, 119,	43
123, 699	44
MPI_COMBINER_NAMED, 119, 122, 699	45
MPI_COMBINER_RESIZED, 119, 125, 699	46
MPI_COMBINER_STRUCT, 18, 119, 123, 699	47
MPI_COMBINER_STRUCT_INTEGER, 18,	48

> 5 6

1	624, 719
0	
2	MPI_COMBINER_SUBARRAY, 119, 124, 699
3	MPI_COMBINER_VECTOR, 119, 122, 699
4	MPI_COMM_DUP_FN, 18, <u>271</u> , 697, 723
5	MPI_COMM_NULL, 229, 242, 243, 245–247,
	249, 250, 285, 294, 296, 381, 400-402,
6	604, 605, 696, 727
7	MPI_COMM_NULL_COPY_FN, 18, <u>271</u> , 628,
8	680, 697, 723
9	
	MPI_COMM_NULL_DELETE_FN, 18, <u>271</u> ,
10	697
11	MPI_COMM_PARENT, 285
12	MPI_COMM_SELF, 229, 245, 268, 285, 363,
13	400,  496,  695,  725
	MPI_COMM_TYPE_SHARED, 250, 695, 721
14	MPI_COMM_WORLD, 15, 22, 29, 30,
15	229–231, 238, 239, 254, 263, 285, 295,
16	336-338, 342, 345, 353, 360, 361, 363,
17	365, 373, 374, 376, 377, 381, 383,
18	397-400, 490, 537, 557, 580, 588,
19	$602-604,\ 674,\ 685,\ 695,\ 729$
20	MPI_COMPLEX, 27, 179, 540, 644, 694
21	MPI_COMPLEX16, 179, 694
	MPI_COMPLEX32, 179, 694
22	MPI_COMPLEX4, 179, 694
23	MPI_COMPLEX8, 179, 694
24	MPI_CONGRUENT, 239, 261, 695
25	MPI_CONVERSION_FN_NULL, <u>545</u> , 697
26	MPI_COUNT, 27, 29, 179, 573, 693, 694, 720
20	MPI_COUNT_KIND, 15, 28, 692
27	MPI_CXX_BOOL, 29, 179, 694, 718
28	
29	MPI_CXX_DOUBLE_COMPLEX, 29, 179,
30	694, 718
	MPI_CXX_FLOAT_COMPLEX, 29, 179, 694,
31	718
32	MPI_CXX_LONG_DOUBLE_COMPLEX, 29,
33	179,694,718
34	MPI_DATATYPE_NULL, 113, 696
35	MPI_DISPLACEMENT_CURRENT, 506,
	700, 730
36	MPI_DIST_GRAPH, 304, 697, 725
37	MPI_DISTRIBUTE_BLOCK, 100, 700
38	MPI_DISTRIBUTE_CYCLIC, 100, 700
39	MPI_DISTRIBUTE_DFLT_DARG, 100, 700
40	MPI_DISTRIBUTE_NONE, 100, 700
	MPI_DOUBLE, 28, 178, 573, 582–584, 643,
41	
42	693
43	MPI_DOUBLE_COMPLEX, 27, 179, 540, 644,
44	694
	MPI_DOUBLE_INT, 182, 695
45	MPI_DOUBLE_PRECISION, 27, 178, 644,
46	694
47	MPI_DUP_FN, 18, 271, <u>620</u> , 698
48	MPI_EMPTY, 610
	_ / ·

MPI\_ERR\_ACCESS, 352, 499, 558, 690 MPI\_ERR\_AMODE, 352, 497, 558, 690 MPI\_ERR\_ARG, 351, 689 MPI\_ERR\_ASSERT, 351, 454, 690 MPI\_ERR\_BAD\_FILE, 352, 558, 690 MPI\_ERR\_BASE, 340, 351, 454, 690 MPI\_ERR\_BUFFER, 351, 689 MPI\_ERR\_COMM, 351, 689 MPI\_ERR\_CONVERSION, 352, 546, 558, 690 MPI\_ERR\_COUNT, 351, 689 MPI\_ERR\_DIMS, 351, 689 MPI\_ERR\_DISP, 351, 454, 690 MPI\_ERR\_DUP\_DATAREP, 352, 543, 558, 690 MPI\_ERR\_FILE, 352, 558, 690 MPI\_ERR\_FILE\_EXISTS, 352, 558, 690 MPI\_ERR\_FILE\_IN\_USE, 352, 499, 558, 690 MPI\_ERR\_GROUP, 351, 689 MPI\_ERR\_IN\_STATUS, 32, 34, 55, 61, 63, 344, 351, 481, 511, 690 MPI\_ERR\_INFO, 351, 690 MPI\_ERR\_INFO\_KEY, 351, 368, 690 MPI\_ERR\_INFO\_NOKEY, 351, 369, 690 MPI\_ERR\_INFO\_VALUE, 351, 368, 690 MPI\_ERR\_INTERN, 351, 689 MPI\_ERR\_IO, 352, 558, 690 MPI\_ERR\_KEYVAL, 281, 351, 690 MPI\_ERR\_LASTCODE, 350, 352, 353, 355, 598, 691 MPI\_ERR\_LOCKTYPE, 351, 454, 690 MPI\_ERR\_NAME, 351, 394, 690 MPI\_ERR\_NO\_MEM, 340, 351, 690 MPI\_ERR\_NO\_SPACE, 352, 558, 690 MPI\_ERR\_NO\_SUCH\_FILE, 352, 498, 558, 690 MPI\_ERR\_NOT\_SAME, 352, 558, 690 MPI\_ERR\_OP, 351, 454, 689 MPI\_ERR\_OTHER, 350, 351, 689 MPI\_ERR\_PENDING, 61, 351, 689 MPI\_ERR\_PORT, 351, 391, 690 MPI\_ERR\_PROC\_FAILED, 352, 603-610, 613, 690 MPI\_ERR\_PROC\_FAILED\_PENDING, 352, 603, 608, 609, 613, 690 MPI\_ERR\_QUOTA, 352, 558, 690 MPI\_ERR\_RANK, 351, 454, 689 MPI\_ERR\_READ\_ONLY, 352, 558, 690 MPI\_ERR\_REQUEST, 351, 689 MPI\_ERR\_REVOKED, 352, 604-608, 610-613, 690 MPI\_ERR\_RMA\_ATTACH, 352, 454, 690 MPI\_ERR\_RMA\_CONFLICT, 351, 454, 690 MPI\_ERR\_RMA\_FLAVOR, 352, 411, 454, 690 MPI\_ERR\_RMA\_RANGE, 352, 454, 690

MPI_ERR_RMA_SHARED, 352, 454, 690
MPI_ERR_RMA_SYNC, 351, 454, 690
MPI_ERR_ROOT, 351, 689
MPI_ERR_SERVICE, 351, 393, 690
MPI_ERR_SIZE, 351, 454, 690
MPI ERR SPAWN 351 379 380 605 690
MPI_ERR_TAG, 351, 689 MPI_ERR_TOPOLOGY, 351, 689 MPI_ERR_TRUNCATE, 351, 689 MPI_ERR_TYPE, 351, 689 MPI_ERR_TYPE, 351, 689
MPL EBB TOPOLOGY 351 689
MPLEBR TRUNCATE 351 689
MPI FRR TVPF 351 680
MPI_ERR_UNKNOWN, 350, 351, 689
MPI_ERR_UNSUPPORTED_DATAREP, 352,
$\frac{112}{558,690}$
MPI_ERR_UNSUPPORTED_OPERATION,
352, 558, 690 MDI EDD. WIN, 251, 454, 600
MPI_ERR_WIN, 351, 454, 690
MPI_ERRCODES_IGNORE, 16, 380, 652, 700
MPI_ERRHANDLER_NULL, 349, 696
MPI_ERROR, 32, 55, 199, 432, 692, 716, 722
MPI_ERRORS_ARE_FATAL, 342, 343, 355,
356, 454, 557, 602, 692
MPI_ERRORS_RETURN, 342, 343, 356, 557,
685,  692
MPI_F08_STATUS_IGNORE, 677, 701, 722
$\mathrm{MPI\_F08\_STATUSES\_IGNORE,\ 677,\ 701,\ 722}$
MPI_F_STATUS_IGNORE, 676, 701
MPI_F_STATUSES_IGNORE, 676, 701
MPI_FILE_NULL, 498, 557, 607, 696
MPI_FINALIZE, 361
MPI_FLOAT, 28, 95, 176, 178, 539, 693
MPI_FLOAT_INT, 12, 182, 695
MPI_FT, 337, 338, 602, 715
MPI_GRAPH, 304, 697
MPI_GROUP_EMPTY, 228, 234, 242, 243,
245,609,696
MPI_GROUP_NULL, 228, 237, 696
MPI_HOST, 337, 695
MPI_IDENT, 231, 239, 695
MPI_IN_PLACE, 16, 146, 173, 631, 652, 691
MPI_INFO_ENV, 358, 359, 695, 721
MPI_INFO_NULL, 302, 372, 380, 389, 497,
498, 507, 696
MPI_INT, 12, 28, 86, 178, 539, 540, 573, 574,
577, 582, 586, 643, 685, 687, 693
MPI_INT16_T, 28, 178, 693, 724, 726
MPI_INT32_T, 28, 178, 693, 724, 726
MPI_INT64_T, 28, 178, 693, 724, 726
MPI_INT8_T, 28, 178, 693, 724, 726
MPI_INTEGER, 27, 35, 178, 643, 644, 687,
694 MI 1_IN I EGEN, 27, 35, 178, 045, 044, 087,
MPI_INTEGER1, 27, 178, 694
MPI_INTEGER1, 27, 178, 694 MPI_INTEGER16, 178, 694
MPI_INTEGER2, 27, 178, 540, 694
MPI_INTEGER4, 27, 178, 694

MPI_INTEGER8, 178, 647, 694	1
MPI_INTEGER_KIND, 15, 692	$^{2}$
MPI_IO, 337, 695	3
MPI_KEYVAL_INVALID, 272, 273, 691	4
MPI_LAND, 178, 179, 696	5
MPI_LASTUSEDCODE, 353, 354, 698	
MPI_LB, 18, 624, 719	6
MPI_LOCK_EXCLUSIVE, 447, 691	7
MPI_LOCK_SHARED, 447, 448, 691	8
MPI_LOGICAL, 27, 179, 694	9
MPI_LONG, 28, 178, 693	10
MPI_LONG_DOUBLE, 28, 178, 693	11
MPI_LONG_DOUBLE_INT, 182, 695	12
MPI_LONG_INT, 182, 695	13
MPI_LONG_LONG, 28, 178, 693, 726	
MPI_LONG_LONG_INT, 28, 178, 693, 726	14
MPI_LOR, 178, 179, 696	15
MPI_LXOR, 178, 179, 696	16
MPI_MAX, 176, 178, 179, 196, 696	17
MPI_MAX_DATAREP_STRING, 15, 508,	18
543,692	19
MPI_MAX_ERROR_STRING, 15, 349, 354,	20
692	21
MPI_MAX_INFO_KEY, 15, 351, 367, 370, 692	22
MPI_MAX_INFO_VAL, 15, 351, 367, 692	
MPI_MAX_LIBRARY_VERSION_STRING,	23
15, 336, 692, 719	24
MPI_MAX_OBJECT_NAME, 15, 284–286,	25
692, 721, 727	26
MPI_MAX_PORT_NAME, 15, 389, 692	27
MPI_MAX_PROCESSOR_NAME, 15, 339,	28
692, 728	29
MPI_MAXLOC, 178, 181, 182, 185, 696	30
MPI_MESSAGE_NO_PROC, 70, 72, 73, 691,	31
720 MDI MESSACE NULL 70 72 72 606 720	32
MPI_MESSAGE_NULL, 70, 72, 73, 696, 720	
MPI_MIN, 178, 179, 696 MPI_MINLOC_178_181_182_185_606	33
MPI_MINLOC, 178, 181, 182, 185, 696	34
MPI_MODE_APPEND, 496, 497, 699 MPI_MODE_CREATE, 496, 497, 505, 699	35
MPI_MODE_DELETE_ON_CLOSE, 496–498,	36
699	37
MPI_MODE_EXCL, 496, 497, 699	38
MPI_MODE_NOCHECK, 448, 452, 453, 699	39
MPI_MODE_NOPRECEDE, 443, 452, 453, 039	40
699	
MPI_MODE_NOPUT, 452, 453, 699	41
MPI_MODE_NOSTORE, 452, 453, 699	42
MPI_MODE_NOSUCCEED, 452, 453, 699	43
MPI_MODE_RDONLY, 496, 497, 502, 699	44
MPI_MODE_RDWR, 496, 497, 699	45
MPI_MODE_SEQUENTIAL, 496, 497, 499,	46
500, 506, 511, 516, 528, 549, 699, 730	47
MPI_MODE_UNIQUE_OPEN, 496, 497, 699	48

1	MPI_MODE_WRONLY, 496, 497, 699
2	MPI_NO_OP, 429, 430, 696, 716
3	MPI_NULL_COPY_FN, 18, 271, 620, 698
	MPI_NULL_DELETE_FN, 18, 271, <u>620</u> , 698
4	MPI_OFFSET, 27, 179, 693, 694, 724, 726
5	MPI_OFFSET_KIND, 15, <u>17</u> , 28, 551, 652, 692
6	MPI_OP_NULL, 188, 696
7	
8	MPI_ORDER_C, 15, 97, 100, 101, 700
	MPI_ORDER_FORTRAN, 15, 97, 100, 700
9	MPI_PACKED, 12, 27, 28, 35, 134, 136, 140,
10	541, 686, 693, 694
11	MPI_PROC_NULL, 26, 29, 31, 32, 67, 70–73,
12	82, 83, 148, 150, 152, 154, 162, 164,
13	177, 231, 312, 316, 337, 411, 420, 691,
14	720, 727, 729
	MPI_PROD, 178, 179, 696
15	MPI_REAL, 27, 35, 178, 540, 643, 644, 650,
16	694
17	MPI_REAL16, 179, 694
18	MPI_REAL2, 27, 179, 694
19	MPI_REAL4, 27, 179, 643, 647, 694
20	MPI_REAL8, 27, 179, 643, 694, 724
	MPI_REPLACE, 427–430, 470, 696, 726, 729
21	MPI_REQUEST_NULL, 54–57, 59–63, 480,
22	696
23	MPI_ROOT, 148, 691
24	MPI_SEEK_CUR, 523, 529, 700
25	MPI_SEEK_END, 523, 529, 700
26	MPI_SEEK_SET, 523, 529, 607, 700
27	MPI_SHORT, 28, 178, 693
	MPI_SHORT_INT, 182, 695
28	MPI_SIGNED_CHAR, 28, 178, 180, 181, 693,
29	726
30	MPI_SIMILAR, 231, 239, 261, 695
31	MPI_SOURCE, 32, 199, 692, 716, 722
32	MPI_STATUS_IGNORE, 10, 15, 34, 479, 511,
33	630, 652, 676, 677, 686, 700, 701, 720
	MPI_STATUS_SIZE, 15, 32, 632, 692, 722
34	MPI_STATUSES_IGNORE, 14, 15, 34, 479,
35	481, 652, 676, 677, 700, 701
36	MPI_SUBARRAYS_SUPPORTED, 15, 626,
37	627, 630–634, 638–641, 653–655, 692,
38	722
39	
	MPI_SUBVERSION, 15, 336, 701
40	MPI_SUCCESS, 19, 21, 54, 61, 63, 271,
41	273–276, 278, 279, 350, 351, 355, 356,
42	380, 546, 569, 578, 586, 589, 591, 596,
43	598, 599, 603–607, 610, 620, 689
44	MPI_SUM, 178, 179, 685, 696
	MPI_T_BIND_MPI_COMM, 571, 702
45	MPI_T_BIND_MPI_DATATYPE, 571, 702
46	MPI_T_BIND_MPI_ERRHANDLER, 571,
47	702
48	MPI_T_BIND_MPI_FILE, 571, 702

MPI_T_BIND_MPI_GROUP, 571, 702
MPI_T_BIND_MPI_INFO, 571, 702
MPI_T_BIND_MPI_MESSAGE, 571, 702
MPI_T_BIND_MPI_OP, 571, 702
MPI_T_BIND_MPI_REQUEST, 571, 702
MPI_T_BIND_MPI_WIN, 571, 702
MPI_T_BIND_NO_OBJECT, 571, 577, 579,
586, 588, 702
MPI_T_CVAR_HANDLE_NULL, 580, 701
MPI_T_CVAR_READ, WRITE, 599
MPI_T_ENUM_NULL, 577, 586, 701
MPI_I_ENUM_NULL, 577, 580, 701
MPI_T_ERR_XXX, 598
MPI_T_ERR_CANNOT_INIT, 599, 691
$\mathrm{MPI\_T\_ERR\_CVAR\_SET\_NEVER,\ 581,\ 599,}$
691
MPI_T_ERR_CVAR_SET_NOT_NOW, 581,
599,691
MPI_T_ERR_INVALID, 599, 691, 718
MPI_T_ERR_INVALID_HANDLE, 588, 599,
691
MPI_T_ERR_INVALID_INDEX, 599, 691
MPI_T_ERR_INVALID_ITEM, 599, 691
MPI_T_ERR_INVALID_NAME, 578, 586,
596, 599, 691, 718
MPI_T_ERR_INVALID_SESSION, 599, 691
MPI_T_ERR_MEMORY, 599, 691
MPI_T_ERR_NOT_INITIALIZED, 599, 691
MPI_I_ERR_NOI_INIIIALIZED, 599, 091
MPI_T_ERR_OUT_OF_HANDLES, 599, 691
MPI_T_ERR_OUT_OF_SESSIONS, 599, 691
MPI_T_ERR_PVAR_NO_ATOMIC, 591, 599,
691
MPI_T_ERR_PVAR_NO_STARTSTOP, 589,
599, 691
MPI_T_ERR_PVAR_NO_WRITE, 590, 591,
599,691
MPI_T_PVAR_ALL_HANDLES, 589–592, 702
MPI_T_PVAR_CLASS_AGGREGATE, 583,
584,702
MPI_T_PVAR_CLASS_COUNTER, 583, 702
MPI_T_PVAR_CLASS_GENERIC, 584, 702
MPI_T_PVAR_CLASS_HIGHWATERMARK,
583, 702
MPI_T_PVAR_CLASS_LEVEL, 582, 702
MPI_T_PVAR_CLASS_LOWWATERMARK,
583, 702
MPI_T_PVAR_CLASS_PERCENTAGE, 583,
702
MPI_T_PVAR_CLASS_SIZE, 582, 702
MPI_1_PVAR_CLASS_SIZE, 582, 702 MPI_T_PVAR_CLASS_STATE, 582, 702
MPI_T_PVAR_CLASS_TIMER, 584, 702
MPI_T_PVAR_HANDLE_NULL, 589, 701
MPI_T_PVAR_SESSION_NULL, 587, 701 MPI_T_SCOPE_ALL, 577, 702

MPI_T_SCOPE_CONSTANT, 577, 702	MPI_UNWEIGHTED, 16, 299, 300, 302, 303,
MPI_T_SCOPE_GROUP, 577, 702	310, 311, 652, 700, 719, 725
MPI_T_SCOPE_GROUP_EQ, 577, 581, 702	MPI_VAL, 12, 674
MPI_T_SCOPE_LOCAL, 577, 702	MPI_VERSION, 15, 336, 701
MPI_T_SCOPE_READONLY, 577, 702	MPI_WCHAR, 28, 180, 181, 286, 540, 693, 726
MPI_T_VERBOSITY_MPIDEV_ALL, 570,	MPI_WEIGHTS_EMPTY, 16, 299, 300, 302,
701	652, 700, 719
MPI_T_VERBOSITY_MPIDEV_BASIC, 570,	MPI_WIN_BASE, 416, 685, 698
701	MPI_WIN_CREATE_FLAVOR, 416, 698
MPI_T_VERBOSITY_MPIDEV_DETAIL,	MPI_WIN_DISP_UNIT, 416, 698
570, 701	MPI_WIN_DUP_FN, <u>275</u> , 697
	MPI_WIN_FLAVOR_ALLOCATE, 417, 698
MPI_T_VERBOSITY_TUNER_ALL, 570, 701	
MPI_T_VERBOSITY_TUNER_BASIC, 570,	MPI_WIN_FLAVOR_CREATE, 417, 698
701 MDI TI MEDDOCITIN THINED DETAIL 570	MPI_WIN_FLAVOR_DYNAMIC, 417, 698
MPI_T_VERBOSITY_TUNER_DETAIL, 570,	MPI_WIN_FLAVOR_SHARED, 417, 698
	MPI_WIN_MODEL, 416, 438, 698
MPI_T_VERBOSITY_USER_ALL, 570, 701	MPI_WIN_NULL, 415, 606, 696
MPI_T_VERBOSITY_USER_BASIC, 570,	MPI_WIN_NULL_COPY_FN, <u>275</u> , 697
701	MPI_WIN_NULL_DELETE_FN, <u>275</u> , 697
MPI_T_VERBOSITY_USER_DETAIL, 570,	MPI_WIN_SEPARATE, 417, 438, 456, 698
701	MPI_WIN_SIZE, 416, 698
MPI_TAG, 32, 199, 692, 716, 722	MPI_WIN_UNIFIED, 417, 438, 457, 466, 698
MPI_TAG_UB, 29, 337, 681, 684, 695	MPI_WTIME_IS_GLOBAL, 337, 338, 357,
MPI_THREAD_FUNNELED, 490, 699	681,695
MPI_THREAD_MULTIPLE, 490, 492, 699,	
717	
MPI_THREAD_SERIALIZED, 490, 699	
MPI_THREAD_SINGLE, 490, 491, 699	
MPI_TYPE_DUP_FN, <u>278</u> , 697	
MPI_TYPE_NULL_COPY_FN, <u>278</u> , 697	
MPI_TYPE_NULL_DELETE_FN, <u>278</u> , 697,	
723	
MPI_TYPECLASS_COMPLEX, 649, 700	
MPI_TYPECLASS_INTEGER, 649, 700	
MPI_TYPECLASS_REAL, 649, 700	
MPI_UB, 4, 18, 624, 719	
MPI_UINT16_T, 28, 178, 693, 724, 726	
MPI_UINT32_T, 28, 178, 693, 724, 726	
MPI_UINT64_T, 28, 178, 693, 724, 726	
MPI_UINT8_T, 28, 178, 693, 724, 726	
MPI_UNDEFINED, 33, 60, 63, 64, 106, 109,	
$111, 116, 137, 230, 231, 246, 247, 304, \\214, 215, 645, 601, 720, 726$	
314, 315, 645, 691, 720, 726	
MPI_UNEQUAL, 231, 239, 261, 695	
MPI_UNIVERSE_SIZE, 376, 397, 398, 698	
MPI_UNSIGNED, 28, 178, 573, 582–584, 693	
MPI_UNSIGNED_CHAR, 28, 178, 180, 181,	
693	
MPI_UNSIGNED_LONG, 28, 178, 573,	
582-584, 693	
MPI_UNSIGNED_LONG_LONG, 28, 178,	
573, 582 - 584, 693, 726	
MPI_UNSIGNED_SHORT, 28, 178, 693	

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

46

# **MPI** Declarations Index

 This index refers to declarations needed in C, such as address kind integers, handles, etc. The underlined page numbers is the "main" reference (sometimes there are more than one when key concepts are discussed in multiple areas).

13	MPI_Aint, <u>16</u> , 16, 17, 27, <u>87</u> , 87, 89, 92, 95,
14	103-105, 108-110, 120, 141, 142, 405,
15	407, 409, 412, 413, 420, 422, 426, 428,
16	430-434,  436,  539,  543,  652,  681,  703
17	MPI_Comm, 12, <u>26</u> , 232, 237–242, <u>244</u> , 246,
18	$249, 250, \underline{251}, 261-264, 270, 272-274,$
19	<u>607–611</u> , 695, 696, 703
	MPI_Count, <u>17</u> , 17, 27, 703, 719
20	MPI_Datatype, <u>87</u> , 661, 693–696, 703
21	MPI_ERR, 350
22	MPI_Errhandler, <u>344</u> , 345–349, 675, 692, 696,
23	703 MPI_F08_status, <u>677</u> , 701, 703, 722
24	MPI_File, 347, 348, 356, <u>495</u> , 497, 499–501,
25	503, 505, 508, 511-529, 531-536, 539,
26	548, 549, 612, 675, 696, 703
27	MPI_Fint, <u>674</u> , 674, 701, 703, 726
28	MPI_Group, <u>230</u> , 230–237, 242, 262, 417, 443,
29	445, 501, 609, 612, 674, 675, 696, 703
30	MPI_Info, 339, <u>367</u> , 367–371, 376, 379, 382,
31	388, 390-394, 418, 495, 498, 503, 505,
32	675,  695,  696,  703,  729
	MPI_Message, <u>70</u> , 675, 691, 696, 703, 720
33	MPI_Offset, <u>17</u> , 17, 27, 499, 500, 505, 508,
34	511-516, 522, 523, 528, 529, 531, 532,
35	543, <u>551</u> , 551, 673, 703
36	MPI_Op, 176, <u>185</u> , 187, 189, 191–193, 195,
37	196, 211-216, 426, 428, 430, 434, 436, 675 - 696 - 702
38	675, 696, 703
39	MPI_Request, 51–53, <u>55</u> , 56, 57, 59–63, 66, 73, 76–79, 478, 481, 514–516, 520–522,
40	526, 611, 654, 675, 696, 703
41	MPI_Status, <u>30</u> , <u>32</u> – <u>34</u> , <u>55</u> , <u>56</u> , <u>59</u> – <u>63</u> , <u>66</u> , <u>67</u> ,
42	70-72, 74, 81, 82, 115, 479, 484, 485,
43	511-513, 517-519, 524, 525, 527, 528,
44	532-536, 629, 676-678, 700, 703, 716,
45	719, 722
46	MPI_T_cvar_handle, <u>579</u> , 579, 580, 701
40	MPI_T_enum, <u>574</u> , 574–576, 585, 701
47	MPI_T_ERR_XXX, 598
-±0	

 $\begin{array}{c} \text{MPI\_T\_pvar\_handle, } \underline{587}, 587{-}591, 701 \\ \text{MPI\_T\_pvar\_session, } \underline{587}, 587{-}591, 701 \\ \text{MPI\_Win, } 275{-}277, 286, 287, 346, 347, 355, \\ \underline{405}, \underline{407}, \underline{409}, \underline{412}, 415, 417, 418, 420, \\ 422, 426, 428, 430{-}434, 436, 442{-}452, \\ \underline{611}, \underline{612}, 675, 696, 703 \\ \end{array}$ 

# MPI Callback Function Prototype Index

This index lists the C type def names for callback routines, such as those used with attribute caching or user-defined reduction operations. For tran example prototypes are given near the text of the C name.

	14
MPI_Comm_copy_attr_function, 18, 19, 270,	15
628,697,704	16
MPI_Comm_delete_attr_function, 18, 270,	17
697, 704	18
MPI_Comm_errhandler_fn, 622, 725	
MPI_Comm_errhandler_function, 18, 344, 622,	19
624, 704, 725	20
MPI_Copy_function, 18, 619, 698, 709	21
MPI_Datarep_conversion_function, 544, 697,	22
704	23
MPI_Datarep_extent_function, 543, 704	24
MPI_Delete_function, 18, 620, 698, 709	25
MPI_File_errhandler_fn, 622, 725	26
MPI_File_errhandler_function, 347, 622, 704,	
725	27
MPI_Grequest_cancel_function, 480, 704	28
MPI_Grequest_free_function, 479, 704	29
MPI_Grequest_query_function, 479, 704	30
MPI_Handler_function, 18, 624, 719	31
MPI_Type_copy_attr_function, 278, 697, 704	32
MPI_Type_delete_attr_function, 278, 697,	33
704, 723	
MPI_User_function, 185, 189, 704	34
MPI_Win_copy_attr_function, 275, 697, 704	35
MPI_Win_delete_attr_function, 275, 697, 704	36
MPI_Win_errhandler_fn, 622, 725	37
MPI_Win_errhandler_function, 346, 622, 704,	38
725	39
	40

## **MPI** Function Index

5 6

78 9 The underlined page numbers refer to the function definitions. 10 11MPI\_ABORT, 186, 342, 359, 362, 400, 573, 12674, 729 MPI\_ACCUMULATE, 403, 419, <u>426</u>, 427, 429, 13435, 439, 463, 469, 470, 726, 729 14MPI\_ADD\_ERROR\_CLASS, 353, 353 15MPI\_ADD\_ERROR\_CODE, 354 16MPI\_ADD\_ERROR\_STRING, 354, 354, 355 17MPI\_ADDRESS, 18, 623, 637, 719 18MPI\_AINT\_ADD, 20, 103, 104, 105, 105, 413, 19717 MPI\_AINT\_DIFF, 20, 103, 104, 105, 105, 413, 2071721MPI\_ALLGATHER, 143, 147, 148, 167, 22167-170, 206 23MPI\_ALLGATHERV, 143, 147, 148, 168, 169,  $^{24}$ 20725MPI\_ALLOC\_MEM, 339, 340, 341, 351, 26407-411, 414, 421, 450, 638-640, 652, 27716, 723 MPI\_ALLOC\_MEM\_CPTR, 340, 716 28 MPI\_ALLREDUCE, 143, 146-148, 178, 185, 29189, 190, 212, 616, 727 30 MPI\_ALLTOALL, 143, 147, 148, 170, 170-173,  $^{31}$ 208.724 32 MPI\_ALLTOALLV, 143, 147, 148, 172, 172, 33 173, 175, 209, 724 34MPI\_ALLTOALLW, 143, 147, 148, <u>174</u>, 175, 211, 724 35MPI\_ATTR\_DELETE, 18, 281, 620, 621 36 MPI\_ATTR\_GET, 18, 281, 621, 681, 682 37 MPI\_ATTR\_PUT, 18, 281, 621, 681, 682, 684, 38 685 39 MPI\_BARRIER, 143, 147, <u>149</u>, 149, 200, 460, 40461, 553  $^{41}$ MPI\_BCAST, 143, 147, 150, 150, 177, 201, 42220, 603 MPI\_BSEND, 40, 48 43 MPI\_BSEND\_INIT, 76, 79 44MPI\_BUFFER\_ATTACH, 46, 55 45MPI\_BUFFER\_DETACH, 47, 722 46MPI\_CANCEL, 43, 55, 66, 73, 73-75, 199, 361, 47432, 477, 480, 481 48

MPI\_CART\_COORDS, 293, 307, 307, 728 MPI\_CART\_CREATE, 260, 292, 293, 294, 294-296, 306, 314-316, 653, 716, 727 MPI\_CART\_GET, 293, 306, 306, 728 MPI\_CART\_MAP, 293, <u>314</u>, 315, 721 MPI\_CART\_RANK, 293, 307, 307, 728 MPI\_CART\_SHIFT, 293, 312, 312, 313, 316, 728 MPI\_CART\_SUB, 293, <u>313</u>, 314, 315, 728 MPI\_CARTDIM\_GET, 293, 306, 306, 728 MPI\_CLOSE\_PORT, 389, 389, 393 MPI\_COMM\_ACCEPT, 388, 390, 390, 391, 398, 399, 604 MPI\_COMM\_AGREE, 604, 607-609, 610, 610, 611, 613, 614, 616 MPI\_COMM\_C2F, 674 MPI\_COMM\_CALL\_ERRHANDLER, 355, 356 MPI\_COMM\_COMPARE, 239, 261 MPI\_COMM\_CONNECT, 351, 391, 391, 398, 399, 604 MPI\_COMM\_CREATE, 237, 239, 242, 242-247, 293, 725 MPI\_COMM\_CREATE\_ERRHANDLER, 18, 343, 344, 345, 623, 706, 708, 723 MPI\_COMM\_CREATE\_GROUP, 239, 244, 245, 246, 721 MPI\_COMM\_CREATE\_KEYVAL, 18, 268, 269, 271, 272, 281, 619, 680, 681, 705, 707, 723, 727 MPI\_COMM\_DELETE\_ATTR, 18, 268, 271-273, 274, 281, 621 MPI\_COMM\_DISCONNECT, 281, 381, 399, <u>400</u>, 400, 605 MPI\_COMM\_DUP, 232, 237, 239, 240, 240, 241, 243, 250, 251, 262, 264, 268, 271, 274, 281, 288, 603, 619, 720 MPI\_COMM\_DUP\_FN, 18, 271, 271, 272, 633, 697, 717, 723 MPI\_COMM\_DUP\_WITH\_INFO, 239, 240, 241, 250, 720 MPI\_COMM\_F2C, 674

MPI_COMM_FAILURE_ACK, <u>609</u> , 609, 610, 614	Ν
MPI_COMM_FAILURE_GET_ACKED, <u>609</u> ,	Ν
610, 614 MPI_COMM_FREE, 237, 240, <u>250</u> , 250, 262,	N N
264, 271, 272, 274, 281, 360, 363, 381, 399–401, 604, 620	Ν
MPI_COMM_FREE_KEYVAL, 18, 268, <u>272</u> , 281, 620	
MPI_COMM_GET_ATTR, 18, 268, <u>273</u> , 273, 281, 336, 621, 634, 681, 682, 684	Ν
$\begin{array}{c} \text{MPI_COMM\_GET\_ERRHANDLER, 18, 343,} \\ 345, 623, 728 \end{array}$	Ν
MPI_COMM_GET_INFO, <u>251</u> , 252, 720	Ν
MPI_COMM_GET_NAME, <u>284</u> , 284, 285, 727	N
MPI_COMM_GET_PARENT, 285, 377, <u>380</u> ,	N
380, 381, 604	10
MPI_COMM_GROUP, 14, 230, <u>232</u> , 232, 237,	Ν
238, 261, 343, 728	Ν
MPI_COMM_IAGREE, <u>608</u> , <u>611</u>	
MPI_COMM_IDUP, 237, 239, <u>241</u> , 241, 250,	Ν
251, 259, 268, 271, 274, 281, 716, 720	Ν
MPI_COMM_JOIN, <u>401</u> , 401, 402, 605	Ν
MPI_COMM_NULL_COPY_FN, 18, <u>271</u> , 271,	Ν
272, 628, 680, 697, 717, 723	
MPI_COMM_NULL_DELETE_FN, 18, <u>271</u> , 271, 272, 697, 717	Ν
MPI_COMM_RANK, <u>238</u> , 238, 261, 635	Ν
MPI_COMM_RANK_F08, 635	1.
MPI_COMM_REMOTE_GROUP, <u>262</u>	Ν
MPI_COMM_REMOTE_SIZE, <u>262</u> , <u>262</u>	
MPI_COMM_REVOKE, <u>607</u> , <u>608</u> , <u>616</u>	Ν
MPI_COMM_SET_ATTR, 18, 268, 271, <u>272</u> ,	Ν
281,620,634,681,682,685	Ν
MPI_COMM_SET_ERRHANDLER, 18, 343,	
345, 623	Ν
MPI_COMM_SET_INFO, 250, <u>251</u> , 251, 716, 720	Ν
MPI_COMM_SET_NAME, <u>283</u> , 283, 284	
MPI_COMM_SHRINK, <u>608</u> , 608, 609, 614	Ν
MPI_COMM_SIZE, <u>237</u> , 238, 261	Ν
MPI_COMM_SPAWN, 358, 365, 374, 375, <u>376</u> ,	Ν
376, 377, 379–381, 383–385, 397–399,	Ν
604 MDL COMM CDANNA MULTIDLE 250, 205	٦
MPI_COMM_SPAWN_MULTIPLE, 359, 365,	Ν
374, 375, 380, <u>382</u> , 383, 398, 399, 604 MPL COMM SPLIT, 220, 242, 242, 246	N
MPI_COMM_SPLIT, 239, 242, 243, <u>246</u> , 246–248, 288, 203, 204, 206, <u>314–316</u>	N N
246–248, 288, 293, 294, 296, 314–316, 603, 608, 613, 725	N
MPI_COMM_SPLIT_TYPE, <u>249</u> , 250, 721	N
MPI_COMM_SPERI_TIPE, <u>249</u> , 250, 721 MPI_COMM_TEST_INTER, <u>260</u> , <u>261</u>	IV
MPI_COMM_WORLD, 491	Ν
$MPI_COMPARE_AND_SWAP, 403, 419, 431,$	N
469	10

MPI_CONVERSION_FN_NULL, <u>545</u> , 697,	1
717	2
MPI_CWIN_GET_ATTR, 634	3
MPI_DIMS_CREATE, 293, 294, 295, 295	4
MPI_DIST_GRAPH_CREATE, 250, 292, 293,	5
298, 300, 301, 303, 311, 312, 316, 725	
MPI_DIST_GRAPH_CREATE_ADJACENT,	6
250, 292, 293, <u>298</u> , 298, 299, 303, 311,	7
316, 721, 725	8
MPI_DIST_GRAPH_NEIGHBORS, 293, 310,	9
<u>311</u> , 311, 316, 721, 725	10
MPI_DIST_GRAPH_NEIGHBORS_COUNT,	11
293, 310, 310–312, 718, 725	
MPI_DUP_FN, 18, 271, <u>620</u> , 698	12
MPI_ERRHANDLER_C2F, <u>675</u>	13
MPI_ERRHANDLER_CREATE, 18, 623, 719,	14
723	15
	16
MPI_ERRHANDLER_F2C, <u>675</u>	
MPI_ERRHANDLER_FREE, 343, <u>349</u> , 360,	17
728	18
MPI_ERRHANDLER_GET, 18, 623, 719, 728	19
MPI_ERRHANDLER_SET, 18, 623, 719	20
MPI_ERROR_CLASS, <u>350</u> , 350, 598	21
MPI_ERROR_STRING, <u>349</u> , 349, 350, 353,	22
355	
MPI_EXSCAN, 144, 147, 178, 185, <u>196</u> , 196,	23
216, 725	24
MPI_F_SYNC_REG, 104, 471, 626, <u>642</u> , 642,	25
643, 662, 664-667, 723	26
MPI_FETCH_AND_OP, 403, 419, 427, 429,	27
430, 430	28
MPI_FILE_C2F, <u>675</u>	
MPI_FILE_CALL_ERRHANDLER, 356, 356	29
MPI_FILE_CLOSE, 400, 495, 496, 497, 498,	30
607	31
MPI_FILE_CREATE_ERRHANDLER, 343,	32
<u>347</u> , 348, 706, 708, 723	33
MPI_FILE_DELETE, 497, <u>498</u> , 498, 502, 505,	34
557	34
MPI_FILE_F2C, <u>675</u>	35
MPI_FILE_GET_AMODE, <u>501</u> , 501	36
MPI_FILE_GET_ATOMICITY, <u>549</u> , 549	37
MPI_FILE_GET_BYTE_OFFSET, 516, 523,	38
	39
523, 524, 529 MDI EU E CET EDDIIANDI ED 242, 248	
MPI_FILE_GET_ERRHANDLER, 343, <u>348</u> ,	40
557, 728	41
MPI_FILE_GET_GROUP, <u>501</u> , 501	42
MPI_FILE_GET_INFO, <u>503</u> , 503, 505, 729	43
MPI_FILE_GET_POSITION, <u>523</u> , 523	44
MPI_FILE_GET_POSITION_SHARED, 528,	45
<u>529</u> , 529, 549	
MPI_FILE_GET_SIZE, <u>500</u> , 501, 552	46
MPI_FILE_GET_TYPE_EXTENT, <u>539</u> , 539,	47
545	48

1	MDI EILE CET VIEW FOO FOO
	MPI_FILE_GET_VIEW, <u>508</u> , 508
2	MPI_FILE_IXXX, 510
3	MPI_FILE_IREAD, 509, <u>520</u> , 520, 530, 547
4	MPI_FILE_IREAD_ALL, 509, <u>521</u> , 521, 718
5	MPI_FILE_IREAD_AT, 509, <u>514</u> , 514
	MPI_FILE_IREAD_AT_ALL, 509, 514, 515,
6	718
7	MPI_FILE_IREAD_SHARED, 509, 526, 526
8	MPI_FILE_IWRITE, 509, <u>521</u> , 522
9	
	MPI_FILE_IWRITE_ALL, 509, <u>522</u> , 522, 718
10	MPI_FILE_IWRITE_AT, 509, <u>515</u> , 515
11	$MPI\_FILE\_IWRITE\_AT\_ALL, 509, \underline{516}, 516,$
12	718
13	MPI_FILE_IWRITE_SHARED, 509, <u>526</u> , 527
	MPI_FILE_OPEN, 352, 488, <u>495</u> , 495–497,
14	502, 504-506, 524, 551, 552, 557, 558
15	MPI_FILE_PREALLOCATE, 499, 500, 500,
16	547, 552
17	MPI_FILE_READ, 508, 509, <u>517</u> , 517, 518,
	520, 551, 552
18	· · ·
19	MPI_FILE_READ_ALL, 509, <u>518</u> , 518, 521,
20	530, 531
21	MPI_FILE_READ_ALL_BEGIN, 509, 530,
22	$531, \underline{533}, 547, 667$
	MPI_FILE_READ_ALL_END, 509, 530, 531,
23	$\underline{533}, 547, 667$
24	MPI_FILE_READ_AT, 509, <u>511</u> , 512, 514
25	MPI_FILE_READ_AT_ALL, 509, <u>512</u> , 512,
26	515
27	MPI_FILE_READ_AT_ALL_BEGIN, 509,
28	531, 667
	MPI_FILE_READ_AT_ALL_END, 509, 532,
29	667
30	MPI_FILE_READ_ORDERED, 509, <u>527</u> , 528
31	MPI_FILE_READ_ORDERED_BEGIN, 509,
32	535, 667
33	MPI_FILE_READ_ORDERED_END, 509,
	<u>535</u> , 667
34	MPI_FILE_READ_SHARED, 509, <u>524</u> , 525,
35	526, 528
36	MPI_FILE_REVOKE, <u>612</u> , 612
37	
38	MPI_FILE_SEEK, <u>522</u> , 523, 607
	MPI_FILE_SEEK_SHARED, <u>528</u> , 528, 529,
39	549
40	MPI_FILE_SET_ATOMICITY, 497, 547, <u>548</u> ,
41	548
42	MPI_FILE_SET_ERRHANDLER, 343, <u>348</u> ,
43	557
44	MPI_FILE_SET_INFO, 502, <u>503</u> , 503–505, 729
	MPI_FILE_SET_SIZE, <u>499</u> , 499, 500, 547,
45	550, 552
46	MPI_FILE_SET_VIEW, 98, 352, 496, 502,
47	$504,  \underline{505},  505 - 507,  523,  529,  537,  543,$
48	551,558,729,730

MPI_FILE_SYNC,	498,	509,	546-548,	<u>549</u> ,
549,555				

- MPI\_FILE\_WRITE, 508, 509, <u>518</u>, 519, 522, 551
- MPI\_FILE\_WRITE\_ALL, 509, <u>519</u>, 519, 522
- MPI\_FILE\_WRITE\_ALL\_BEGIN, 509, <u>534</u>, 654, 667
- MPI\_FILE\_WRITE\_ALL\_END, 509, 534, 667
- MPI\_FILE\_WRITE\_AT, 509, <u>513</u>, 513–515
- MPI\_FILE\_WRITE\_AT\_ALL, 509, <u>513</u>, 514, 516
- MPI\_FILE\_WRITE\_AT\_ALL\_BEGIN, 509, 532, 667
- $\begin{array}{c} \text{MPI_FILE\_WRITE\_AT\_ALL\_END, 509, } \underline{533}, \\ 667 \end{array}$
- MPI\_FILE\_WRITE\_ORDERED, 509, 527, <u>528</u>, 528
- MPI\_FILE\_WRITE\_ORDERED\_BEGIN, 509, <u>536</u>, 667
- MPI\_FILE\_WRITE\_ORDERED\_END, 509, 536, 667
- MPI\_FILE\_WRITE\_SHARED, 509, <u>525</u>, 525, 527, 528
- MPI\_FINALIZE, 15, 22, 336, 337, <u>359</u>, 359–364, 400, 487, 496, 569, 579, 592, 594, 603, 674, 676, 677, 715, 721, 729
- MPI\_FINALIZED, 358, 361, 363, <u>364</u>, 364, 486, 492, 674, 717
- MPI\_FREE\_MEM, <u>340</u>, 340, 341, 351, 408, 409
- MPI\_GATHER, 143, 146, 147, <u>151</u>, 153, 154, 161, 162, 167, 177, 203
- MPI\_GATHERV, 143, 147, <u>153</u>, 153–155, 163, 169, 204
- MPI\_GET, 403, 419, <u>422</u>, 423, 429, 434, 439, 459, 462, 471, 606, 665, 729
- MPI\_GET\_ACCUMULATE, 403, 419, 427, <u>428</u>, 429, 430, 437, 463, 469, 716
- MPI\_GET\_ADDRESS, 18, 87, 103, <u>104</u>, 104, 105, 117, 413, 623, 637, 656, 661, 679, 680
- MPI\_GET\_COUNT, <u>33</u>, 33, 34, 54, 116, 432, 485, 511, 720, 726
- MPI\_GET\_ELEMENTS, 54, <u>115</u>, 115, 116, 485, 486, 511, 720
- MPI\_GET\_ELEMENTS\_X, 54, <u>115</u>, 115, 116, 485, 511, 719
- MPI\_GET\_LIBRARY\_VERSION, <u>336</u>, 336, 358, 361, 486, 716, 717, 719
- MPI\_GET\_PROCESSOR\_NAME, <u>338</u>, 339, 728
- MPI\_GET\_VERSION, <u>335</u>, 336, 358, 361, 486, 492, 641, 717

MPI\_GRAPH\_CREATE, 292, 293, 296, 296,

298, 302, 305, 308, 309, 315, 316, 727, 728 MPI\_GRAPH\_GET, 293, <u>305</u>, 305 MPI\_GRAPH\_MAP, 293, 315, 316 MPI\_GRAPH\_NEIGHBORS, 293, 308, 308, 309, 316, 725 MPI\_GRAPH\_NEIGHBORS\_COUNT, 293, <u>308</u>, 308, 309, 725 MPI\_GRAPHDIMS\_GET, 293, 305, 305 MPI\_GREQUEST\_COMPLETE, 478–480, 481.481 MPI\_GREQUEST\_START, <u>478</u>, 479, 706, 708, 726 MPI\_GROUP\_C2F, 675 MPI\_GROUP\_COMPARE, 231, 234 MPI\_GROUP\_DIFFERENCE, 233, 610 MPI\_GROUP\_EXCL, 234, 235, 236 MPI\_GROUP\_F2C, <u>675</u> MPI\_GROUP\_FREE, 237, 237, 238, 343, 360, 728 MPI\_GROUP\_INCL, 234, 234, 236 MPI\_GROUP\_INTERSECTION, 233  $\mathrm{MPI\_GROUP\_RANGE\_EXCL},\,\underline{236},\,236$ MPI\_GROUP\_RANGE\_INCL, 235, 236 MPI\_GROUP\_RANK, <u>230</u>, 238 MPI\_GROUP\_SIZE, <u>230</u>, 238 MPI\_GROUP\_TRANSLATE\_RANKS, 231, 231.727 MPI\_GROUP\_UNION, 232 MPI\_IALLGATHER, 143, 147, 148, 206 MPI\_IALLGATHERV, 143, 147, 148, 207 MPI\_IALLREDUCE, 143, 147, 148, 212 MPI\_IALLTOALL, 143, 147, 148, 208 MPI\_IALLTOALLV, 143, 147, 148, 209 MPI\_IALLTOALLW, 143, 147, 148, 210 MPI\_IBARRIER, 143, 147, 199, 200, 200, 201, 220MPI\_IBCAST, 143, 147, 201, 201, 224 MPI\_IBSEND, 51, 55, 79 MPI\_IEXSCAN, 144, 147, 216 MPI\_IGATHER, 143, 147, 202 MPI\_IGATHERV, 143, 147, 203 MPI\_IMPROBE, 66, 69, <u>70</u>, 70, 71, 73, 488, 716, 720 MPI\_IMRECV, 69–71, 73, 73, 720 MPI\_INEIGHBOR\_ALLGATHER, 293, 326, 721 MPI\_INEIGHBOR\_ALLGATHERV, 293, 327, 721MPI\_INEIGHBOR\_ALLTOALL, 293, 328, 721 MPI\_INEIGHBOR\_ALLTOALLV, 293, 329, 721MPI\_INEIGHBOR\_ALLTOALLW, 294, 330, 721

MPI_INFO_C2F, <u>675</u>	1
MPI_INFO_CREATE, <u>368</u> , 368	2
MPI_INFO_DELETE, 351, <u>369</u> , 369, 371	3
MPI_INFO_DUP, <u>371</u> , 371	
	4
MPI_INFO_F2C, <u>675</u>	5
MPI_INFO_FREE, 252, 360, <u>371</u> , 419, 503	6
MPI_INFO_GET, 367, <u>369</u> , 729	7
MPI_INFO_GET_NKEYS, 367, <u>370</u> , 370, 371,	
729	8
MPI_INFO_GET_NTHKEY, 367, <u>371</u> , 729	9
MPI_INFO_GET_VALUELEN, 367, <u>370</u> , 729	10
MPI_INFO_SET, <u>368</u> , 368, 369, 371	11
MPI_INIT, 15, 22, 229, 336, 337, <u>358</u> , 358,	12
361 - 364, 377 - 379, 381, 397, 398, 489,	
490, 492, 565, 569, 572–574, 579, 592,	13
598, 603-605, 673, 674, 676, 677, 720,	14
721, 724, 725	18
MPI_INIT_THREAD, 229, 358, 363, <u>489</u> ,	16
	17
490-492, 572, 598, 673, 720, 721, 725	
MPI_INITIALIZED, 358, 361, <u>362</u> , 362–364,	18
486, 492, 674, 717	19
MPI_INTERCOMM_CREATE, 239, 245, 262,	20
$\underline{263}, 263, 264, 721$	21
MPI_INTERCOMM_MERGE, 239, 245, 260,	
$262, 263, \underline{264}, 264, 723$	22
MPI_IPROBE, 33, <u>66</u> , 66–71, 73, 488, 720	23
MPI_IRECV, <u>53</u> , 73, 655, 656, 659, 660	24
MPI_IREDUCE, 143, 147, 148, <u>211</u> , 212	25
MPI_IREDUCE_SCATTER, 143, 147, 148,	26
214	
MPI_IREDUCE_SCATTER_BLOCK, 143,	27
147, 148, <u>213</u>	28
MPI_IRSEND, <u>53</u>	29
MPI_IS_THREAD_MAIN, 486, 490, <u>492</u> , 717	30
MPI_ISCAN, 144, 147, <u>215</u>	31
	32
MPI_ISCATTER, 143, 147, <u>204</u>	
MPI_ISCATTERV, 143, 147, <u>205</u>	33
MPI_ISEND, <u>51</u> , 79, 633, 634, 637, 654, 655,	34
660, 661	35
MPI_ISSEND, $52$	36
MPI_KEYVAL_CREATE, 18, <u>619</u> , 621, 709	37
MPI_KEYVAL_FREE, 18, 281, <u>620</u>	
MPI_LOOKUP_NAME, 351, 388, 392, <u>394</u> ,	38
394	39
MPI_MESSAGE_C2F, <u>675</u> , 720	40
MPI_MESSAGE_F2C, <u>675</u> , 720	41
MPI_MPROBE, 66, 69, 70, 71, 71, 73, 488, 720	42
MPI_MRECV, 69–71, <u>72</u> , 72, 73, 720	
$MPI_NEIGHBOR_ALLGATHER, 293, 317,$	43
319, 320, 326, 721	44
MPI_NEIGHBOR_ALLGATHERV, 293, <u>319</u> ,	45
327, 721	46
MPI_NEIGHBOR_ALLTOALL, 293, <u>321</u> , 322,	47
328,721	48

1	MPI_NEIGHBOR_ALLTOALLV, 293, <u>322</u> ,
2	329, 721
3	MPI_NEIGHBOR_ALLTOALLW, 293, 323,
4	<u>324, 331, 721</u>
5	MPI_NULL_COPY_FN, 18, 19, 271, <u>620</u> , 698
6	MPI_NULL_DELETE_FN, 18, 271, <u>620</u> , 698
7	MPI_OP_C2F, <u>675</u> MPI_OP_COMMUTATIVE_101_725
8	MPI_OP_COMMUTATIVE, <u>191</u> , 725
9	MPI_OP_CREATE, <u>185</u> , 185, 187, 633, 704,
	707, 722, 723
10	MPI_OP_F2C, <u>675</u>
11	MPI_OP_FREE, <u>187</u> , 360 MPI_OPEN_POPT_288_288_200_201_202
12	MPI_OPEN_PORT, <u>388</u> , 388, 390, 391, 393,
13	394 MDI DACK 49, 194, 197, 140, 541, 544
14	MPI_PACK, 48, <u>134</u> , 137, 140, 541, 544
15	MPI_PACK_EXTERNAL, 8, 140, <u>141</u> , 647,
16	727 MDI DACK EXTEDNAL CIZE 149
	MPI_PACK_EXTERNAL_SIZE, <u>142</u>
17	MPI_PACK_SIZE, 48, <u>137</u> , 137, 720
18	MPI_PCONTROL, 564, <u>565</u> , 565
19	MPI_PROBE, 31, 33, 34, 66, <u>67</u> , 67–69, 71, 73,
20	488, 720 MDI DUDI ICH NAME 288, 202, 202, 204
21	MPI_PUBLISH_NAME, 388, <u>392</u> , 392–394
22	MPI_PUT, 403, 419, <u>420</u> , 422, 426, 427, 433,
23	439, 444, 454, 456, 460, 462, 471, 606,
24	654, 665, 729 MPI_QUERY_THREAD, 486, <u>491</u> , 492, 717
25	MPI_RACCUMULATE, 403, 419, 427, 429,
26	<u>434</u> , 435
	MPI_RECV, 26, <u>30</u> , 32–34, 67, 69, 70, 86, 114,
27	115, 135, 144, 152, 221, 486, 553, 592,
28	661, 664, 665
29	MPI_RECV_INIT, <u>78</u> , 78
30	MPI_REDUCE, 143, 147, 148, <u>176</u> , 176–178,
31	185-187, 190, 193-196, 211, 212, 426,
32	427, 429, 430, 726
33	MPI_REDUCE_LOCAL, 177, 178, 185, <u>191</u> ,
34	723, 725
	MPI_REDUCE_SCATTER, 143, 147, 148,
35	$178, 185, \underline{193}, 193, 194, 214$
36	MPI_REDUCE_SCATTER_BLOCK, 143,
37	147, 148, 178, 185, <u>192</u> , 192, 193, 213,
38	725
39	MPI_REGISTER_DATAREP, 352, 541,
40	543-545, 558, 706, 709
41	MPI_REQUEST_C2F, 675
42	MPI_REQUEST_F2C, 675
43	MPI_REQUEST_FREE, <u>57</u> , 57, 74, 79, 199,
	360, 432, 480, 481, 724
44	MPI_REQUEST_GET_STATUS, 34, <u>66</u> , 66,
45	479, 724
46	MPI_RGET, 403, 419, <u>433</u> , 434
47	MPI_RGET_ACCUMULATE, 403, 419, 427,
48	429, <u>436</u> , 437

MPI\_RPUT, 403, 419, 432, 433 MPI\_RSEND, 41 MPI\_RSEND\_INIT, 77 MPI\_SCAN, 144, 147, 178, 185, 195, 195, 197, 215MPI\_SCATTER, 143, 147, 161, 161, 163, 164, 193, 204 MPI\_SCATTERV, 143, 147, 163, 163, 164, 194, 205 MPI\_SEND, 25, 26, 27, 34, 36, 86, 113, 114, 134, 221, 496, 553, 566, 661, 662, 664, 665 MPI\_SEND\_INIT, <u>76</u>, 79 MPI\_SENDRECV, 81, 312 MPI\_SENDRECV\_REPLACE, 82 MPI\_SIZEOF, 626, 648, 649 MPI\_SSEND, 41 MPI\_SSEND\_INIT, 77 MPI\_START, 78, 79, 79, 80, 661 MPI\_STARTALL, 79, 79, 661 MPI\_STATUS\_C2F, 676 MPI\_STATUS\_C2F08, 677, 722 MPI\_STATUS\_F082C, <u>677</u>, 722 MPI\_STATUS\_F082F, 678, 722 MPI\_STATUS\_F2C, 676 MPI\_STATUS\_F2F08, 678, 722 MPI\_STATUS\_SET\_CANCELLED, 485 MPI\_STATUS\_SET\_ELEMENTS, <u>484</u>, 485 MPI\_STATUS\_SET\_ELEMENTS\_X, 485, 485, 719 MPI\_T\_CATEGORY\_CHANGED, 597 MPI\_T\_CATEGORY\_GET\_CATEGORIES, 597, 597–599 MPI\_T\_CATEGORY\_GET\_CVARS, 596, 596, 598, 599 MPI\_T\_CATEGORY\_GET\_INDEX, 596, 596, 599, 718 MPI\_T\_CATEGORY\_GET\_INFO, 595, 595, 598, 599, 717, 718 MPI\_T\_CATEGORY\_GET\_NUM, 595 MPI\_T\_CATEGORY\_GET\_PVARS, <u>597</u>, 597 - 599MPI\_T\_CVAR\_GET\_INDEX, <u>578</u>, 578, 599, 718 MPI\_T\_CVAR\_GET\_INFO, 574, 576, 576, 577, 579–581, 598, 599, 717, 718 MPI\_T\_CVAR\_GET\_NUM, 576, 580 MPI\_T\_CVAR\_HANDLE\_ALLOC, 574, 579, 580, 581, 599 MPI\_T\_CVAR\_HANDLE\_FREE, 580, 580, 599 MPI\_T\_CVAR\_READ, 580

- MPI\_T\_CVAR\_WRITE, <u>580</u>
- MPI\_T\_ENUM\_GET\_INFO, <u>574</u>, 574, 599

MPI_T_ENUM_GET_ITEM, 574, <u>575</u> , 599
MPI_T_FINALIZE, <u>573</u> , 573
MPI_T_INIT_THREAD, <u>572</u> , 572, 573
MPI_T_PVAR_GET_INDEX, <u>586</u> , 586, 599,
718
MPI_T_PVAR_GET_INFO, 574, 584, 585,
585, 588, 590, 591, 598, 599, 717, 718
MPI_T_PVAR_GET_NUM, <u>584</u> , 588
MPI_T_PVAR_HANDLE_ALLOC, 574, 588,
588, 590, 599
MPI_T_PVAR_HANDLE_FREE, <u>588</u> , 589,
599, 717
MPI_T_PVAR_READ, <u>590</u> , 590, 591, 599, 717
MPI_T_PVAR_READRESET, 586, 591, 591,
599, 717
MPI_T_PVAR_RESET, <u>591</u> , 591, 599, 717
MPI_T_PVAR_SESSION_CREATE, <u>587</u> , 599
MPI_T_PVAR_SESSION_FREE, <u>587</u> , 599
MPI_T_PVAR_START, 589, 599, 717
MPI_T_PVAR_STOP, <u>589</u> , 599, 717
MPI_T_PVAR_WRITE, <u>590</u> , 590, 599, 717
MPI_TEST, 34, 54, 55, <u>56</u> , 56, 57, 59, 60, 74,
79, 360, 481, 510, 511
MPI_TEST_CANCELLED, 54–56, <u>74</u> , 75, 479,
486, 511
486, 511 MPI_TESTALL, 59, <u>62</u> , 62, 479–481, 484, 487
486, 511 MPI_TESTALL, 59, <u>62</u> , 62, 479–481, 484, 487 MPI_TESTANY, 55, 59, <u>60</u> , 60, 64, 479–481,
$\begin{array}{c} 486,511\\ \text{MPI\_TESTALL},59,\underline{62},62,479-\!481,484,487\\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479-\!481,\\ 484,487\end{array}$
$\begin{array}{c} 486,511\\ \text{MPI\_TESTALL},59,\underline{62},62,479481,484,487\\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479481,\\ 484,487\\ \text{MPI\_TESTSOME},59,\underline{63},64,479481,484, \end{array}$
$\begin{array}{c} & 486,511 \\ \text{MPI\_TESTALL},59,\underline{62},62,479481,484,487 \\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479481,\\ & 484,487 \\ \\ \text{MPI\_TESTSOME},59,\underline{63},64,479481,484,\\ & 487 \\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI\_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI\_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI\_TOPO\_TEST},293,\underline{304},304\\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI\_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI\_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI\_TOPO\_TEST},293,\underline{304},304\\ \text{MPI\_TYPE\_C2F},\underline{674}\\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI\_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI\_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI\_TOPO\_TEST},293,\underline{304},304\\ \text{MPI\_TYPE\_C2F},\underline{674}\\ \text{MPI\_TYPE\_COMMIT},\underline{112},112,675\\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI\_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI\_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI\_TOPO\_TEST},293,\underline{304},304\\ \text{MPI\_TYPE\_C2F},\underline{674}\\ \text{MPI\_TYPE\_COMMIT},\underline{112},112,675\\ \text{MPI\_TYPE\_CONTIGUOUS},12,\underline{87},87,89,\\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI_TOPO_TEST},293,\underline{304},304\\ \text{MPI_TYPE_C2F},\underline{674}\\ \text{MPI_TYPE_C2F},\underline{674}\\ \text{MPI_TYPE_COMMIT},\underline{112},112,675\\ \text{MPI_TYPE_CONTIGUOUS},12,\underline{87},87,89,\\ 107,119,494,539\\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI_TOPO_TEST},293,\underline{304},304\\ \text{MPI_TYPE_C2F},\underline{674}\\ \text{MPI_TYPE_C2F},\underline{674}\\ \text{MPI_TYPE_CONTIGUOUS},12,\underline{87},87,89,\\ 107,119,494,539\\ \text{MPI_TYPE_CREATE_DARRAY},12,33,\underline{99}, \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI\_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI\_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI\_TOPO\_TEST},293,\underline{304},304\\ \text{MPI\_TYPE\_C2F},\underline{674}\\ \text{MPI\_TYPE\_COMMIT},\underline{112},112,675\\ \text{MPI\_TYPE\_CONTIGUOUS},12,\underline{87},87,89,\\ 107,119,494,539\\ \text{MPI\_TYPE\_CREATE\_DARRAY},12,33,\underline{99},\\ 99,119\\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI_TOPO_TEST},293,\underline{304},304\\ \text{MPI_TYPE_C2F},\underline{674}\\ \text{MPI_TYPE_CQF},\underline{674}\\ \text{MPI_TYPE_COMMIT},\underline{112},112,675\\ \text{MPI_TYPE_CONTIGUOUS},12,\underline{87},87,89,\\ 107,119,494,539\\ \text{MPI_TYPE_CREATE_DARRAY},12,33,\underline{99},\\ 99,119\\ \text{MPI_TYPE_CREATE_F90_COMPLEX},12,\\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI\_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI\_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI\_TOPO\_TEST},293,\underline{304},304\\ \text{MPI\_TYPE\_C2F},\underline{674}\\ \text{MPI\_TYPE\_C2F},\underline{674}\\ \text{MPI\_TYPE\_COMMIT},\underline{112},112,675\\ \text{MPI\_TYPE\_CONTIGUOUS},12,\underline{87},87,89,\\ 107,119,494,539\\ \text{MPI\_TYPE\_CREATE\_DARRAY},12,33,\underline{99},\\ 99,119\\ \text{MPI\_TYPE\_CREATE\_F90\_COMPLEX},12,\\ 119,121,179,541,626,\underline{645},647\\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI_TOPO_TEST},293,\underline{304},304\\ \text{MPI_TYPE_C2F},\underline{674}\\ \text{MPI_TYPE_C2F},\underline{674}\\ \text{MPI_TYPE_COMMIT},\underline{112},112,675\\ \text{MPI_TYPE_CONTIGUOUS},12,\underline{87},87,89,\\ 107,119,494,539\\ \text{MPI_TYPE_CREATE_DARRAY},12,33,\underline{99},\\ 99,119\\ \text{MPI_TYPE_CREATE_F90_COMPLEX},12,\\ 119,121,179,541,626,\underline{645},647\\ \text{MPI_TYPE_CREATE_F90_INTEGER},12,\\ \end{array}$
$\begin{array}{c} 486,511\\ \text{MPI\_TESTALL},59,\underline{62},62,479-481,484,487\\ \text{MPI\_TESTANY},55,59,\underline{60},60,64,479-481,\\ 484,487\\ \text{MPI\_TESTSOME},59,\underline{63},64,479-481,484,\\ 487\\ \text{MPI\_TOPO\_TEST},293,\underline{304},304\\ \text{MPI\_TYPE\_C2F},\underline{674}\\ \text{MPI\_TYPE\_C2F},\underline{674}\\ \text{MPI\_TYPE\_COMMIT},\underline{112},112,675\\ \text{MPI\_TYPE\_CONTIGUOUS},12,\underline{87},87,89,\\ 107,119,494,539\\ \text{MPI\_TYPE\_CREATE\_DARRAY},12,33,\underline{99},\\ 99,119\\ \text{MPI\_TYPE\_CREATE\_F90\_COMPLEX},12,\\ 119,121,179,541,626,\underline{645},647\\ \text{MPI\_TYPE\_CREATE\_F90\_INTEGER,12,\\ 119,121,178,541,626,\underline{645},647\\ \end{array}$
$\begin{array}{c} 486,511\\ \\ \text{MPI_TESTALL},59,\underline{62},62,479-481,484,487\\ \\ \text{MPI_TESTANY},55,59,\underline{60},60,64,479-481,\\ &484,487\\ \\ \text{MPI_TESTSOME},59,\underline{63},64,479-481,484,\\ &487\\ \\ \text{MPI_TOPO_TEST},293,\underline{304},304\\ \\ \text{MPI_TYPE_C2F},\underline{674}\\ \\ \text{MPI_TYPE_COMMIT},\underline{112},112,675\\ \\ \text{MPI_TYPE_CONTIGUOUS},12,\underline{87},87,89,\\ &107,119,494,539\\ \\ \text{MPI_TYPE_CREATE_DARRAY},12,33,99,\\ &99,119\\ \\ \text{MPI_TYPE_CREATE_F90_COMPLEX},12,\\ &119,121,179,541,626,\underline{645},647\\ \\ \text{MPI_TYPE_CREATE_F90_INTEGER},12,\\ &119,121,178,541,626,\underline{645},647\\ \\ \text{MPI_TYPE_CREATE_F90_REAL},12,119, \end{array}$
$\begin{array}{c} 486,511\\ \\ \text{MPI_TESTALL},59,\underline{62},62,479-481,484,487\\ \\ \text{MPI_TESTANY},55,59,\underline{60},60,64,479-481,\\ &484,487\\ \\ \text{MPI_TESTSOME},59,\underline{63},64,479-481,484,\\ &487\\ \\ \text{MPI_TOPO_TEST},293,\underline{304},304\\ \\ \text{MPI_TYPE_C2F},\underline{674}\\ \\ \text{MPI_TYPE_CONTIGUOUS},12,\underline{87},87,89,\\ &107,119,494,539\\ \\ \text{MPI_TYPE_CREATE_DARRAY},12,33,99,\\ &99,119\\ \\ \text{MPI_TYPE_CREATE_F90_COMPLEX},12,\\ &119,121,179,541,626,\underline{645},647\\ \\ \text{MPI_TYPE_CREATE_F90_INTEGER},12,\\ &119,121,178,541,626,\underline{645},647\\ \\ \text{MPI_TYPE_CREATE_F90_REAL},12,119,\\ &121,179,541,626,\underline{644},645-647,724\\ \end{array}$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL}, 59, \underline{62}, 62, 479-481, 484, 487 \\ \text{MPI_TESTANY}, 55, 59, \underline{60}, 60, 64, 479-481, \\ 484, 487 \\ \text{MPI_TESTSOME}, 59, \underline{63}, 64, 479-481, 484, \\ 487 \\ \text{MPI_TOPO_TEST}, 293, \underline{304}, 304 \\ \text{MPI_TYPE_C2F}, \underline{674} \\ \text{MPI_TYPE_COMMIT}, \underline{112}, 112, 675 \\ \text{MPI_TYPE_CONTIGUOUS}, 12, \underline{87}, 87, 89, \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY}, 12, 33, \underline{99}, \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX}, 12, \\ 119, 121, 179, 541, 626, \underline{645}, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER}, 12, \\ 119, 121, 178, 541, 626, \underline{645}, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL}, 12, 119, \\ 121, 179, 541, 626, \underline{644}, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED}, 12, 18, \\ \end{array}$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL}, 59, \underline{62}, 62, 479-481, 484, 487 \\ \text{MPI_TESTANY}, 55, 59, \underline{60}, 60, 64, 479-481, \\ 484, 487 \\ \text{MPI_TESTSOME}, 59, \underline{63}, 64, 479-481, 484, \\ 487 \\ \text{MPI_TOPO_TEST}, 293, \underline{304}, 304 \\ \text{MPI_TYPE_C2F}, \underline{674} \\ \text{MPI_TYPE_COMMIT}, \underline{112}, 112, 675 \\ \text{MPI_TYPE_CONTIGUOUS}, 12, \underline{87}, 87, 89, \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY}, 12, 33, \underline{99}, \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX}, 12, \\ 119, 121, 179, 541, 626, \underline{645}, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER}, 12, \\ 119, 121, 178, 541, 626, \underline{645}, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL}, 12, 119, \\ 121, 179, 541, 626, \underline{644}, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED}, 12, 18, \\ 87, \underline{92}, 92, 94, 96, 119, 623 \\ \end{array}$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL, 59, } \underline{62}, 62, 479-481, 484, 487 \\ \text{MPI_TESTANY, 55, 59, } \underline{60}, 60, 64, 479-481, \\ 484, 487 \\ \text{MPI_TESTSOME, 59, } \underline{63}, 64, 479-481, 484, \\ 487 \\ \text{MPI_TOPO_TEST, 293, } \underline{304}, 304 \\ \text{MPI_TYPE_C2F, } \underline{674} \\ \text{MPI_TYPE_COMMIT, } \underline{112}, 112, 675 \\ \text{MPI_TYPE_CONTIGUOUS, } 12, \underline{87}, 87, 89, \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY, } 12, 33, \underline{99}, \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX, } 12, \\ 119, 121, 179, 541, 626, \underline{645}, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER, } 12, \\ 119, 121, 178, 541, 626, \underline{645}, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL, } 12, 119, \\ 121, 179, 541, 626, \underline{644}, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED, } 12, 18, \\ 87, \underline{92}, 92, 94, 96, 119, 623 \\ \text{MPI_TYPE_CREATE_HINDEXED_BLOCK, \\ \end{array}$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL}, 59, \underline{62}, 62, 479-481, 484, 487 \\ \text{MPI_TESTANY}, 55, 59, \underline{60}, 60, 64, 479-481, \\ 484, 487 \\ \text{MPI_TESTSOME}, 59, \underline{63}, 64, 479-481, 484, \\ 487 \\ \text{MPI_TOPO_TEST}, 293, \underline{304}, 304 \\ \text{MPI_TYPE_C2F}, \underline{674} \\ \text{MPI_TYPE_COMMIT}, \underline{112}, 112, 675 \\ \text{MPI_TYPE_CONTIGUOUS}, 12, \underline{87}, 87, 89, \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY}, 12, 33, \underline{99}, \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX}, 12, \\ 119, 121, 179, 541, 626, \underline{645}, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER}, 12, \\ 119, 121, 178, 541, 626, \underline{645}, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL}, 12, 119, \\ 121, 179, 541, 626, \underline{644}, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED}, 12, 18, \\ 87, \underline{92}, 92, 94, 96, 119, 623 \\ \text{MPI_TYPE_CREATE_HINDEXED_BLOCK}, \\ 12, 87, \underline{94}, 94, 119, 720 \\ \end{array}$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL, 59, 62, 62, 479-481, 484, 487} \\ \text{MPI_TESTANY, 55, 59, 60, 60, 64, 479-481, } \\ 484, 487 \\ \text{MPI_TESTSOME, 59, 63, 64, 479-481, 484, } \\ 487 \\ \text{MPI_TOPO_TEST, 293, 304, 304} \\ \text{MPI_TYPE_C2F, 674} \\ \text{MPI_TYPE_COMMIT, 112, 112, 675} \\ \text{MPI_TYPE_CONTIGUOUS, 12, 87, 87, 89, } \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY, 12, 33, 99, } \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX, 12, } \\ 119, 121, 179, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER, 12, } \\ 119, 121, 178, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL, 12, 119, } \\ 121, 179, 541, 626, 644, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED, 12, 18, } \\ 87, 92, 92, 94, 96, 119, 623 \\ \text{MPI_TYPE_CREATE_HINDEXED_BLOCK, } \\ 12, 87, 94, 94, 119, 720 \\ \text{MPI_TYPE_CREATE_HVECTOR, 12, 18, } \\ \end{array}$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL, 59, 62, 62, 479-481, 484, 487} \\ \text{MPI_TESTANY, 55, 59, 60, 60, 64, 479-481, } \\ 484, 487 \\ \text{MPI_TESTSOME, 59, 63, 64, 479-481, 484, } \\ 487 \\ \text{MPI_TOPO_TEST, 293, 304, 304} \\ \text{MPI_TYPE_C2F, 674} \\ \text{MPI_TYPE_COMMIT, 112, 112, 675} \\ \text{MPI_TYPE_CONTIGUOUS, 12, 87, 87, 89, } \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY, 12, 33, 99, } \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX, 12, } \\ 119, 121, 179, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER, 12, } \\ 119, 121, 178, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL, 12, 119, } \\ 121, 179, 541, 626, 644, 645-647, 724 \\ \text{MPI_TYPE_CREATE_F90_REAL, 12, 119, } \\ 121, 179, 541, 626, 644, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED, 12, 18, } \\ 87, 92, 92, 94, 96, 119, 623 \\ \text{MPI_TYPE_CREATE_HVECTOR, 12, 18, } \\ 87, 89, 89, 119, 623 \\ \end{array}$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL, 59, 62, 62, 479-481, 484, 487} \\ \text{MPI_TESTANY, 55, 59, 60, 60, 64, 479-481, } \\ 484, 487 \\ \text{MPI_TESTSOME, 59, 63, 64, 479-481, 484, } \\ 487 \\ \text{MPI_TOPO_TEST, 293, 304, 304} \\ \text{MPI_TYPE_C2F, 674} \\ \text{MPI_TYPE_COMMIT, 112, 112, 675} \\ \text{MPI_TYPE_CONTIGUOUS, 12, 87, 87, 89, } \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY, 12, 33, 99, } \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX, 12, } \\ 119, 121, 179, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER, 12, } \\ 119, 121, 178, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL, 12, 119, } \\ 121, 179, 541, 626, 644, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED, 12, 18, } \\ 87, 92, 92, 94, 96, 119, 623 \\ \text{MPI_TYPE_CREATE_HINDEXED_BLOCK, } \\ 12, 87, 94, 94, 119, 720 \\ \text{MPI_TYPE_CREATE_INDEXED_BLOCK, } \\ 12, 87, 99, 89, 119, 623 \\ \text{MPI_TYPE_CREATE_INDEXED_BLOCK, } \\ \text{MPI_TYPE_CREATE_INDEXED_BLOCK, } \\ 12, 79E_CREATE_INDEXED_BLOCK, \\ 1300000000000000000000000000000000000$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL, 59, 62, 62, 479-481, 484, 487} \\ \text{MPI_TESTANY, 55, 59, 60, 60, 64, 479-481, } \\ 484, 487 \\ \text{MPI_TESTSOME, 59, 63, 64, 479-481, 484, } \\ 487 \\ \text{MPI_TOPO_TEST, 293, 304, 304} \\ \text{MPI_TYPE_C2F, 674} \\ \text{MPI_TYPE_CONTIGUOUS, 12, 87, 87, 89, } \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY, 12, 33, 99, } \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX, 12, } \\ 119, 121, 179, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER, 12, } \\ 119, 121, 178, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL, 12, 119, } \\ 121, 179, 541, 626, 644, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED, 12, 18, } \\ 87, 92, 92, 94, 96, 119, 623 \\ \text{MPI_TYPE_CREATE_HINDEXED_BLOCK, } \\ 12, 87, 94, 94, 119, 623 \\ \text{MPI_TYPE_CREATE_INDEXED_BLOCK, } \\ 12, 93, 94, 119 \\ \end{array}$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL, 59, 62, 62, 479-481, 484, 487} \\ \text{MPI_TESTANY, 55, 59, 60, 60, 64, 479-481, \\ 484, 487 \\ \text{MPI_TESTSOME, 59, 63, 64, 479-481, 484, \\ 487 \\ \text{MPI_TOPO_TEST, 293, 304, 304} \\ \text{MPI_TYPE_C2F, 674} \\ \text{MPI_TYPE_COMMIT, 112, 112, 675} \\ \text{MPI_TYPE_CONTIGUOUS, 12, 87, 87, 89, \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY, 12, 33, 99, \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX, 12, \\ 119, 121, 179, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER, 12, \\ 119, 121, 178, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL, 12, 119, \\ 121, 179, 541, 626, 644, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED, 12, 18, \\ 87, 92, 92, 94, 96, 119, 623 \\ \text{MPI_TYPE_CREATE_HINDEXED_BLOCK, \\ 12, 87, 94, 94, 119, 720 \\ \text{MPI_TYPE_CREATE_INDEXED_BLOCK, \\ 12, 93, 94, 119 \\ \text{MPI_TYPE_CREATE_INDEXED_BLOCK, \\ 12, 93, 94, 119 \\ \text{MPI_TYPE_CREATE_KEYVAL, 268, 278, \\ \end{array}$
$\begin{array}{c} 486, 511 \\ \text{MPI_TESTALL, 59, 62, 62, 479-481, 484, 487} \\ \text{MPI_TESTANY, 55, 59, 60, 60, 64, 479-481, } \\ 484, 487 \\ \text{MPI_TESTSOME, 59, 63, 64, 479-481, 484, } \\ 487 \\ \text{MPI_TOPO_TEST, 293, 304, 304} \\ \text{MPI_TYPE_C2F, 674} \\ \text{MPI_TYPE_CONTIGUOUS, 12, 87, 87, 89, } \\ 107, 119, 494, 539 \\ \text{MPI_TYPE_CREATE_DARRAY, 12, 33, 99, } \\ 99, 119 \\ \text{MPI_TYPE_CREATE_F90_COMPLEX, 12, } \\ 119, 121, 179, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_INTEGER, 12, } \\ 119, 121, 178, 541, 626, 645, 647 \\ \text{MPI_TYPE_CREATE_F90_REAL, 12, 119, } \\ 121, 179, 541, 626, 644, 645-647, 724 \\ \text{MPI_TYPE_CREATE_HINDEXED, 12, 18, } \\ 87, 92, 92, 94, 96, 119, 623 \\ \text{MPI_TYPE_CREATE_HINDEXED_BLOCK, } \\ 12, 87, 94, 94, 119, 623 \\ \text{MPI_TYPE_CREATE_INDEXED_BLOCK, } \\ 12, 93, 94, 119 \\ \end{array}$

109, 110, 119, 539, 624, 722	1
MPI_TYPE_CREATE_STRUCT, 12, 18, 87,	2
94, <u>95</u> , 95, 96, 107, 119, 175, 623	
MPI_TYPE_CREATE_SUBARRAY, 12, 15,	3
<u>96</u> , 98, 100, 119	4
MPI_TYPE_DELETE_ATTR, 268, 280, 281,	5
$\begin{array}{c} \text{MP1_1 I FE_DELETE_ATTR, 208, } \underline{200}, 281, \\ 722 \end{array}$	6
122 MDI TVDE DIID 19 112 113 110 799	7
MPI_TYPE_DUP, 12, <u>113</u> , 113, 119, 722 MPI_TYPE_DUP_FN, <u>278</u> , 278, 697, 717 MPI_TYPE_EXTENT, 18, 623, 719 MPI_TYPE_F2C, <u>674</u> MPI_TYPE_FREE, <u>112</u> , 121, 279, 360 MPI_TYPE_FREE_KEYVAL, 268, <u>279</u> , 281	8
$MFI_1FE_DUF_FN, 270, 270, 097, 717$ $MDI_TVDE_EVTENT_18, 692, 710$	9
$MPI_1 PE_EATENT, 10, 023, 719$ $MDI_TVDE_E2C_674$	
$MPI_I YPE_F 2O, 074$ $MDI_T YDE_E DEE_{110} 101, 070, 000$	10
$MPI_1YPE_FREE, 112, 121, 279, 300$	11
MPI_1YPE_FREE_KEYVAL, 208, <u>279</u> , 281 MPI_TYPE_GET_ATTR, 268, <u>280</u> , 281, 634,	12
$MPI_1 YPE_GEI_AIIR, 268, 280, 281, 634, COL 500$	13
681, 722	14
MPI_TYPE_GET_CONTENTS, 118, 119,	15
<u>120</u> , 121, 122	16
MPI_TYPE_GET_ENVELOPE, <u>118</u> , 118,	
120, 121, 646	17
MPI_TYPE_GET_EXTENT, 18, <u>108</u> , 111,	18
623, 649, 679	19
MPI_TYPE_GET_EXTENT_X, <u>109</u> , 719	20
MPI_TYPE_GET_NAME, <u>286</u> , 723	21
MPI_TYPE_GET_TRUE_EXTENT, <u>110</u> , 110	22
MPI_TYPE_GET_TRUE_EXTENT_X, 110,	23
$\underline{111}, 719$	
MPI_TYPE_HINDEXED, 18, 623, 719	24
MPI_TYPE_HVECTOR, 18, 623, 719	25
MPI_TYPE_INDEXED, 12, 90, <u>91</u> , 91–93, 119	26
MPI_TYPE_LB, 18, 623, 719	27
MPI_TYPE_MATCH_SIZE, 626, <u>649</u> , 649, 723	28
MPI_TYPE_NULL_COPY_FN, <u>278</u> , 278, 697,	29
717	30
MPI_TYPE_NULL_DELETE_FN, <u>278</u> , 697,	
717, 723	31
MPI_TYPE_SET_ATTR, 268, <u>280</u> , 281, 634,	32
681,685,722	33
MPI_TYPE_SET_NAME, <u>286</u> , 722	34
MPI_TYPE_SIZE, <u>106</u> , 106, 566, 720	35
MPI_TYPE_SIZE_X, <u>106</u> , 106, 719	36
MPI_TYPE_STRUCT, 18, 623, 719	37
MPI_TYPE_UB, 18, 623, 719	
MPI_TYPE_VECTOR, 12, <u>88</u> , 88, 89, 92, 119	38
MPI_UNPACK, <u>135</u> , 135, 136, 140, 544	39
MPI_UNPACK_EXTERNAL, 8, <u>141</u> , 647	40
MPI_UNPUBLISH_NAME, 351, <u>393</u> , 393	41
MPI_WAIT, 32, 34, 54, <u>55</u> , 55–58, 60, 61, 74,	42
79, 199, 221, 360, 477, 481, 488, 510,	43
511, 530, 547, 548, 654, 660, 661, 664	44
MPI_WAITALL, 59, <u>61</u> , 61, 62, 199, 222, 432,	
479-481, 484, 487, 488	45
MPI_WAITANY, 43, 55, <u>59</u> , 59, 60, 64,	46
479-481,  484,  487,  488	47
	48

1	MPI_WAITSOME, 59, <u>62</u> , 63–65, 479–481,
2	484, 487, 488
3	MPI_WIN_ALLOCATE, 404, <u>407</u> , 408, 410,
4	415, 417, 421, 450, 638, 640, 716
5	MPI_WIN_ALLOCATE_CPTR, 408, 716
6	$MPI_WIN\_ALLOCATE\_SHARED, 404, \underline{409},$
	409, 411, 415, 417, 640, 716
7	MPI_WIN_ALLOCATE_SHARED_CPTR,
8	410, 716
9	MPI_WIN_ATTACH, 412, <u>413</u> , 413–415, 450
10	MPI_WIN_C2F, <u>675</u>
11	MPI_WIN_CALL_ERRHANDLER, <u>355</u> , 356
12	MPI_WIN_COMPLETE, 415, 439, <u>444</u> , 444–447, 455, 461
13	MPI_WIN_CREATE, 404, <u>405</u> , 407, 408, 410,
14	413–415, 417, 454, 488
15	MPI_WIN_CREATE_DYNAMIC, 352, 404,
16	<u>412</u> , 412–415, 417, 454
17	MPI_WIN_CREATE_ERRHANDLER, 343,
18	<u>346,</u> 347, 706, 708, 723
19	MPI_WIN_CREATE_KEYVAL, 268, <u>274</u> , 281,
20	681,705,707,727
21	MPI_WIN_DELETE_ATTR, 268, <u>277</u> , 281
22	MPI_WIN_DETACH, 412, <u>414</u> , 414, 415
23	MPI_WIN_DUP_FN, <u>275</u> , 275, 697, 717
24	MPI_WIN_F2C, <u>675</u>
	MPI_WIN_FENCE, 415, 423, 439, <u>442</u> , 443,
25	452, 453, 455, 456, 459, 464, 665 MPI_WIN_FLUSH, 410, 432, 433, <u>450</u> , 451,
26	$\begin{array}{c} 1111 \\ 1200 \\ 11$
27	MPI_WIN_FLUSH_ALL, 432, 433, 451, 455
28	MPI_WIN_FLUSH_LOCAL, 432, 451, 455,
29	606
30	$MPI_WIN_FLUSH_LOCAL_ALL, 432, \underline{451},$
31	452,  455,  606
32	MPI_WIN_FREE, 276, 360, 400, <u>415</u> , 415,
33	416, 606
34	MPI_WIN_FREE_KEYVAL, 268, <u>276</u> , 281
35	MPI_WIN_GET_ATTR, 268, <u>277</u> , 281, 416, 681, 685
36	MPI_WIN_GET_ERRHANDLER, 343, 347,
37	728
38	MPI_WIN_GET_FAILED, 612, 614
39	MPI_WIN_GET_GROUP, $417$ , 417
40	MPI_WIN_GET_INFO, <u>418</u> , 418, 720
41	MPI_WIN_GET_NAME, <u>287</u>
42	MPI_WIN_LOCK, 406, 440, <u>447</u> , 448–450,
43	452, 453, 455, 459–461
44	MPI_WIN_LOCK_ALL, 406, 440, <u>448</u> , 448,
45	449, 452, 453, 455, 461, 469
46	MPI_WIN_NULL_COPY_FN, <u>275</u> , 275, 697,
47	717 MPI_WIN_NULL_DELETE_FN, 275, 697,
48	717
	1 ÷ 1

MPI_WIN_POST, 415, 439, 444, <u>445</u> , 445–447,
449, 452, 453, 455, 462, 464
MPI_WIN_REVOKE, 606, <u>611</u> , 611, 612
MPI_WIN_SET_ATTR, 268, <u>276</u> , 281, 416,
634,681,685
MPI_WIN_SET_ERRHANDLER, 343, <u>346</u>
MPI_WIN_SET_INFO, 417, <u>418</u> , 418, 720
MPI_WIN_SET_NAME, <u>286</u>
$MPI_WIN_SHARED_QUERY, 409, \underline{411}, 640,$
716
MPI_WIN_SHARED_QUERY_CPTR, 412,
716
MPI_WIN_START, 439, <u>443</u> , 444, 445, 447,
452, 453, 462, 469
MPI_WIN_SYNC, <u>452</u> , 452, 455, 457, 458,
461,  469 – 471
MPI_WIN_TEST, <u>446</u> , 446
MPI_WIN_UNLOCK, 433, 440, <u>448</u> , 450, 455,
459,  460
MPI_WIN_UNLOCK_ALL, 433, 440, <u>449</u> ,
455, 459, 469
MPI_WIN_WAIT, 415, 439, <u>445</u> , 445–447, 449,
455, 459, 461, 462
MPI_WTICK, 20, <u>357</u> , 357
MPI_WTIME, 20, 338, <u>357</u> , 357, 566, 584
mpiexec, 358, 363, 364, <u>365</u> , 490
mpirun, 364
PMPI_, 563, 634
$1 \text{ WH} 1_{-}, 003, 034$

PMPI\_, 505, 654 PMPI\_AINT\_ADD, 20 PMPI\_AINT\_DIFF, 20 PMPI\_ISEND, 634, 637 PMPI\_WTICK, 20 PMPI\_WTIME, 20