MPI: A Message-Passing Interface Standard Version 4.0

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

November 26, 2019

1	
	This document describes a draft version of the Message-Passing Interface (MPI) stan-
2	dard, version 4.0, intended for comment. It is not an official version of the standard. The
3	MPI standard includes point-to-point message-passing, collective communications, group
4	and communicator concepts, process topologies, environmental management, process cre-
5	ation and management, one-sided communications, extended collective operations, external
6	interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for
7	C and Fortran are defined.
8	Historically, the evolution of the standards is from MPI-1.0 (May 5, 1994) to MPI-1.1
9	(June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and
10	published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality,
11	to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2
12	and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008),
13	combining the previous documents. Version MPI-2.2 (September 4, 2009) added additional
14	clarifications and seven new routines. Version MPI-3.0 (September 21, 2012) is an extension
15	of MPI-2.2. Version MPI-3.1 (June 4, 2015) adds clarifications and minor extensions to
16	MPI-3.0.
17	
18	Comments . Please send comments on MPI to the MPI Forum as follows:
19	
20	1. Subscribe to http://lists.mpi-forum.org/mailman/listinfo.cgi/mpi-comments
21	2. Can de anno anno ante tra comi a companya frances and transforme and the transformer state that the LUDI of
22	2. Send your comment to: mpi-comments@mpi-forum.org, together with the URL of
23	the version of the MPI standard and the page and line numbers on which you are
24	commenting.
25	Your comment will be forwarded to MPI Forum committee members for consideration.
26	Messages sent from an unsubscribed e-mail address will not be considered.
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	© 1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015, 2019 University of Tennessee,
46	Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted,
47	provided the University of Tennessee copyright notice and the title of this document appear,
48	and notice is given that copying is by permission of the University of Tennessee.
	and neares is Siven and copying is by permission of the emission of remission.

Version 4.0: XXXX 2020.

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.

 $\mathbf{5}$

 $\overline{7}$

 31

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

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

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

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

Contents

A	cknov	wledgments	ix
1	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	Background of MPI-4.0	5
	1.9	Who Should Use This Standard?	5
	1.10	What Platforms Are Targets for Implementation?	5
	1.11	What Is Included in the Standard?	5
	1.12	What Is Not Included in the Standard?	6
	1.13	Organization of This Document	6
2	MPI	Terms and Conventions	9
	2.1	Document Notation	9
	2.2	Naming Conventions	9
	2.3	Procedure Specification	10
	2.4	Semantic Terms	11
	2.5	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 Interfaces	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

	2.8	Error]	$Handling \dots \dots$
	2.9	Impler	nentation Issues
		2.9.1	Independence of Basic Runtime Routines
		2.9.2	Interaction with Signals
	2.10	Examp	bles \ldots 23
3	Poir	nt-to-P	oint Communication 25
	3.1	Introd	$uction \dots \dots$
	3.2	Blocki	ng Send and Receive Operations
		3.2.1	Blocking Send
		3.2.2	Message Data
		3.2.3	Message Envelope 29
		3.2.4	Blocking Receive
		3.2.5	Return Status
		3.2.6	Passing MPI_STATUS_IGNORE for Status
	3.3		Type Matching and Data Conversion 35
	0.0	3.3.1	Type Matching Rules
		0.0.1	Type MPI_CHARACTER
		3.3.2	Data Conversion 38
	3.4	0.0	unication Modes
			tics of Point-to-Point Communication
	3.5		
	3.6		Allocation and Usage
	0.7	3.6.1	Model Implementation of Buffered Mode
	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 59
		3.7.5	Multiple Completions 60
		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	ent Communication Requests
	3.10	Send-F	Receive
	3.11	Null P	rocesses
4	Dat	atypes	
	4.1	Derive	d Datatypes
		4.1.1	Type Constructors with Explicit Addresses 89
		4.1.2	Datatype Constructors 89
		4.1.3	Subarray Datatype Constructor 99
		4.1.4	Distributed Array Datatype Constructor
		4.1.5	Address and Size Functions
		4.1.6	Lower-Bound and Upper-Bound Markers 110
		4.1.7	Extent and Bounds of Datatypes
			· ·

		4.1.8	True Extent of Datatypes	114
		4.1.9	Commit and Free	115
		4.1.10	Duplicating a Datatype	117
		4.1.11	Use of General Datatypes in Communication	117
		4.1.12	Correct Use of Addresses	121
		4.1.13	Decoding a Datatype	122
		4.1.14	Examples	129
	4.2	Pack a	und Unpack	138
	4.3	Canon	ical MPI_PACK and MPI_UNPACK	144
5	Coll		Communication	149
	5.1		uction and Overview	
	5.2	Comm	unicator Argument	
		5.2.1	Specifics for Intracommunicator Collective Operations	
		5.2.2	Applying Collective Operations to Intercommunicators	
		5.2.3	Specifics for Intercommunicator Collective Operations	
	5.3	Barrie	r Synchronization	155
	5.4	Broad	cast	
		5.4.1	Example using MPI_BCAST	
	5.5	Gather	r	157
		5.5.1	Examples using MPI_GATHER, MPI_GATHERV	160
	5.6	Scatte	r	
		5.6.1	Examples using MPI_SCATTER, MPI_SCATTERV	170
	5.7	Gathe	r-to-all	
		5.7.1	Example using MPI_ALLGATHER	175
	5.8	All-to-	All Scatter/Gather	176
	5.9	Global	l Reduction Operations	181
		5.9.1	Reduce	
		5.9.2	Predefined Reduction Operations	184
		5.9.3	Signed Characters and Reductions	
		5.9.4	MINLOC and MAXLOC	187
		5.9.5	User-Defined Reduction Operations	191
			Example of User-defined Reduce	194
		5.9.6	All-Reduce	195
		5.9.7	Process-Local Reduction	197
	5.10		e-Scatter	
			MPI_REDUCE_SCATTER_BLOCK	
		5.10.2	MPI_REDUCE_SCATTER	200
	5.11	Scan		202
		5.11.1	Inclusive Scan	202
		5.11.2	Exclusive Scan	203
		5.11.3	Example using MPI_SCAN	204
	5.12	Nonble	ocking Collective Operations	205
		5.12.1	Nonblocking Barrier Synchronization	207
		5.12.2	Nonblocking Broadcast	
			Example using MPI_IBCAST	209
			Nonblocking Gather	
		5.12.4	Nonblocking Scatter	211

		5.12.5	Nonblocking Gather-to-all	213
		5.12.6	Nonblocking All-to-All Scatter/Gather	215
		5.12.7	Nonblocking Reduce	218
		5.12.8	Nonblocking All-Reduce	219
		5.12.9	Nonblocking Reduce-Scatter with Equal Blocks	220
			Nonblocking Reduce-Scatter	221
			Nonblocking Inclusive Scan	222
			Nonblocking Exclusive Scan	223
	5.13		ent Collective Operations	223
			Persistent Barrier Synchronization	225
			Persistent Broadcast	225
			Persistent Gather	226
			Persistent Scatter	229
			Persistent Gather-to-all	231
			Persistent All-to-All Scatter/Gather	233
			Persistent Reduce	236
			Persistent All-Reduce	230 237
			Persistent Reduce-Scatter with Equal Blocks	238
			Persistent Reduce-Scatter	239
			Persistent Inclusive Scan	233 240
			Persistent Exclusive Scan	240 241
	5 14		tness	$241 \\ 241$
	0.14	Correc	tiless	241
6	Gro	ups, C	ontexts, Communicators, and Caching	251
	6.1	Introd	uction	251
	6.1	Introdu 6.1.1	uction	$251 \\ 251$
	6.1		Features Needed to Support Libraries	251
	6.1 6.2	$6.1.1 \\ 6.1.2$	Features Needed to Support Libraries	$\begin{array}{c} 251 \\ 252 \end{array}$
	-	6.1.1 6.1.2 Basic (Features Needed to Support Libraries	$251 \\ 252 \\ 254$
	-	6.1.1 6.1.2 Basic (6.2.1	Features Needed to Support Libraries	251 252 254 254
	-	6.1.1 6.1.2 Basic (6.2.1 6.2.2	Features Needed to Support Libraries MPI's Support for Libraries Concepts Groups Contexts Groups	251 252 254 254 254
	-	6.1.1 6.1.2 Basic (6.2.1 6.2.2 6.2.3	Features Needed to Support Libraries	251 252 254 254 254 254 255
	6.2	6.1.1 6.1.2 Basic 6 6.2.1 6.2.2 6.2.3 6.2.4	Features Needed to Support Libraries MPI's Support for Libraries MPI's Support for Libraries Groups Concepts Groups Groups Groups Intra-Communicators Groups Predefined Intra-Communicators Groups	251 252 254 254 254 255 255
	-	6.1.1 6.1.2 Basic (6.2.1 6.2.2 6.2.3 6.2.4 Group	Features Needed to Support Libraries MPI's Support for Libraries MPI's Support for Libraries Groups Concepts Concepts Groups Groups Contexts Contexts Intra-Communicators Contexts Predefined Intra-Communicators Contexts Management Contexts	251 252 254 254 254 255 255 256
	6.2	6.1.1 6.1.2 Basic (6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1	Features Needed to Support Libraries	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \end{array}$
	6.2	6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup Constructors	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 255 \\ 256 \\ 256 \\ 258 \end{array}$
	6.26.3	6.1.1 6.1.2 Basic (6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3	Features Needed to Support Libraries	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 258 \\ 264 \end{array}$
	6.2	6.1.1 6.1.2 Basic C 6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3 Comm	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup Destructorsunicator Management	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 258 \\ 264 \\ 264 \end{array}$
	6.26.3	6.1.1 6.1.2 Basic 0 6.2.1 6.2.2 6.2.3 6.2.4 Group 6.3.1 6.3.2 6.3.3 Comm 6.4.1	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup ConstructorsGroup Destructorsunicator ManagementCommunicator Accessors	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 264 \\ 264 \\ 264 \end{array}$
	6.26.3	$\begin{array}{c} 6.1.1 \\ 6.1.2 \\ Basic \\ 6.2.1 \\ 6.2.2 \\ 6.2.3 \\ 6.2.4 \\ Group \\ 6.3.1 \\ 6.3.2 \\ 6.3.3 \\ Comm \\ 6.4.1 \\ 6.4.2 \end{array}$	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup DestructorsUnicator ManagementCommunicator AccessorsCommunicator Constructors	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 264 \\ 264 \\ 264 \\ 266 \end{array}$
	6.26.3	$\begin{array}{c} 6.1.1 \\ 6.1.2 \\ Basic \\ 6.2.1 \\ 6.2.2 \\ 6.2.3 \\ 6.2.4 \\ Group \\ 6.3.1 \\ 6.3.2 \\ 6.3.3 \\ Comm \\ 6.4.1 \\ 6.4.2 \\ 6.4.3 \end{array}$	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup DestructorsGroup Destructorsunicator ManagementCommunicator AccessorsCommunicator ConstructorsCommunicator Destructors	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 264 \\ 264 \\ 264 \\ 266 \\ 278 \end{array}$
	6.26.36.4	$\begin{array}{c} 6.1.1 \\ 6.1.2 \\ Basic \\ 6.2.1 \\ 6.2.2 \\ 6.2.3 \\ 6.2.4 \\ Group \\ 6.3.1 \\ 6.3.2 \\ 6.3.3 \\ Comm \\ 6.4.1 \\ 6.4.2 \\ 6.4.3 \\ 6.4.4 \end{array}$	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup DestructorsGroup Destructorsunicator ManagementCommunicator AccessorsCommunicator ConstructorsCommunicator Info	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 264 \\ 264 \\ 264 \\ 264 \\ 266 \\ 278 \\ 279 \end{array}$
	6.26.3	$\begin{array}{c} 6.1.1 \\ 6.1.2 \\ Basic \\ 6.2.1 \\ 6.2.2 \\ 6.2.3 \\ 6.2.4 \\ Group \\ 6.3.1 \\ 6.3.2 \\ 6.3.3 \\ Comm \\ 6.4.1 \\ 6.4.2 \\ 6.4.3 \\ 6.4.4 \\ Motive \end{array}$	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup ConstructorsGroup Destructorsunicator ManagementCommunicator AccessorsCommunicator ConstructorsCommunicator DestructorsCommunicator InfoCommunicator Info	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 258 \\ 264 \\ 264 \\ 264 \\ 264 \\ 266 \\ 278 \\ 279 \\ 281 \end{array}$
	6.26.36.4	$\begin{array}{c} 6.1.1 \\ 6.1.2 \\ Basic \\ 6.2.1 \\ 6.2.2 \\ 6.2.3 \\ 6.2.4 \\ Group \\ 6.3.1 \\ 6.3.2 \\ 6.3.3 \\ Comm \\ 6.4.1 \\ 6.4.2 \\ 6.4.3 \\ 6.4.4 \\ Motiva \\ 6.5.1 \end{array}$	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup ConstructorsGroup Destructorsunicator ManagementCommunicator AccessorsCommunicator ConstructorsCommunicator DestructorsCommunicator InfoCommunicator InfoCurrent Practice #1	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 264 \\ 264 \\ 264 \\ 266 \\ 278 \\ 279 \\ 281 \\ 281 \end{array}$
	6.26.36.4	$\begin{array}{c} 6.1.1 \\ 6.1.2 \\ Basic \\ 6.2.1 \\ 6.2.2 \\ 6.2.3 \\ 6.2.4 \\ Group \\ 6.3.1 \\ 6.3.2 \\ 6.3.3 \\ Comm \\ 6.4.1 \\ 6.4.2 \\ 6.4.3 \\ 6.4.4 \\ Motiva \\ 6.5.1 \\ 6.5.2 \end{array}$	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup ConstructorsGroup DestructorsUnicator ManagementCommunicator AccessorsCommunicator ConstructorsCommunicator DestructorsCommunicator InfoCurrent Practice #1Current Practice #2	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 264 \\ 264 \\ 264 \\ 264 \\ 264 \\ 266 \\ 278 \\ 279 \\ 281 \\ 281 \\ 282 \end{array}$
	6.26.36.4	$\begin{array}{c} 6.1.1 \\ 6.1.2 \\ Basic \\ 6.2.1 \\ 6.2.2 \\ 6.2.3 \\ 6.2.4 \\ Group \\ 6.3.1 \\ 6.3.2 \\ 6.3.3 \\ Comm \\ 6.4.1 \\ 6.4.2 \\ 6.4.3 \\ 6.4.4 \\ Motive \\ 6.5.1 \\ 6.5.2 \\ 6.5.3 \end{array}$	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup DestructorsGroup DestructorsUnicator ManagementCommunicator AccessorsCommunicator ConstructorsCommunicator DestructorsCommunicator InfoCommunicator InfoCurrent Practice #1Current Practice #2(Approximate) Current Practice #3	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 258 \\ 264 \\ 264 \\ 264 \\ 264 \\ 266 \\ 278 \\ 279 \\ 281 \\ 281 \\ 282 \\ 283 \end{array}$
	6.26.36.4	$\begin{array}{c} 6.1.1 \\ 6.1.2 \\ Basic (0 \\ 6.2.1 \\ 6.2.2 \\ 6.2.3 \\ 6.2.4 \\ Group \\ 6.3.1 \\ 6.3.2 \\ 6.3.3 \\ Comm \\ 6.4.1 \\ 6.4.2 \\ 6.4.3 \\ 6.4.4 \\ Motiva \\ 6.5.1 \\ 6.5.2 \\ 6.5.3 \\ 6.5.4 \end{array}$	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup ConstructorsGroup Destructorsunicator ManagementCommunicator AccessorsCommunicator ConstructorsCommunicator DestructorsCommunicator Predefined InfoCommunicator ConstructorsGroup ConstructorsCommunicator AccessorsCommunicator PestructorsCommunicator InfoCurrent Practice #1Current Practice #2(Approximate) Current Practice #3Example #4	$\begin{array}{c} 251\\ 252\\ 254\\ 254\\ 255\\ 255\\ 256\\ 256\\ 256\\ 256\\ 264\\ 264\\ 264\\ 266\\ 278\\ 279\\ 281\\ 281\\ 281\\ 282\\ 283\\ 284 \end{array}$
	6.26.36.4	$\begin{array}{c} 6.1.1 \\ 6.1.2 \\ Basic (0 \\ 6.2.1 \\ 6.2.2 \\ 6.2.3 \\ 6.2.4 \\ Group \\ 6.3.1 \\ 6.3.2 \\ 6.3.3 \\ Comm \\ 6.4.1 \\ 6.4.2 \\ 6.4.3 \\ 6.4.4 \\ Motive \\ 6.5.1 \\ 6.5.2 \\ 6.5.3 \end{array}$	Features Needed to Support LibrariesMPI's Support for LibrariesConceptsGroupsContextsIntra-CommunicatorsPredefined Intra-CommunicatorsManagementGroup AccessorsGroup DestructorsGroup DestructorsUnicator ManagementCommunicator AccessorsCommunicator ConstructorsCommunicator DestructorsCommunicator InfoCommunicator InfoCurrent Practice #1Current Practice #2(Approximate) Current Practice #3	$\begin{array}{c} 251 \\ 252 \\ 254 \\ 254 \\ 255 \\ 255 \\ 256 \\ 256 \\ 256 \\ 258 \\ 264 \\ 264 \\ 264 \\ 264 \\ 266 \\ 278 \\ 279 \\ 281 \\ 281 \\ 282 \\ 283 \end{array}$

	6.6	Inter-Communication	 	 	 •	 	288
		6.6.1 Inter-communicator Accessors	 	 		 	290
		6.6.2 Inter-communicator Operations	 	 		 	292
		6.6.3 Inter-Communication Examples	 	 		 	294
		Example 1: Three-Group "Pipeline"	 	 		 	294
		Example 2: Three-Group "Ring"	 	 		 	296
	6.7	Caching	 	 		 	297
		6.7.1 Functionality	 	 		 	298
		6.7.2 Communicators	 	 		 	299
		6.7.3 Windows	 	 		 . :	304
		6.7.4 Datatypes	 	 		 . ;	308
		6.7.5 Error Class for Invalid Keyval	 	 		 . ;	311
		6.7.6 Attributes Example	 	 		 . ;	311
	6.8	Naming Objects	 	 		 . :	313
	6.9	Formalizing the Loosely Synchronous Model.	 	 		 . :	318
		6.9.1 Basic Statements	 	 		 . ;	318
		6.9.2 Models of Execution	 	 		 . :	318
		Static Communicator Allocation	 	 		 . :	319
		Dynamic Communicator Allocation .	 	 		 . ;	319
		The General Case	 	 		 . ;	319
7		ocess Topologies					321
	7.1	Introduction					321
	7.2	1 0					322
	7.3	0					322
	7.4						322
	7.5	Topology Constructors					324
		7.5.1 Cartesian Constructor					324
		7.5.2 Cartesian Convenience Function: MPI					325
		7.5.3 Graph Constructor					326
		7.5.4 Distributed Graph Constructor					328
		7.5.5 Topology Inquiry Functions					335
		7.5.6 Cartesian Shift Coordinates					343
		7.5.7 Partitioning of Cartesian Structures .					345
		7.5.8 Low-Level Topology Functions					346
	7.6	Neighborhood Collective Communication					348
		7.6.1 Neighborhood Gather					349
		7.6.2 Neighbor Alltoall					352
	7.7	Nonblocking Neighborhood Communication .					357
		7.7.1 Nonblocking Neighborhood Gather					358
		7.7.2 Nonblocking Neighborhood Alltoall					360
	7.8	Persistent Neighborhood Communication					363
		7.8.1 Persistent Neighborhood Gather					363
		7.8.2 Persistent Neighborhood Alltoall					365
	7.9	An Application Example	 	 		 . :	369

8	MPI	Envir	onmental Management	375
	8.1	Impler	nentation Information	375
		8.1.1	Version Inquiries	375
		8.1.2	Environmental Inquiries	377
			Tag Values	377
			Host Rank	377
			IO Rank	377
			Clock Synchronization	378
			Inquire Processor Name	
	8.2	Memo	ry Allocation	
	8.3		Handling	382
	0.0	8.3.1	Error Handlers for Communicators	384
		8.3.2	Error Handlers for Windows	386
		8.3.3	Error Handlers for Files	388
		8.3.4	Freeing Errorhandlers and Retrieving Error Strings	3 90
	8.4		Codes and Classes	$390 \\ 391$
	8.5		Classes, Error Codes, and Error Handlers	
	8.6		s and Synchronization \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots	398 200
	8.7	Startu		399
		8.7.1	Allowing User Functions at Process Termination	405
	0.0	8.7.2	Determining Whether MPI Has Finished	
	8.8	Portat	ble MPI Process Startup	406
9	The	Info C	Dbject	409
10	Pro	cess C	reation and Management	415
	10.1	Introd	uction	415
			uction	
		The D	ynamic Process Model	416
		The D 10.2.1	ynamic Process Model	416
	10.2	The D 10.2.1 10.2.2	ynamic Process Model Starting Processes The Runtime Environment	$416 \\ 416 \\ 416$
	10.2	The D 10.2.1 10.2.2 Proces	ynamic Process Model Starting Processes The Runtime Environment ss Manager Interface	$ \begin{array}{r} 416 \\ 416 \\ 416 \\ 418 \end{array} $
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1	ynamic Process Model	416 416 416 418 418
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2	ynamic Process Model	416 416 416 418 418 418 418
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3	ynamic Process Model	$\begin{array}{c} 416 \\ 416 \\ 416 \\ 418 \\ 418 \\ 418 \\ 418 \\ 423 \end{array}$
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4	ynamic Process Model	$\begin{array}{c} 416 \\ 416 \\ 416 \\ 418 \\ 418 \\ 418 \\ 418 \\ 423 \\ 426 \end{array}$
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4	ynamic Process Model	$\begin{array}{c} 416 \\ 416 \\ 416 \\ 418 \\ 418 \\ 418 \\ 423 \\ 426 \\ 427 \end{array}$
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5	ynamic Process Model	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 427\\ \end{array}$
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5 Establ	ynamic Process Model	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 429\\ \end{array}$
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5 Establ 10.4.1	ynamic Process Model	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 429\\ 429\\ 429\\ 429\end{array}$
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5 Establ 10.4.1 10.4.2	ynamic Process Model	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 427\\ 429\\ 429\\ 430\\ \end{array}$
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5 Establ 10.4.1 10.4.2 10.4.3	ynamic Process Model.Starting Processes.The Runtime Environment.is Manager Interface.Processes in MPI.Starting Processes and Establishing Communication.Starting Multiple Executables and Establishing Communication.Reserved Keys.Spawn Example.Manager-worker Example Using MPI_COMM_SPAWN.ishing Communication.Names, Addresses, Ports, and All That.Server Routines.Client Routines.	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 429\\ 429\\ 430\\ 433\\ \end{array}$
	10.2	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5 Establ 10.4.1 10.4.2 10.4.3 10.4.4	ynamic Process Model.Starting Processes.The Runtime Environment.as Manager Interface.Processes in MPI.Starting Processes and Establishing Communication.Starting Multiple Executables and Establishing Communication.Reserved Keys.Spawn Example.Manager-worker Example Using MPI_COMM_SPAWN.ishing Communication.Names, Addresses, Ports, and All That.Server Routines.Name Publishing.	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 427\\ 429\\ 429\\ 430\\ 433\\ 434\\ \end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ 10.2.1 \\ 10.2.2 \\ {\rm Proces} \\ 10.3.1 \\ 10.3.2 \\ 10.3.3 \\ 10.3.4 \\ 10.3.5 \\ \\ {\rm Establ} \\ 10.4.1 \\ 10.4.2 \\ 10.4.3 \\ 10.4.4 \\ 10.4.5 \end{array}$	ynamic Process Model	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 427\\ 429\\ 429\\ 430\\ 433\\ 434\\ 436\end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ 10.2.1 \\ 10.2.2 \\ {\rm Proces} \\ 10.3.1 \\ 10.3.2 \\ 10.3.3 \\ 10.3.4 \\ 10.3.5 \\ \\ {\rm Establ} \\ 10.4.1 \\ 10.4.2 \\ 10.4.3 \\ 10.4.4 \\ 10.4.5 \end{array}$	ynamic Process Model	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 429\\ 429\\ 430\\ 433\\ 434\\ 436\\ 437\\ \end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ 10.2.1 \\ 10.2.2 \\ {\rm Proces} \\ 10.3.1 \\ 10.3.2 \\ 10.3.3 \\ 10.3.4 \\ 10.3.5 \\ \\ {\rm Establ} \\ 10.4.1 \\ 10.4.2 \\ 10.4.3 \\ 10.4.4 \\ 10.4.5 \end{array}$	ynamic Process Model	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 427\\ 429\\ 429\\ 430\\ 433\\ 434\\ 436\\ 437\\ 437\end{array}$
	10.2	$\begin{array}{c} {\rm The} \ {\rm D} \\ 10.2.1 \\ 10.2.2 \\ {\rm Proces} \\ 10.3.1 \\ 10.3.2 \\ 10.3.3 \\ 10.3.4 \\ 10.3.5 \\ \\ {\rm Establ} \\ 10.4.1 \\ 10.4.2 \\ 10.4.3 \\ 10.4.4 \\ 10.4.5 \end{array}$	ynamic Process Model	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 429\\ 429\\ 429\\ 430\\ 433\\ 434\\ 436\\ 437\\ 437\\ 437\\ 437\\ \end{array}$
	10.2 10.3 10.4	The D 10.2.1 10.2.2 Proces 10.3.1 10.3.2 10.3.3 10.3.4 10.3.5 Establ 10.4.1 10.4.2 10.4.3 10.4.4 10.4.5 10.4.6	ynamic Process Model	$\begin{array}{c} 416\\ 416\\ 416\\ 418\\ 418\\ 418\\ 423\\ 426\\ 427\\ 427\\ 429\\ 429\\ 429\\ 430\\ 433\\ 434\\ 436\\ 437\\ 437\\ 437\\ 438\\ \end{array}$

	10.5.1 Universe Size	39
	10.5.2 Singleton MPI_INIT	40
	10.5.3 MPI_APPNUM	41
	10.5.4 Releasing Connections	41
	10.5.5 Another Way to Establish MPI Communication	43
11 One	-Sided Communications 44	47
11.1	Introduction	47
11.2	Initialization	48
	11.2.1 Window Creation	49
	11.2.2 Window That Allocates Memory	51
	11.2.3 Window That Allocates Shared Memory	53
	11.2.4 Window of Dynamically Attached Memory 44	56
	11.2.5 Window Destruction	59
	11.2.6 Window Attributes	60
	11.2.7 Window Info	62
11.3	Communication Calls 44	63
	11.3.1 Put	64
	11.3.2 Get	67
	11.3.3 Examples for Communication Calls	68
		70
		71
	Get Accumulate Function	73
	Fetch and Op Function	75
		76
	11.3.5 Request-based RMA Communication Operations	77
	U U	82
11.5	Synchronization Calls 44	84
		86
		89
		93
		96
		98
		00
11.6		00
		00
		00
11.7		00
		09
	0	09
	9	10
		12
11.8	Examples	13

12	Exte	ernal I	nterfaces						523
	12.1	Introd	uction				 	 •	523
	12.2	Genera	alized Requests				 	 •	523
		12.2.1	Examples				 	 •	528
	12.3	Associ	ating Information with Status				 	 •	530
	12.4	MPI ai	nd Threads				 	 •	532
		12.4.1	General				 	 •	533
		12.4.2	Clarifications				 	 •	534
		12.4.3	Initialization				 	 •	535
13	I/O								541
	13.1		$uction \ldots \ldots$						541
			Definitions						541
	13.2		anipulation						543
			Opening a File						543
			Closing a File						546
		13.2.3	Deleting a File				 	 •	546
		13.2.4	Resizing a File				 	 •	547
		13.2.5	Preallocating Space for a File				 	 •	548
		13.2.6	Querying the Size of a File				 	 •	549
		13.2.7	Querying File Parameters				 	 •	549
		13.2.8	File Info				 	 •	551
			Reserved File Hints				 	 •	552
	13.3	File V	iews				 	 •	554
	13.4	Data A	Access				 	 •	557
		13.4.1	Data Access Routines				 	 •	557
			Positioning				 	 •	558
			Synchronism				 	 •	558
			Coordination				 	 	559
			Data Access Conventions				 	 	559
		13.4.2	Data Access with Explicit Offsets				 	 	560
		13.4.3	Data Access with Individual File Pointers .				 	 	565
		13.4.4	Data Access with Shared File Pointers				 	 	573
			Noncollective Operations				 	 	574
			Collective Operations						576
			Seek						578
		13.4.5	Split Collective Data Access Routines						580
	13.5		teroperability						587
			Datatypes for File Interoperability						589
			External Data Representation: "external32"						591
			User-Defined Data Representations						593
			Extent Callback						594
			Datarep Conversion Functions						594
		1354	Matching Data Representations						596
	13.6		tency and Semantics						597
	10.0		File Consistency						597
			Random Access vs. Sequential Files						600
			Progress						601
				-	•	-			

	13.6.4 Collective File Operations
	13.6.5 Nonblocking Collective File Operations
	13.6.6 Type Matching
	13.6.7 Miscellaneous Clarifications
	13.6.8 MPI_Offset Type
	13.6.9 Logical vs. Physical File Layout
	13.6.10 File Size
	13.6.11 Examples
	Asynchronous I/O
13 7	I/O Error Handling
	$I/O \text{ Error Classes} \dots \dots$
	Examples
15.9	1
	13.9.2 Subarray Filetype Constructor
14 Dor	orecated Interfaces 61
-	Deprecated since MPI-2.0
	Deprecated since MPI-2.2
	Deprecated since MPI-2.2
14.0	
15 Rer	noved Interfaces 61
	Removed MPI-1 Bindings
10.1	$15.1.1$ Overview \ldots
	15.1.2 Removed MPI-1 Functions
	15.1.3 Removed MPI-1 Datatypes
	15.1.3 Removed MPI-1 Datatypes
15.0	15.1.5 Removed MPI-1 Callback Prototypes
15.2	C++ Bindings 61
16 Bac	kward Incompatibilities 61
	Backward Incompatible since MPI-3.2
10.1	
17 Lan	guage Bindings 62
17.1	Fortran Support
	17.1.1 Overview
	17.1.2 Fortran Support Through the mpi_f08 Module 62
	17.1.3 Fortran Support Through the mpi Module
	17.1.4 Fortran Support Through the mpif.h Include File
	17.1.5 Interface Specifications, Procedure Names, and the Profiling Interface 62
	17.1.6 MPI for Different Fortran Standard Versions
	17.1.7 Requirements on Fortran Compilers
	17.1.8 Additional Support for Fortran Register-Memory-Synchronization . 63
	17.1.9 Additional Support for Fortran Numeric Intrinsic Types 63
	Parameterized Datatypes with Specified Precision and Exponent Range64
	Support for Size-specific MPI Datatypes
	Communication With Size-specific Types
	17.1.10 Problems With Fortran Bindings for MPI
	17.1.11 Problems Due to Strong Typing

	17.1	.12 Problems Due to Data Copying and Sequence Association with Sub-
		script Triplets
	17.1	.13 Problems Due to Data Copying and Sequence Association with Vector
		Subscripts
	17.1	.14 Special Constants
		.15 Fortran Derived Types
		.16 Optimization Problems, an Overview
		.17 Problems with Code Movement and Register Optimization 650
		Nonblocking Operations
		Persistent Operations
		One-sided Communication
		MPI_BOTTOM and Combining Independent Variables in Datatypes 65'
		Solutions
		The Fortran ASYNCHRONOUS Attribute
		Calling MPI_F_SYNC_REG
		0
		Module Variables and COMMON Blocks
		The (Poorly Performing) Fortran VOLATILE Attribute 665
	1 🖛 1	The Fortran TARGET Attribute
		.18 Temporary Data Movement and Temporary Memory Modification . 66
		.19 Permanent Data Movement
		.20 Comparison with C
		guage Interoperability
		.1 Introduction
		.2 Assumptions
		.3 Initialization
		.4 Transfer of Handles 670
	17.2	.5 Status
	17.2	.6 MPI Opaque Objects
		Datatypes
		Callback Functions
		Error Handlers
		Reduce Operations
	17.2	.7 Attributes
	17.2	.8 Extra-State
	17.2	.9 Constants
	17.2	.10 Interlanguage Communication
Α	Languag	e Bindings Summary 688
		ned Values and Handles
	A.1.	
	A.1.	
	A.1.	
		C Bindings
		Fortran 2008 Bindings with the mpi_f08 Module
		Fortran Bindings with mpif.h or the mpi Module
	A.1.	
	A.1. A.1.	
	A.1.	$5 \operatorname{Imo}(\operatorname{Reys}) = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$

	A.1.6	Info Values	706
A.2	C Bine	lings	708
	A.2.1	Point-to-Point Communication C Bindings	708
	A.2.2	Datatypes C Bindings	710
	A.2.3	Collective Communication C Bindings	712
	A.2.4	Groups, Contexts, Communicators, and Caching C Bindings	716
	A.2.5	Process Topologies C Bindings	718
	A.2.6	MPI Environmental Management C Bindings	721
	A.2.7	The Info Object C Bindings	722
	A.2.8	Process Creation and Management C Bindings	722
	A.2.9	One-Sided Communications C Bindings	723
	A.2.10	External Interfaces C Bindings	725
		I/O C Bindings	726
		Language Bindings C Bindings	728
		Tools / Profiling Interface C Bindings	729
		Tools / MPI Tool Information Interface C Bindings	729
		Deprecated C Bindings	731
A.3		n 2008 Bindings with the mpi_f08 Module	732
	A.3.1	Point-to-Point Communication Fortran 2008 Bindings	732
	A.3.2	Datatypes Fortran 2008 Bindings	737
	A.3.3	Collective Communication Fortran 2008 Bindings	742
	A.3.4	Groups, Contexts, Communicators, and Caching Fortran 2008 Bindi	
	A.3.5	Process Topologies Fortran 2008 Bindings	 760
	A.3.6	MPI Environmental Management Fortran 2008 Bindings	766
	A.3.7	The Info Object Fortran 2008 Bindings	768
	A.3.8	Process Creation and Management Fortran 2008 Bindings	769
	A.3.9	One-Sided Communications Fortran 2008 Bindings	771
	A.3.10	External Interfaces Fortran 2008 Bindings	776
	A.3.11	I/O Fortran 2008 Bindings	777
	A.3.12	Language Bindings Fortran 2008 Bindings	785
	A.3.13	Tools / Profiling Interface Fortran 2008 Bindings	786
A.4	Fortra	n Bindings with mpif.h or the mpi Module	787
		Point-to-Point Communication Fortran Bindings	787
	A.4.2	Datatypes Fortran Bindings	790
	A.4.3	Collective Communication Fortran Bindings	792
	A.4.4	Groups, Contexts, Communicators, and Caching Fortran Bindings	798
	A.4.5	Process Topologies Fortran Bindings	802
	A.4.6	MPI Environmental Management Fortran Bindings	805
	A.4.7	The Info Object Fortran Bindings	807
	A.4.8	Process Creation and Management Fortran Bindings	808
	A.4.9	One-Sided Communications Fortran Bindings	809
	A.4.10	External Interfaces Fortran Bindings	813
		I/O Fortran Bindings	814
		Language Bindings Fortran Bindings	818
		Tools / Profiling Interface Fortran Bindings	819
		Deprecated Fortran Bindings	819

Β	Cha	nge-Log	$\boldsymbol{821}$
	B.1	Changes from Version 3.1 to Version 3.2	821
		B.1.1 Changes in MPI-3.2	821
	B.2	Changes from Version 3.0 to Version 3.1	822
		B.2.1 Fixes to Errata in Previous Versions of MPI	822
		B.2.2 Changes in MPI-3.1	824
	B.3	Changes from Version 2.2 to Version 3.0	825
		B.3.1 Fixes to Errata in Previous Versions of MPI	825
		B.3.2 Changes in MPI-3.0	826
	B.4		830
	B.5	Changes from Version 2.0 to Version 2.1	833
Bi	bliog	raphy	839
Ge	enera	l Index	844
Ex	amp	les Index	848
M	PI C	onstant and Predefined Handle Index	851
M	PI D	eclarations Index	856
M	PI C	allback Function Prototype Index	857
\mathbf{M}	PI F	unction Index	858

List of Figures

5.1	Collective communications, an overview	151
5.2	Intercommunicator allgather	154
5.3	Intercommunicator reduce-scatter	155
5.4	Gather example	161
5.5	Gatherv example with strides	162
5.6	Gatherv example, 2-dimensional	163
5.7	Gatherv example, 2-dimensional, subarrays with different sizes	164
5.8	Gatherv example, 2-dimensional, subarrays with different sizes and strides .	166
5.9	Scatter example	171
5.10	Scattery example with strides	171
5.11	Scatterv example with different strides and counts	172
	Race conditions with point-to-point and collective communications	244
5.13	Overlapping Communicators Example	248
6.1	Intercommunicator creation using MPI_COMM_CREATE	272
6.2	Intercommunicator construction with MPI_COMM_SPLIT	276
6.3	Three-group pipeline	294
6.4	Three-group ring	296
$7.1 \\ 7.2 \\ 7.3$	Neighborhood gather communication example	350 370 371
7.4	Communication routine with sparse neighborhood all-to-all-w and without local data copying.	372
7.5	Two-dimensional parallel Poisson solver with persistent sparse neighborhood all-to-all-w and without local data copying.	373
11.1	Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows	483
11.2	Active target communication	486
	Active target communication, with weak synchronization	487
	Passive target communication	488
	Active target communication with several processes	491
	Symmetric communication	511
	Deadlock situation	511
	No deadlock	511
	Etypes and filetypes	542

13.2	Partitioning a file among parallel processes											542
13.3	Displacements								• •			555
13.4	Example array file layout											611
13.5	Example local array filetype for process 1 .											611
17.1	Status conversion routines	•	•	•				•		•	•	673

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	123
6.1	$MPI_COMM_\texttt{*} \ \mathrm{Function} \ \mathrm{Behavior} \ (\mathrm{in} \ \mathrm{Inter}\text{-}\mathrm{Communication} \ \mathrm{Mode}) \ . \ . \ . \ .$	291
8.1 8.2	Error classes (Part 1)	392 393
11.1 11.2	C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR	$\begin{array}{c} 461 \\ 501 \end{array}$
13.2	Data access routines	557 592 609
$\begin{array}{c} 15.2\\ 15.3 \end{array}$	Removed MPI-1 functions and their replacements	617 618 618 618
	Specific Fortran procedure names and related calling conventions Occurrence of Fortran optimization problems	629 656

Acknowledgments

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

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

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

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

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

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communication
- 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.

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
Agreement No. CCR-880 Esprit project P6643 (PI		mmission of the Eur	opean Community through	18
Esplit project r 0045 (r r				19
				20
MPI-1.2 and MPI-2.0:				21
Those who served as prin	mary coordinators i	n MPL1.2 and MPL	2.0 are	22 23
Those who served as prin	mary coordinators in	II WI 1-1.2 and WI 1-	2.0 are.	23 24
• Ewing Lusk, Conve	ener and Meeting Cl	hair		24
	Ū			26
• Steve Huss-Lederm	an, Editor			27
• Ewing Lusk, Miscel	llanv			28
				29
• Bill Saphir, Process	s Creation and Man	agement		30
• Marc Snir, One-Sid	led Communications			31
		5		32
• Bill Gropp and Ant	thony Skjellum, Ext	cended Collective O	perations	33
• Steve Huss-Lederm	on Extornal Interfe	005		34
• Steve Huss-Lederm	an, External interna	ices		35
• Bill Nitzberg, I/O				36
. A., J.,	D:11 Caralia and I		Dis lis	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
9		active participants w	vho attended MPI-2 Forum	42
meetings and are not me	ntioned 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	e 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 eraver support
34				
35	Argonne National	÷		
36 37	Bolt, Beranek, an			
38	California Institu			
39	Center for Compu	0		
40	Convex Computer	r Corporation		
41	Cray Research	+ Composition		
42	Digital Equipmen	nect Solutions, Inc.		
43	*	,		
44	General Electric (el Computing Centre		
45		Research Center for Info	rmation Technology	
46	Hewlett-Packard	TUSCALUI OCIUCI IOI IIIIO	manon recimology	
47	Hitachi			
48	Hughes Aircraft (Company		
	magnes micrait (, ombour		

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 Administration	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
MPI 2 operated on a very tight budget (in reality it had no budget 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 Itan	Ketti Ulderwood		16
Tl	he MPI Forum also ackr	nowledges and appreciates the	valuable input from people via	17
	and in person.			18
	-	s supported the MPI-2 effort the	hrough time and travel support	19
	people listed above.		0 11	20
				20
	rgonne National Labora	itory		22
Bu				22
	sco Systems, Inc.			20
	ray Inc.			24
	he HDF Group			26
	ewlett-Packard	1		20
	BM T.J. Watson Resear	ch		21
	diana University			28 29
		erche en Informatique et Aut	omatique (INRIA)	2 <i>9</i> 30
	tel Corporation			31
	awrence Berkeley Nation	0		
	awrence Livermore Nati	°		32
	os Alamos National Lab	poratory		33
	athworks			34
	ellanox Technologies			35
	icrosoft			36
	yricom			37
	EC Laboratories Europ	, -		38
	ak Ridge National Labo	-		39
	he Ohio State Universit	-		40
Pε	acific Northwest Nation	al Laboratory		41
\mathbf{Q}	Logic Corporation			42
	andia National Laborate	ories		43
Si	Cortex			44
Sil	licon Graphics Incorpor	rated		45
Su	ın Microsystems, Inc.			46
Uı	niversity of Alabama at	Birmingham		47
Uı	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	• Torsten Hoefler, Collective Communication and Process Topologies
20 21	• 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	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.
39 40	
41 42	
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	_	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
e-mail and in person. The following institutions s for the people listed above. Argonne National Laborat Auburn University Bull Cisco Systems, Inc. Cray Inc. Forschungszentrum Jülich Fujitsu The HDF Group Hewlett-Packard International Business Mag		ough time and travel support	24 25 26 27 28 29 30 31 32 33 34 35 36
Indiana University			37
	rche en Informatique et Autom	,	38
	ence & Engineering Corporatio	n	39
Intel Corporation			40
Lawrence Berkeley Nations	-		41
Lawrence Livermore Natio	0		42
Los Alamos National Labo	ratory		43
Mathworks			44
Mellanox Technologies			45
Microsoft			46
Myricom			47
NEC 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.
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	• Richard Granam, Steering committee, Font-to-Font Communication, Meeting Con- vener, and MPI-3.0 chair
31	vener, and wir F3.0 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	• Rajeev Thakur, I/O and One-Sided Communications
40	• Rajeev Thakur, 1/O and One-Sided Communications
41 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	
46	Deprecated Functions, Annex Change-Log, and Annex Language Bindings
46 47	 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	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 Chandrasekar	James Dinan	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-0816909
37	and #CCF-1144042 from the National Science Foundation. In addition, the HDF Group
38	and Sandia National Laboratories provided travel support for one U.S. academic each.
39 40	
40	MPI-3.1:
42	
43	MPI-3.1 is a minor update to the MPI Standard.
44	The editors and organizers of the MPI-3.1 have been:
45	• Martin Schulz, MPI-3.1 chair
46	
47	• William Gropp, Steering committee, Front matter, Introduction, One-Sided Commu-
48	nications, and Bibliography; Overall editor

• Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindings,	1
Deprecated Functions, Annex Change-Log, and Annex Language Bindings	2
- Dishand I. Cusham Stanning committee Masting Convenan	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	20
The following list includes some of the estive participants who attended MPI Ferrum	21 22
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 of participated in the	o man abcassions.		
			24
Charles Archer	Pavan Balaji	Purushotham V. Bangalore	25
Brian Barrett	Wesley Bland	Michael Blocksome	26
George Bosilca	Aurelien Bouteiller	Devendar Bureddy	27
Yohann Burette	Mohamad Chaarawi	Alexey Cheptsov	28
James Dinan	Dmitry Durnov	Thomas Francois	29
Edgar Gabriel	Todd Gamblin	Balazs Gerofi	30
Paddy Gillies	David Goodell	Manjunath Gorentla Venkata	31
Richard L. Graham	Ryan E. Grant	William Gropp	32
Khaled Hamidouche	Jeff Hammond	Amin Hassani	33
Marc-André Hermanns	Nathan Hjelm	Torsten Hoefler	34
Daniel Holmes	Atsushi Hori	Yutaka Ishikawa	35
Hideyuki Jitsumoto	Jithin Jose	Krishna Kandalla	36
Christos Kavouklis	Takahiro Kawashima	Chulho Kim	37
Michael Knobloch	Alice Koniges	Quincey Koziol	38
Sameer Kumar	Joshua Ladd	Ignacio Laguna	39
Huiwei Lu	Guillaume Mercier	Kathryn Mohror	40
Adam Moody	Tomotake Nakamura	Takeshi Nanri	41
Steve Oyanagi	Antonio J. Pẽna	Sreeram Potluri	42
Howard Pritchard	Rolf Rabenseifner	Nicholas Radcliffe	43
Ken Raffenetti	Raghunath Raja	Craig Rasmussen	44
Davide Rossetti	Kento Sato	Martin Schulz	45
Sangmin Seo	Christian Siebert	Anthony Skjellum	46
Brian Smith	David Solt	Jeffrey M. Squyres	47

48

1	Hari Subramoni Sh	inji Sumimoto	Alexander Supalov
2	Bronis R. de Supinski Sa	yantan Sur	Masamichi Takagi
3	Keita Teranishi Ra	ajeev Thakur	Fabian Tillier
4	Yuichi Tsujita Go	eoffroy Vallée	Rolf vandeVaart
5	Akshay Venkatesh Je	rome Vienne	Venkat Vishwanath
6	Anh Vo Hu	useyin S. Yildiz	Junchao Zhang
7	Xin Zhao		
8			
9	The MPI Forum also acknowledges	and appreciates t	the valuable input from people via
10	e-mail and in person.		
11	The following institutions supported	the MPI-3.1 effor	rt through time and travel support
12	for the people listed above.		
13	Argonne National Laboratory		
14	Auburn University		
15	Cisco Systems, Inc.		
16	Cray		
17	EPCC, The University of Edinburg	h	
18	ETH Zurich		
19	Forschungszentrum Jülich		
20	Fujitsu		
21	German Research School for Simula	ation Sciences	
22	The HDF Group		
23	International Business Machines		
24	INRIA		
25	Intel Corporation		
26	Jülich Aachen Research Alliance, H	ligh-Performance	Computing (JARA-HPC)
27	Kyushu University		
28	Lawrence Berkeley National Labora	-	
29	Lawrence Livermore National Labo	ratory	
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 Birmingh	am	
43	University of Houston		
44	University of Illinois at Urbana-Cha	ampaign	
45 46	University of Oregon	C ···	
40 47	University of Stuttgart, High Perfor	rmance Computi	ng Center Stuttgart (HLKS)
47	University of Tennessee, Knoxville		
-10	University of Tokyo		

MPI-4.0:	1
MPI-4.0 is a major update to the MPI Standard.	2
······································	3
	4
	6
	7
	8
	9
	10
	11
	12
	13
	14
	15 16
	10
	18
	19
	20
	21
	22
	23
	24
	25
	26
	27 28
	28
	30
	31
	32
	33
	34
	35
	36
	37
	38
	39 40
	40
	42
	43
	44
	45
	46
	47
	48
Unofficial Draft for Comment Only	xxxiii

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 [52, 53], 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 [59]. At this workshop 24 the basic features essential to a standard message-passing interface were discussed, and a 25working group established to continue the standardization process. 26

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

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

- 44
- $45 \\ 46$

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

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

Unofficial Draft for Comment Only

1

2

3

4

5 6

7

8 9 10

11 12

13

14

15

16

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

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

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

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

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

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

Background of MPI-1.3 and MPI-2.1 1.4

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

Unofficial Draft for Comment Only

1 2

3

4

56

7

8 9

10 11

12

13

1415

16

17

18

19

20

21

22

23

 24 25

26

27

28

29

30

31 32

33

34 35

36

37

38 39

40

41

43

45

46

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

 $15 \\ 16$

17 18

19

20

21 22

23

24 25

26

27 28

29

30 31

32 33

34

35

36

37

1.5 Background of MPI-2.2

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

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

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

1.6 Background of MPI-3.0

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

38 39 40

41

1.7 Background of MPI-3.1

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

47

1.8 Background of MPI-4.0

1.9 Who Should Use This Standard?

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

1.10 What Platforms Are Targets for Implementation?

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

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

1.11 What Is Included in the Standard?

The standard includes:

3536 • Point-to-point communication, 37 38 • Datatypes, 39 • Collective operations, 40 41 • Process groups, 42• Communication contexts. 43 44 • Process topologies, 4546• Environmental management and inquiry, 4748 • The Info object,

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

 $\mathbf{2}$

Process creation and management,
One-sided communication,
External interfaces,
Parallel file I/O,
Language bindings for Fortran and C,
Tool support. **1.12 What Is Not Included in the Standard?**The standard does not specify:

Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,
Program construction tools,
Debugging facilities.

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

1.13 Organization of This Document

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

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

- Chapter 6, Groups, Contexts, Communicators, and Caching, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*.
- Chapter 7, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 8, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.
- 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 ??, ??, covers interfaces that allow debuggers, performance analyzers, and other tools to obtain data about the operation of MPI processes. This chapter includes Section ?? (??), which was a chapter in previous versions of MPI.
- Chapter 14, Deprecated Interfaces, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
- Chapter 15, Removed Interfaces, describes routines and constructs that have been removed from MPI. Some of these were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard. Others of these were deprecated in MPI-3, and the MPI Forum decided to remove these from the MPI-4 standard.
- Chapter 16, Backward Incompatibilities, describes incompatibilities with previous versions of MPI.
- Chapter 17, Language Bindings, discusses Fortran issues, and describes language interoperability aspects between C and Fortran.

The Appendices are:

• Annex A, Language Bindings Summary, gives specific syntax in C and Fortran, for all MPI functions, constants, and types.

Unofficial Draft for Comment Only

- Annex B, Change-Log, summarizes some changes since the previous version of the standard.
- Several Index pages show the locations of examples, constants and predefined handles, callback routine prototypes, and all MPI functions.

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

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

 $\mathbf{2}$

Chapter 2

MPI Terms and Conventions

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

2.1 Document Notation

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

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

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

2.2 Naming Conventions

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

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

 $45 \\ 46$

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

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

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

2.3 Procedure Specification

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

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

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

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

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

28 29

30

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

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

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

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

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

48

1

2

3

4

5

6

7 8

9

10

11 12

13 14

15

16

17

18 19

20 21

<pre>void copyIntBuffer(int *pin, int *pout, int len)</pre>	1
{ int i;	2
<pre>for (i=0; i<len; *pout++="*pin++;</pre" ++i)=""></len;></pre>	$\frac{3}{4}$
}	4 5
then a call to it in the following code fragment has aliased arguments.	6
	7
int $a[10];$	8
<pre>copyIntBuffer(a, a+3, 7);</pre>	9
Although the C language allows this, such usage of MPI procedures is forbidden unless	10
otherwise specified. Note that Fortran prohibits aliasing of arguments.	11
All MPI functions are first specified in the language-independent notation. Immediately	12
below this, language dependent bindings follow:	13
• The ISO C version of the function.	14 15
• The Fortran version used with USE mpi_f08.	16
• The Fortran version used with USE mpi_108.	17
• The Fortran version of the same function used with USE mpi or INCLUDE 'mpif.h'.	18
An exception is Section ?? "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
2.4 Semantic Terms	24
	25
When discussing MPI procedures the following semantic terms are used.	26
	27
nonblocking 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 operation completes 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
local A procedure is local if completion of the procedure depends only on the local executing	36
process.	37
process.	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 44
the same communicator must be executed in the same order by all members of the	44
process group.	46
	47
	48

	12 CHAPTER 2. MPI TERMS AND CONVENTIO	NS
1 2 3 4 5 6	<pre>predefined A predefined datatype is a datatype with a predefined (constant) name (s</pre>	
7	derived A derived datatype is any datatype that is not predefined.	
8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24	portable A datatype is portable if it is a predefined datatype, or it is derived fn a portable datatype using only the type constructors MPI_TYPE_CONTIGUO 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 porta because all displacements in the datatype are in terms of extents of one predefi- datatype. Therefore, if such a datatype fits a data layout in one memory, it fit the corresponding data layout in another memory, if the same declarations w used, even if the two systems have different architectures. On the other hand, 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 displa ments (e.g., providing padding to meet alignment restrictions). These displacement are unlikely to be chosen correctly if they fit data layout on one memory, but used for data layouts on another process, running on a processor with a differ architecture.	US, able ned will vere if a ace- ents are
25 26 27 28	equivalent Two datatypes are equivalent if they appear to have been created with the satisfies sequence of calls (and arguments) and thus have the same typemap. Two equivals datatypes do not necessarily have the same cached attributes or the same names.	ent
29	25 Data Types	

2.5 Data Types

2.5.1 Opaque Objects

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

 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.*)

32 33 34

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

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:

are		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
tra	n are as follows:	45
	MPI_BOTTOM	46
	MPI_STATUS_IGNORE	47
	MPI_STATUSES_IGNORE	48

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

¹⁹ MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to ²⁰ the same routine may pass by reference actual arguments of different types. The mechanism ²¹ for providing such arguments will differ from language to language. For Fortran with the ²² include file mpif.h or the mpi module, the document uses <type> to represent a choice ²³ variable; with the Fortran mpi_f08 module, such arguments are declared with the Fortran ²⁴ 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 17.1.1. (End of advice to implementors.)

³² 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

⁴⁷ For I/O there is a need to give the size, displacement, and offset into a file. These quantities ⁴⁸ 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 Interfaces

A number of chapters refer to deprecated or replaced MPI constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 14, but that users are recommended not to continue using, since better solutions were provided with newer

Unofficial Draft for Comment Only

 $\overline{7}$

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.

⁸ Some of the deprecated constructs are now removed, as documented in Chapter 15.
 ⁹ 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.

MPI_Copy_function2MPI-2.0MPI_Comm_copy_attr_function2COPY_FUNCTION3MPI-2.0COMM_COPY_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_Comm_delete_attr_function2				
MPI_ADDRESSMPI-2.0MPI-3.0MPI_GET_ADDRESSMPI_TYPE_HINDEXEDMPI-2.0MPI-3.0MPI_TYPE_CREATE_HINDEXEDMPI_TYPE_HVECTORMPI-2.0MPI-3.0MPI_TYPE_CREATE_HVECTORMPI_TYPE_STRUCTMPI-2.0MPI-3.0MPI_TYPE_CREATE_STRUCTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_UB ¹ MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_UB ¹ MPI-2.0MPI-3.0MPI_CORM_CREATE_RESIZEDMPI_ERRHANDLER_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_RERHANDLERMPI_ERRHANDLER_GETMPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_ERRHANDLER_SETMPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI-2.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_GREATEMPI-2.0MPI_COMM_SET_ERRHANDLERMPI_LCOPY_FN3MPI-2.0MPI_COMM_SET_KINCTON2MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_RRE_KEYVALMPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_COM_SINER_HINDEXED_INTEGER4-MPI_COMM_SET_ATTRMPI_COMBINER_HINDEXED_INTEGER4-MPI-2.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-2.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-2.0M	1			Replacement
MPI_TYPE_HINDEXEDMPI-2.0MPI-3.0MPI_TYPE_CREATE_HINDEXEDMPI_TYPE_HVECTORMPI-2.0MPI-3.0MPI_TYPE_CREATE_HVECTORMPI_TYPE_STRUCTMPI-2.0MPI-3.0MPI_TYPE_CREATE_STRUCTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_UB ¹ MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_ERRHANDLER_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_ERRHANDLERMPI_ERRHANDLER_SETMPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_Handler_function ² MPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_CREATE_KEYVALMPI_DUP_FN ³ MPI-2.0MPI_COMM_CREATE_KEYVALMPI_DUP_FN ³ MPI-2.0MPI_COMM_NULL_COPY_FN ³ MPI_AUL_COPY_FN ³ MPI-2.0MPI_COMM_NULL_COPY_FN ³ MPI_2.0MPI_COMM_NULL_COPY_FN ³ MPI-2.0MPI_COMM_NULL_COPY_FN ³ MPI-2.0MPI_COMM_NULL_COPY_FN ³ MPI_NULL_DELETE_FN ³ MPI-2.0MPI_COMM_NULL_COPY_FN ³ MPI-2.0MPI_COMM_NULL_COPY_FN ³ MPI-2.0MPI_COMM_CREATE_ATTR_FUNCTIONMPI-2.0MPI_COMM_CREATE_ATTR_FUNCTIONMPI_Delete_function ² MPI-2.0MPI_COMM_CREATE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI-COMM_SET_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMBINER_TRECTOR ⁴				
MPI_TYPE_HVECTORMPI-2.0MPI-3.0MPI_TYPE_CREATE_HVECTORMPI_TYPE_STRUCTMPI-2.0MPI-3.0MPI_TYPE_CREATE_STRUCTMPI_TYPE_EXTENTMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_UB ¹ MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_UB ¹ MPI-2.0MPI-3.0MPI_COMM_CREATE_ERESIZEDMPI_UB ¹ MPI-2.0MPI-3.0MPI_COMM_CREATE_ERENANDLERMPI_ERRHANDLER_GETMPI-2.0MPI-3.0MPI_COMM_CREATE_ERRHANDLERMPI_ERRHANDLER_SETMPI-2.0MPI-3.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_CREATE_KEYVALMPI_UL_COPY_FN3MPI-2.0MPI_COMM_DUP_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_ODUP_FN3MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_COP_function2MPI-2.0MPI_COMM_DUP_FN3MPI_COP_fUNCTION3MPI-2.0MPI_COMM_DUP_FN3MPI_20MPI-2.0MPI_COMM_COPY_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_GET_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_SET_ATTRMPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_NTEGER4-MPI-3.0MPI_COMBINER_S				
MPI_TYPE_STRUCTMPI-2.0MPI-3.0MPI_TYPE_CREATE_STRUCTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_UB ¹ MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_UB ¹ MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_ERRHANDLER_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_RENANDLERMPI_ERRHANDLER_GETMPI-2.0MPI-3.0MPI_COMM_GET_ERRHANDLERMPI_ERRHANDLER_SETMPI-2.0MPI-3.0MPI_COMM_GET_ERRHANDLERMPI_Handler_function ² MPI-2.0MPI-3.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_DUP_FN ³ MPI_DUP_FN ³ MPI-2.0MPI_COMM_NULL_OPY_FN ³ MPI_UL_COPY_FN ³ MPI-2.0MPI_COMM_NULL_OPY_FN ³ MPI_2.0MPI-2.0MPI_COMM_NULL_DELETE_FN ³ MPI-2.0MPI-2.0MPI_COMM_NULL_DELETE_FN ³ MPI-2.0MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION ² MPI_COMJ_OPY_FN ³ MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION ² MPI_COMJ_OPY_FN ³ MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION ² MPI_2.0MPI-2.0MPI_COMM_DELETE_ATTRMPI_COMJ_OPY_FN ³ MPI-2.0MPI_COMM_GET_ATTRMPI_COMJ_OPY_FN ³ MPI-2.0MPI_COMM_DELETE_ATTRMPI_COMJ_OPY_FN ³ MPI-2.0MPI_COMM_DELETE_ATTRMPI_COMJ_OPY_FN ³ MPI-2.0MPI_COMM_DELETE_ATTRMPI_COMJ_OPY_FN		-		
MPI_TYPE_EXTENTMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_TYPE_UBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_UB1MPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_UB1MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_UB1MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_ERRHANDLER_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_ERRHANDLERMPI_ERRHANDLER_SETMPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_RERHANDLER_SETMPI-2.0MPI-2.0MPI_COMM_SET_ERRHANDLERMPI_Handler_function2MPI-2.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_DUP_FN3MPI_DUP_FN3MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_COM_function2MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_COM_function2MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_2.0MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_2.0MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTIONMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI				
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_UB ¹ MPI-2.0 MPI-3.0 MPI_TYPE_GET_EXTENT MPI_UB ¹ MPI-2.0 MPI-3.0 MPI_TYPE_GET_EXTENT MPI_BRHANDLER_CREATE MPI-2.0 MPI-3.0 MPI_COMM_CREATE_ERRHANDLER MPI_ERRHANDLER_GET MPI-2.0 MPI-3.0 MPI_COMM_CREATE_ERRHANDLER MPI_ERRHANDLER_SET MPI-2.0 MPI-3.0 MPI_COMM_SET_ERRHANDLER MPI_Handler_function ² MPI-2.0 MPI-3.0 MPI_COMM_CREATE_KEYVAL MPI_KEYVAL_CREATE MPI-2.0 MPI_COMM_CREATE_KEYVAL MPI_COMM_DUP_FN ³ MPI_DUP_FN ³ MPI-2.0 MPI_COMM_NULL_COPY_FN ³ MPI_VUL_OELETE_FN ³ MPI-2.0 MPI_COMM_NUL_COPY_FN ³ MPI_2.0 MPI_COMM_NUL_COPY_FN ³ MPI-2.0 MPI_COPy_function ² MPI-2.0 MPI_COMM_COPY_ATTR_FUNCTION MPI_COPy_function ² MPI-2.0 MPI_COMM_DELETE_FN ³ MPI_2.0 MPI_COMM_DUP_COM_ADUP_COMM_DELETE_FN ³ MPI-2.0 MPI_COMM_DUP_CONST_ATTR_FUNCTION MPI_2.0 MPI_COMM_DELETE_ATTR_FUNCTION </td <td></td> <td></td> <td></td> <td></td>				
MPI_TYPE_LBMPI-2.0MPI-3.0MPI_TYPE_GET_EXTENTMPI_LB1MPI-2.0MPI-3.0MPI_TYPE_GREATE_RESIZEDMPI_UB1MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_ERRHANDLER_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_ERRHANDLERMPI_ERRHANDLER_SETMPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_Handler_function2MPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_KEYVALMPI_DUP_FN3MPI-2.0MPI-COMM_DUP_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_NULL_DELETE_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_COM_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_COM_STURCTION3MPI-2.0MPI_COMM_DUP_FN3MPI_Delete_function2MPI-2.0MPI_COMM_DULL_COPY_ATTR_FUNCTIONMPI_Delete_function3MPI-2.0MPI_COMM_DELETE_FN3MPI-2.0MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTIONMPI_DELETEMPI-2.0MPI_COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-Predefined datatype.				
MPI_LB1MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_UB1MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_ERRHANDLER_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_RENANDLERMPI_ERRHANDLER_SETMPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_Handler_function2MPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_CREATE_KEYVALMPI_LVL_COPY_FN3MPI-2.0MPI_COMM_DUP_FN3MPI_DUP_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_Cop_function2MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_Copy_function3MPI-2.0MPI_COMM_OUP_TATTR_FUNCTIONMPI_DElete_function2MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_SET_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_SET_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_SET_ATTRMPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI-COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI-COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI-COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI-COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI-COMBINER_HINDEXED_INTEGER4- <td></td> <td></td> <td></td> <td></td>				
MPI_UB1MPI-2.0MPI-3.0MPI_TYPE_CREATE_RESIZEDMPI_ERRHANDLER_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_ERRHANDLERMPI_ERRHANDLER_GETMPI-2.0MPI-3.0MPI_COMM_GET_ERRHANDLERMPI_ERRHANDLER_SETMPI-2.0MPI-3.0MPI_COMM_GET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI-3.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_CREATEMPI-2.0MPI_COMM_CREATE_KEYVALMPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_Copy_function2MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_Copy_function2MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_Copy_function2MPI-2.0MPI_COMM_NULL_COPY_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_COMM_COPY_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_COMM_DELETE_ATTRMPI_COMM_DELETE_FUNCTION3MPI-2.0COMM_COPY_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_GET_ATTRMPI_COMBINER_HINDEXED_INTEGER4-MPI_COMBINER_HINDEXED_ATTRMPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBIN		MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
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_GET_ERRHANDLER MPI_Handler_function ² MPI-2.0 MPI-3.0 MPI_COMM_SET_ERRHANDLER MPI_KEYVAL_CREATE MPI-2.0 MPI_COMM_CREATE_KEYVAL MPI_NULL_COPY_FN3 MPI-2.0 MPI_COMM_DUP_FN3 MPI_OUP_FN3 MPI-2.0 MPI_COMM_NULL_COPY_FN3 MPI_Copy_function ² MPI-2.0 MPI_COMM_NULL_DELETE_FN3 MPI_Copy_function ² MPI-2.0 MPI_COMM_OPT_FN3 MPI_Delete_function ² MPI-2.0 MPI_COMM_NULL_DELETE_FN3 MPI_Copy_function ³ MPI-2.0 MPI_COMM_COPY_ATTR_FUNCTION MPI_Delete_function ² MPI-2.0 COMM_COPY_ATTR_FUNCTION MPI_Copy_functioN3 MPI-2.0 COMM_DELETE_ATTR_FUNCTION MPI_ATTR_DELETE MPI-2.0 MPI_COMM_DELETE_ATTR_FUNCTION MPI_ATTR_GET MPI-2.0 MPI_COMM_DELETE_ATTR MPI_ATTR_DELETE MPI-2.0 MPI_COMM_SET_ATTR MPI_COMBINER_HINDEXED_INTEGER ⁴ MPI-3.0 MPI_COMBINER_HINDEX		MPI-2.0	MPI-3.0	
MPI_ERRHANDLER_GETMPI-2.0MPI-3.0MPI-COMM_GET_ERRHANDLERMPI_ERRHANDLER_SETMPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_Handler_function2MPI-2.0MPI-3.0MPI_COMM_SET_ERRHANDLERMPI_KEYVAL_CREATEMPI-2.0MPI-2.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_FREEMPI-2.0MPI_COMM_DUP_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_Copy_function2MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_Copy_function2MPI-2.0MPI_COMM_OPY_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI_COMM_GET_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_SET_ATTRMPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI-COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-CO		MPI-2.0	MPI-3.0	
MPI_ERRHANDLER_SET MPI-2.0 MPI-3.0 MPI_COMM_SET_ERRHANDLER MPI_Handler_function ² MPI-2.0 MPI-3.0 MPI_Comm_errhandler_function ² MPI_KEYVAL_CREATE MPI-2.0 MPI_COMM_CREATE_KEYVAL MPI_KEYVAL_FREE MPI-2.0 MPI_COMM_FREE_KEYVAL MPI_DUP_FN ³ MPI-2.0 MPI_COMM_NULL_COPY_FN ³ MPI_NULL_OELETE_FN ³ MPI-2.0 MPI_COMM_NULL_DELETE_FN ³ MPI_Copy_function ² MPI-2.0 MPI_COMM_NULL_DELETE_FN ³ MPI_Copy_function ² MPI-2.0 MPI_COMM_COPY_ATTR_FUNCTION MPI_Delete_function ² MPI-2.0 COMM_COPY_ATTR_FUNCTION MPI_Delete_function ² MPI-2.0 COMM_COPY_ATTR_FUNCTION MPI_Delete_function ² MPI-2.0 COMM_DELETE_ATTR_FUNCTION MPI_ATTR_DELETE MPI-2.0 MPI_COMM_DELET_ATTR_FUNCTION MPI_ATTR_DELETE MPI-2.0 MPI_COMM_SET_ATTR MPI_ATTR_DELETE MPI-2.0 MPI_COMM_SET_ATTR MPI_COMBINER_HVECTOR_INTEGER ⁴ - MPI_COMM_SET_ATTR MPI_COMBINER_HVECTOR_INTEGER ⁴ - MPI-3.0 MPI_COMBINER_HINDEXED ⁴ MPI_COMBINER_STRUCT_INTEGER ⁴ - MPI-3.0				
MPI_Handler_function2MPI-2.0MPI-3.0MPI_Comm_errhandler_function2MPI_KEYVAL_CREATEMPI-2.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_FREEMPI-2.0MPI_COMM_DUP_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_Copy_function2MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_Copy_function2MPI-2.0MPI_COMM_COPY_ATTR_FUNCTION2COPY_FUNCTION3MPI-2.0COMM_COPY_ATTR_FUNCTION2MPI_Delete_function2MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION2MPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION2MPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0* Predefined datatypeNPI-3.0* Callback prototype definitionno direct replacement* Predefined callback routine* Constant.Other entries are regular MPI routines.		MPI-2.0	MPI-3.0	
MPI_KEYVAL_CREATEMPI-2.0MPI_COMM_CREATE_KEYVALMPI_KEYVAL_FREEMPI-2.0MPI_COMM_DUP_FN3MPI_DUP_FN3MPI-2.0MPI_COMM_DUP_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_Copy_function2MPI-2.0MPI_COMM_COPY_ATTR_FUNCTION2MPI_Copy_FUNCTION3MPI-2.0COMM_COPY_ATTR_FUNCTION2MPI_ATTR_Delete_function2MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION2MPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION2MPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION2MPI_ATTR_GETMPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION2MPI_COMBINER_HVECTOR_INTEGER4-MPI_COMBINER_HVECTOR4MPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER5-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_TINDEXED_INTEGER5-MPI-3.0<				
MPI_KEYVAL_FREEMPI-2.0MPI_COMM_FREE_KEYVALMPI_DUP_FN3MPI-2.0MPI_COMM_DUP_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_Copy_function2MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_Copy_function2MPI-2.0MPI_COMM_COPY_ATTR_FUNCTION2COPY_FUNCTION3MPI-2.0COMM_COPY_ATTR_FUNCTION3MPI_Delete_function2MPI-2.0COMM_COPY_ATTR_FUNCTION2DELETE_FUNCTION3MPI-2.0COMM_DELETE_ATTR_FUNCTION3MPI-2.0MPI_COMM_DELETE_ATTR_FUNCTION2MPI-2.0DELETE_FUNCTION3MPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_DELETEMPI-2.0MPI_COMM_GET_ATTRMPI_ATTR_GETMPI-2.0MPI_COMM_GET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_CANCEL for send requestsMPI-3.2no direct replacement ¹ Predefined datatypeMPI-3.0C language bindingMPI_CANCEL for send requestsMPI-3.2no direct replacement ¹ Predefined callback routineMPI-3.0A Constant.Other entries are regular MPI routines		MPI-2.0	MPI-3.0	
MPI_DUP_FN3MPI-2.0MPI_COMM_DUP_FN3MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_NULL_DELETE_FN3MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_Copy_function2MPI-2.0COMM_COPY_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_GETMPI-2.0MPI_COMM_DELETE_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-2.2MPI-3.0MPI_COMBINER_STRUCT4MPI-2.3MPI-3.0MPI_COMBINER_STRUCT4MPI_CONSTRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-2.0MPI-3.0MPI_COMBINER_STRUCT4MPI-3.0MPI_COMBINER_STRUCT4MPI-3.0MPI_COMSINER_STRUCT4-MPI-2.2MPI-3.0MPI-2.3NO direct replacementMPI-2.4-MPI	MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL
MPI_NULL_COPY_FN3MPI-2.0MPI_COMM_NULL_COPY_FN3MPI_NULL_DELETE_FN3MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_Copy_function2MPI-2.0MPI_Comm_copy_attr_function2COPY_FUNCTION3MPI-2.0COMM_COPY_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0COMM_DELETE_ATTRMPI_ATTR_GETMPI-2.0MPI_COMM_GET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_CANCEL for send requestsMPI-3.2no direct replacement1Predefined datatype2Callback prototype definition3Predefined callback routine4ConstantOther entries are regular MPI routines	MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL
MPI_NULL_DELETE_FN3MPI-2.0MPI_COMM_NULL_DELETE_FN3MPI_Copy_function2MPI-2.0MPI_Comm_copy_attr_function2COPY_FUNCTION3MPI-2.0COMM_COPY_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_Comm_delete_attr_function2DELETE_FUNCTION3MPI-2.0COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_GETMPI-2.0MPI_COMM_GET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_CANCEL for send requestsMPI-3.2no direct replacement1Predefined datatype.22Callback prototype definition3Predefined callback routine4Constant.Other entries are regular MPI routines	MPI_DUP_FN ³	MPI-2.0		MPI_COMM_DUP_FN ³
MPI_Copy_function2MPI-2.0MPI_Comm_copy_attr_function2COPY_FUNCTION3MPI-2.0COMM_COPY_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_Comm_delete_attr_function2DELETE_FUNCTION3MPI-2.0COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI_COMM_GET_ATTRMPI_ATTR_PUTMPI-2.0MPI_COMM_SET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-2.2MPI-3.0MPI_COMBINER_HINDEXED4MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-2.2MPI-3.0MPI-COMBINER_STRUCT4MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI-2.2MPI-3.0MPI-COMBINER_STRUCT4MPI_2.2MPI-3.0MPI-COMBINER_STRUCT4MPI_CANCEL for send requestsMPI-3.0C language binding1 Predefined datatype2 Callback prototype definition3 Predefined callback routine4 ConstantOther entries are regular MPI routines	MPI_NULL_COPY_FN ³	MPI-2.0		MPI_COMM_NULL_COPY_FN ³
COPY_FUNCTION3MPI-2.0COMM_COPY_ATTR_FUNCTIONMPI_Delete_function2MPI-2.0MPI_Comm_delete_attr_function2DELETE_FUNCTION3MPI-2.0COMM_DELETE_ATTR_FUNCTIONMPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_PUTMPI-2.0MPI_COMM_SET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_INTEGER4-MPI-3.0PI_COMBINER_STRUCT_STRUCT_STRUCT4MPI-3.0MPI_CONSTRUCT_STRUCT4-MPI_CONSTRUCT-PI_CONSTRUCT-MPI_CONSTRUCT-MPI_CONSTRUCT-MPI_SCONSTRUCT-MPI_SCONSTRUCT-MPI_SCONSTRUCT-MPI_SCONSTRUCT-MPI_SCONSTRUCT-MPI_SCONSTRUCT-MPI_SCONSTRUCT	MPI_NULL_DELETE_FN ³	MPI-2.0		MPI_COMM_NULL_DELETE_FN ³
MPI_Delete_function2MPI-2.0MPI_Comm_delete_attr_function2DELETE_FUNCTION3MPI-2.0COMM_DELETE_ATTR_FUNCTION3MPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_GETMPI-2.0MPI_COMM_GET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0Predefined datatype.2MPI-3.0Callback prototype definitionno direct replacementPredefined callback routine4 Constant.Other entries are regular MPI routines.	MPI_Copy_function ²	MPI-2.0		MPI_Comm_copy_attr_function ²
DELETE_FUNCTION3MPI-2.0COMM_DELETE_ATTR_FUNCTION3MPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_GETMPI-2.0MPI_COMM_GET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_CONSTANLMPI-3.2MPI-3.0Callback prototype definition ³ Predefined callback routine ⁴ ConstantOther entries are regular MPI routines	COPY_FUNCTION ³	MPI-2.0		COMM_COPY_ATTR_FUNCTION ³
MPI_ATTR_DELETEMPI-2.0MPI_COMM_DELETE_ATTRMPI_ATTR_GETMPI-2.0MPI_COMM_GET_ATTRMPI_ATTR_PUTMPI-2.0MPI_COMM_SET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_CONSIDER_STRUCT_INTEGER4-MPI-3.0Predefined datatype2 Callback prototype definition3 Predefined callback routine4 ConstantOther entries are regular MPI routines.	$MPI_Delete_function^2$	MPI-2.0		MPI_Comm_delete_attr_function ²
MPI_ATTR_GETMPI-2.0MPI_COMM_GET_ATTRMPI_ATTR_PUTMPI-2.0MPI_COMM_SET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HVECTOR4MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED4MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT4MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT4MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0C language bindingMPI_CANCEL for send requestsMPI-3.2mo direct replacement1 Predefined datatype2 Callback prototype definition3 Predefined callback routine4 Constant.Other entries are regular MPI routines	DELETE_FUNCTION ³	MPI-2.0		COMM_DELETE_ATTR_FUNCTIO
MPI_ATTR_PUTMPI-2.0MPI_COMM_SET_ATTRMPI_COMBINER_HVECTOR_INTEGER4-MPI-3.0MPI_COMBINER_HVECTOR4MPI_COMBINER_HINDEXED_INTEGER4-MPI-3.0MPI_COMBINER_HINDEXED4MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0MPI_COMBINER_STRUCT4MPI_COMBINER_STRUCT_INTEGER4-MPI-3.0C language bindingMPI_CANCEL for send requestsMPI-3.2MPI-3.0C language binding1 Predefined datatypeo direct replacement2 Callback prototype definition3 Predefined callback routine4 Constant.Other entries are regular MPI routines	MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR
MPI_COMBINER_HVECTOR_INTEGER ⁴ - MPI-3.0 MPI_COMBINER_HVECTOR ⁴ MPI_COMBINER_HINDEXED_INTEGER ⁴ - MPI-3.0 MPI_COMBINER_HINDEXED ⁴ MPI_COMBINER_STRUCT_INTEGER ⁴ - MPI-3.0 MPI_COMBINER_STRUCT ⁴ MPI_COMBINER_STRUCT_INTEGER ⁴ - MPI-3.0 MPI_COMBINER_STRUCT ⁴ MPI_COMBINER_STRUCT_INTEGER ⁴ - MPI-3.0 C language binding MPI_CANCEL for send requests MPI-3.2 mo direct replacement ¹ Predefined datatype. - - o direct replacement ² Callback prototype definition. - - - ³ Predefined callback routine. - - - ⁴ Constant. Other entries are regular MPI routines. - -	MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR
MPI_COMBINER_HINDEXED_INTEGER ⁴ - MPI-3.0 MPI_COMBINER_STRUCT MPI_COMBINER_STRUCT_INTEGER ⁴ - MPI-3.0 MPI_COMBINER_STRUCT ⁴ MPI: MPI-2.2 MPI-3.0 C language binding MPI_CANCEL for send requests MPI-3.2 no direct replacement ¹ Predefined datatype. - - - ² Callback prototype definition. - - - ³ Predefined callback routine. - - - ⁴ Constant. Other entries are regular MPI routines. - -	MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR
MPI_COMBINER_STRUCT_INTEGER ⁴ - MPI-3.0 MPI_COMBINER_STRUCT ⁴ MPI: MPI-2.2 MPI-3.0 C language binding MPI_CANCEL for send requests MPI-3.2 no direct replacement ¹ Predefined datatype. 2 Callback prototype definition. ³ Predefined callback routine. - - ⁴ Constant. Other entries are regular MPI routines. -	MPI_COMBINER_HVECTOR_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_HVECTOR ⁴
MPI_COMBINER_STRUCT_INTEGER ⁴ - MPI-3.0 MPI_COMBINER_STRUCT ⁴ MPI: MPI-2.2 MPI-3.0 C language binding MPI_CANCEL for send requests MPI-3.2 no direct replacement ¹ Predefined datatype. - - - ² Callback prototype definition. - - - ³ Predefined callback routine. - - - ⁴ Constant. Other entries are regular MPI routines. - -	MPI_COMBINER_HINDEXED_INTEGER ⁴	-	MPI-3.0	MPI_COMBINER_HINDEXED ⁴
MPI_CANCEL for send requests MPI-3.2 no direct replacement 1 Predefined datatype. 2 Callback prototype definition. 3 Predefined callback routine. 4 Constant. Other entries are regular MPI routines. 0 0		-	MPI-3.0	MPI_COMBINER_STRUCT ⁴
 ¹ Predefined datatype. ² Callback prototype definition. ³ Predefined callback routine. ⁴ Constant. Other entries are regular MPI routines. 		MPI-2.2	MPI-3.0	C language binding
 ² Callback prototype definition. ³ Predefined callback routine. ⁴ Constant. Other entries are regular MPI routines. 	MPI_CANCEL for send requests	MPI-3.2		no direct replacement
 ³ Predefined callback routine. ⁴ Constant. Other entries are regular MPI routines. 				
⁴ Constant. Other entries are regular MPI routines.	² Callback prototype definition.			
Other entries are regular MPI routines.	³ Predefined callback routine.			
	⁴ Constant.			
	Other entries are regular MPI routines.			
Table 2.1: Deprecated and Removed constructs	0			
Table 2.1: Deprecated and Removed constructs				
	Table 2.1: D_{2}	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 17.2.9.

Handles are represented in Fortran as INTEGERS, or as a BIND(C) derived type with the mpi_f08 module; see Section 2.5.1. Binary-valued variables are of type LOGICAL.

Array arguments are indexed from one.

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

2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an MPI_ prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare 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

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 17.2.4, and no others, as macros in C.

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

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

2.7 Processes

¹⁵ An MPI program consists of autonomous processes, executing their own code, in an MIMD ¹⁶ style. The codes executed by each process need not be identical. The processes communicate ¹⁷ via calls to MPI communication primitives. Typically, each process executes in its own ¹⁸ 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

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

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

Unofficial Draft for Comment Only

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

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

Several factors limit the ability of MPI calls to return with meaningful error codes when an error occurs. MPI may not be able to detect some errors; other errors may be too expensive to detect in normal execution mode; finally some errors may be "catastrophic" and may prevent MPI from returning control to the caller. On the other hand, some errors may be detected after the associated operation has completed; some errors may not have a communicator, window, or file on which an error may be raised. In such cases, these errors will be raised on the communicator MPI_COMM_SELF.

An example of such a case arises because of the nature of asynchronous communications: 23MPI calls may initiate operations that continue asynchronously after the call returned. Thus, the operation may return with a code indicating successful completion, yet later cause an error exception to be raised. If there is a subsequent call that relates to the same operation (e.g., a call that verifies that an asynchronous operation has completed) then the error 27argument 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, 2930 so that no error value can be used to indicate the nature of the error (e.g., an error on the receiver in a send with the ready mode).

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

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

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

- 444546
- 47
- 48

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22

 24

25

26

28

 31

32 33

34

35

36 37

3839

40 41

42

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.

8 9

1

2.9.1 Independence of Basic Runtime Routines

Implementation Issues

¹⁰ MPI programs require that library routines that are part of the basic language environment ¹² (such as write in Fortran and printf and malloc in ISO C) and are executed after ¹³ MPI_INIT and before MPI_FINALIZE operate independently and that their *completion* is ¹⁴ independent of the action of other processes in an MPI program.

¹⁵ Note that this in no way prevents the creation of library routines that provide parallel ¹⁶ services whose operation is collective. However, the following program is expected to com-¹⁷ plete in an ISO C environment regardless of the size of MPI_COMM_WORLD (assuming that ¹⁸ printf is available at the executing nodes).

```
<sup>19</sup> int rank;
```

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

```
<sup>21</sup> MPI_Comm_rank(MPI_COMM_WORLD, &rank);
<sup>22</sup> if (rank == 0) printf("Starting program\n");
```

```
<sup>23</sup> MPI_Finalize();
```

²⁵ 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).

30
30
31 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
32 printf("Output from task rank %d\n", rank);

³³ In addition, calls that fail because of resource exhaustion or other error are not con-³⁴ sidered a violation of the requirements here (however, they are required to complete, just ³⁵ not to complete successfully).

36 37

38

 24

2.9.2 Interaction with Signals

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

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

48

Unofficial Draft for Comment Only

2.9

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. $\frac{1}{2}$

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);
  }
                                                                                    33
  else if (myrank == 1) /* code for process one */
                                                                                    34
  {
                                                                                    35
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
                                                                                    36
      printf("received :%s:\n", message);
                                                                                    37
  }
 MPI_Finalize();
  return 0;
}
```

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

7

9

1516

17

18

19

 24

25

26

32

38

39

40

41

46

47

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

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

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

```
36 C binding
```

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

```
39
     F08 binding
40
     MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
41
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
42
         INTEGER, INTENT(IN) :: count, dest, tag
43
         TYPE(MPI_Datatype), INTENT(IN) ::
                                             datatype
44
         TYPE(MPI_Comm), INTENT(IN) :: comm
45
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
46
47
     F binding
```

```
<sup>48</sup> MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

17 18

19 20

21

22 23 24

25

<type> BUF(*)</type>					
INTEGER COUNT,	DATATYPE,	DEST,	TAG,	COMM,	IERROR

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

3.2.2 Message Data

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

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

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
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 ⁴⁵ library, with no need for additional preprocessing or compilation. Thus, one cannot ⁴⁶ assume that a communication call has information on the datatype of variables in the ⁴⁷ communication buffer; this information must be supplied by an explicit argument. ⁴⁸

Unofficial Draft for Comment Only

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		
13	MPI_UNSIGNED_SHORT	(treated as integral value)
14		unsigned short int
14		unsigned int
	MPI_UNSIGNED_LONG	unsigned long int
16	MPI_UNSIGNED_LONG_LONG	unsigned long long int
17	MPI_FLOAT	float
18	MPI_DOUBLE	double
19	MPI_LONG_DOUBLE	long double
20	MPI_WCHAR	wchar_t
21		(defined in <stddef.h>)</stddef.h>
22		(treated as printable character)
23	MPI_C_BOOL	_Bool
24	MPI_INT8_T	int8_t
25	MPI_INT16_T	int16_t
26	MPI_INT32_T	int32_t
27	MPI_INT64_T	int64_t
28	MPI_UINT8_T	uint8_t
29	MPI_UINT16_T	uint16_t
30	MPI_UINT32_T	uint32_t
31	MPI_UINT64_T	uint64_t
32	MPI_C_COMPLEX	float _Complex
33	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
34	MPI_C_DOUBLE_COMPLEX	double _Complex
35	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
36	MPI_BYTE	
37	MPI_PACKED	
38		
39		
40	Table 3.2: Predefined MPI datatypes co	prresponding to C datatypes
41		
42	The need for such datatype information will	become clear in Section 3.3.2. (End of
43	rationale.)	
44	········ /	
45	The datatypes MPI_AINT, MPI_OFFSET, and	d MPI_COUNT correspond to the MPI-
46	defined C types MPI_Aint, MPI_Offset, and MPI_C	-
47	INTEGER (KIND=MPI_ADDRESS_KIND), INTEGER (I	-
48	(KIND=MPI_COUNT_KIND). This is described in Ta	
		I J I I I I I I I I I I I I I I I I I I

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

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

are available in all language bindings. See Sections 17.2.6 and 17.2.10 on page 674 and 682 for information on interlanguage communication with these types.

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

MPI datatype	C++ datatype
MPI_CXX_BOOL	bool
MPI_CXX_FLOAT_COMPLEX	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 MP	I datatypes corresp	bonding to $C++$	datatypes
--------------------------	---------------------	------------------	-----------

3.2.3 Message Envelope

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

source	
destination	
tag	
communicator	

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

The message destination is specified by the dest argument.

The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,..., 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 ⁴⁷ context. This **process group** is ordered and processes are identified by their rank within ⁴⁸

Unofficial Draft for Comment Only

 24

 $45 \\ 46$

¹ this group. Thus, the range of valid values for dest is $0, \ldots, n-1 \cup \{\text{MPI_PROC_NULL}\}$, where ² *n* is the number of processes in the group. (If the communicator is an inter-communicator, ³ 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.

Advice to users. Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing communication libraries, need only use the predefined variable MPI_COMM_WORLD as the comm argument. This will allow communication with all the processes available at initialization time.

- ¹³ Users may define new communicators, as explained in Chapter 6. Communicators ¹⁴ provide an important encapsulation mechanism for libraries and modules. They allow ¹⁵ modules to have their own disjoint communication universe and their own process ¹⁶ numbering scheme. (*End of advice to users.*)
 - Advice to implementors. The message envelope would normally be encoded by a fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (*End of advice to implementors.*)
- 22 23 24

25

18

19

20

21

7

8 9

10

11

12

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

26 27 28

29

MPI_RECV(buf, count, datatype, source, tag, comm, status)

30	OUT	buf	initial address of receive buffer (choice)
31 32	IN	count	number of elements in receive buffer (non-negative in-teger)
33 34	IN	datatype	datatype of each receive buffer element (handle)
35 36	IN	source	rank of source or MPI_ANY_SOURCE (non-negative integer)
37	IN	tag	message tag or $MPI_ANY_TAG\xspace$ (integer)
38 39	IN	comm	communicator (handle)
40 41	OUT	status	status object (Status)
42	C bindin	g	
43 44	int MPI_H		, MPI_Datatype datatype, int source, mm, MPI_Status *status)
45 46	F08 bind	0	
47		v 1	nrce, tag, comm, status, ierror)
48	TYPE	(*), DIMENSION() :: bu	11

```
INTEGER, INTENT(IN) ::
                                                                                             1
                                count, source, tag
                                                                                             2
    TYPE(MPI_Datatype), INTENT(IN) ::
                                             datatype
                                                                                             3
    TYPE(MPI_Comm), INTENT(IN) ::
                                         comm
    TYPE(MPI_Status) :: status
                                                                                             4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                             5
                                                                                             6
F binding
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
                                                                                             9
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
                                                                                             10
    IERROR
                                                                                             11
    The blocking semantics of this call are described in Section 3.4.
                                                                                             12
                                                                                             13
    The receive buffer consists of the storage containing count consecutive elements of the
                                                                                             14
type specified by datatype, starting at address buf. The length of the received message must
                                                                                             15
be less than or equal to the length of the receive buffer. An overflow error occurs if all
                                                                                             16
incoming data does not fit, without truncation, into the receive buffer.
                                                                                             17
    If a message that is shorter than the receive buffer arrives, then only those locations
                                                                                             18
corresponding to the (shorter) message are modified.
                                                                                             19
     Advice to users. The MPI_PROBE function described in Section 3.8 can be used to
                                                                                             20
     receive messages of unknown length. (End of advice to users.)
                                                                                             21
                                                                                            22
     Advice to implementors. Even though no specific behavior is mandated by MPI for
                                                                                            23
     erroneous programs, the recommended handling of overflow situations is to return in
                                                                                             ^{24}
     status information about the source and tag of the incoming message. The receive
                                                                                             25
     operation will return an error code. A quality implementation will also ensure that
                                                                                             26
     no memory that is outside the receive buffer will ever be overwritten.
                                                                                            27
                                                                                             28
     In the case of a message shorter than the receive buffer, MPI is quite strict in that it
                                                                                             29
     allows no modification of the other locations. A more lenient statement would allow
                                                                                             30
     for some optimizations but this is not allowed. The implementation must be ready to
                                                                                             31
     end a copy into the receiver memory exactly at the end of the receive buffer, even if
                                                                                             32
     it is an odd address. (End of advice to implementors.)
                                                                                             33
                                                                                            34
    The selection of a message by a receive operation is governed by the value of the
                                                                                             35
message envelope. A message can be received by a receive operation if its envelope matches
                                                                                            36
the source, tag and comm values specified by the receive operation. The receiver may
                                                                                            37
specify a wildcard MPI_ANY_SOURCE value for source, and/or a wildcard MPI_ANY_TAG
                                                                                             38
```

value for tag, indicating that any source and/or tag are acceptable. It cannot specify a 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 source unless source=MPI_ANY_SOURCE in the pattern, and has a matching tag unless tag=MPI_ANY_TAG in the pattern.

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

Unofficial Draft for Comment Only

39

40

41

42

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

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

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

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

¹⁸ 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,

⁴¹ MPI_ERROR, and TYPE(MPI_Status) are defined to allow conversion between both status ⁴² representations. Conversion routines are provided in Section 17.2.5.

- 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.*)

6

 $\overline{7}$

8

9

10

11

12

13

14 15

16

3.2. BLOCKING SEND AND RECEIVE OPERATIONS

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

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

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

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 (non-negative integer)

C binding

F08 binding

Ν

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

F binding

```
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.

 $\overline{7}$

 24

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. 10

Rationale. Zero-length datatypes may be created in a number of cases. An important case is MPI_TYPE_CREATE_DARRAY, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style will not check for this special case and may want to use MPI_GET_COUNT to check the status. (End of rationale.)

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

All send and receive operations use the buf, count, datatype, source, dest, tag, comm, and status arguments in the same way as the blocking MPI_SEND and MPI_RECV operations described in this section.

26 Passing MPI_STATUS_IGNORE for Status 3.2.6 27

28Every call to MPI_RECV includes a status argument, wherein the system can return details 29about the message received. There are also a number of other MPI calls where status 30 is returned. An object of type MPI_Status is not an MPI opaque object; its structure 31 is declared in mpi.h and mpif.h, and it exists in the user's program. In many cases, 32 application programs are constructed so that it is unnecessary for them to examine the 33 status fields. In these cases, it is a waste for the user to allocate a status object, and it is 34particularly wasteful for the MPI implementation to fill in fields in this object.

35 To cope with this problem, there are two predefined constants, MPI_STATUS_IGNORE 36 and MPI_STATUSES_IGNORE, which when passed to a receive, probe, wait, or test function, 37 inform the implementation that the status fields are not to be filled in. Note that

38MPI_STATUS_IGNORE is not a special type of MPI_Status object; rather, it is a special value 39 for the argument. In C one would expect it to be NULL, not the address of a special 40MPI_Status.

41 MPI_STATUS_IGNORE, and the array version MPI_STATUSES_IGNORE, can be used every-42where a status argument is passed to a receive, wait, or test function. MPI_STATUS_IGNORE 43cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE 44and MPI_STATUSES_IGNORE are objects like MPI_BOTTOM (not usable for initialization or 45assignment). See Section 2.5.4.

46 In general, this optimization can apply to all functions for which status or an array of 47statuses is an OUT argument. Note that this converts status into an INOUT argument. The 48functions that can be passed MPI_STATUS_IGNORE are all the various forms of MPI_RECV,

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

11

12

13

14

15

1617

18

19

20

2122

23

 24

MPI_PROBE, MPI_TEST, and MPI_WAIT, as well as MPI_REQUEST_GET_STATUS. When an array is passed, as in the MPI_{TEST|WAIT}{ALL|SOME} functions, a separate constant, MPI_STATUSES_IGNORE, is passed for the array argument. It is possible for an MPI function to return MPI_ERR_IN_STATUS even when MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE has been passed to that function.

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

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

3.3 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.

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

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

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
         • Communication of typed values (e.g., with datatype different from MPI_BYTE), where
\mathbf{2}
           the datatypes of the corresponding entries in the sender program, in the send call, in
3
           the receive call and in the receiver program must all match.
4
         • Communication of untyped values (e.g., of datatype MPI_BYTE), where both sender
5
           and receiver use the datatype MPI_BYTE. In this case, there are no requirements on
6
           the types of the corresponding entries in the sender and the receiver programs, nor is
7
           it required that they be the same.
8
9
         • Communication involving packed data, where MPI_PACKED is used.
10
11
          The following examples illustrate the first two cases.
12
                      Sender and receiver specify matching types.
     Example 3.1
13
14
     CALL MPI_COMM_RANK(comm, rank, ierr)
15
     IF (rank.EQ.0) THEN
16
          CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
17
     ELSE IF (rank.EQ.1) THEN
18
          CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
19
     END IF
20
21
          This code is correct if both a and b are real arrays of size > 10. (In Fortran, it might be
22
     correct to use this code even if a or b have size < 10: e.g., when a(1) can be equivalenced
23
     to an array with ten reals.)
^{24}
                      Sender and receiver do not specify matching types.
25
     Example 3.2
26
     CALL MPI_COMM_RANK(comm, rank, ierr)
27
     IF (rank.EQ.0) THEN
28
          CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
29
     ELSE IF (rank.EQ.1) THEN
30
          CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
^{31}
     END IF
32
33
          This code is erroneous, since sender and receiver do not provide matching datatype
34
     arguments.
35
36
     Example 3.3
                      Sender and receiver specify communication of untyped values.
37
38
     CALL MPI_COMM_RANK(comm, rank, ierr)
39
     IF (rank.EQ.0) THEN
40
          CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
41
     ELSE IF (rank.EQ.1) THEN
42
          CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
43
     END IF
44
          This code is correct, irrespective of the type and size of a and b (unless this results in
45
     an out of bounds memory access).
46
47
48
```

CHAPTER 3. POINT-TO-POINT COMMUNICATION

Advice to users. If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND, then MPI will send the data stored at contiguous locations, starting from the address indicated by the buf argument. This may have unexpected results when the data layout is not as a casual user would expect it to be. For example, some Fortran compilers implement variables of type CHARACTER as a structure that contains the character length and a pointer to the actual string. In such an environment, sending and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have the anticipated result of transferring the character string. For this reason, the user is advised to use typed communications whenever possible. (*End of advice to users.*)

Type MPI_CHARACTER

The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER, rather than the entire character string stored in the variable. Fortran variables of type CHARACTER or substrings are transferred as if they were arrays of characters. This is illustrated in the example below.

Example 3.4

Transfer of Fortran CHARACTERs.

```
CHARACTER*10 a
CHARACTER*10 b
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
```

END IF

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

Rationale. The alternative choice would be for MPI_CHARACTER to match a character of arbitrary length. This runs into problems.

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

This problem forces us to provide explicit information on character length with the MPI call. One could add a length parameter to the type MPI_CHARACTER, but this does not add much convenience and the same functionality can be achieved by defining a suitable derived datatype. (*End of rationale.*) 47

 $\mathbf{5}$

 31

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

3.3.2 Data Conversion

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

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

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

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical and character values, and to convert a floating point value to the nearest value that can be represented on the target system.

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

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

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

Consider the three examples, 3.1-3.3. The first program is correct, assuming that **a** and 35 b are REAL arrays of size ≥ 10 . If the sender and receiver execute in different environments, 36 then the ten real values that are fetched from the send buffer will be converted to the 37 representation for reals on the receiver site before they are stored in the receive buffer. 38 While the number of real elements fetched from the send buffer equal the number of real 39 elements stored in the receive buffer, the number of bytes stored need not equal the number 40 of bytes loaded. For example, the sender may use a four byte representation and the receiver 41 an eight byte representation for reals. 42

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

Unofficial Draft for Comment Only

1

2

3

4

5 6

7

8

9

10 11

12 13

14

in the receive buffer may encode values that are different from the values they encoded in the send buffer.

Data representation conversion also applies to the envelope of a message: source, destination 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.*)

MPI requires support for inter-language communication, i.e., if messages are sent by a C or C++ process and received by a Fortran process, or vice-versa. The behavior is defined in Section 17.2.

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

Unofficial Draft for Comment Only

 $\overline{7}$

 24

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.

6 A **buffered** mode send operation can be started whether or not a matching receive $\overline{7}$ 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 8 9 of a matching receive. Thus, if a send is executed and no matching receive is posted, then 10 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 11 by the user — see Section 3.6. Buffer allocation by the user may be required for the buffered 12mode to be effective. 13

A send that uses the **synchronous** mode can be started whether or not a matching 14receive was posted. However, the send will complete successfully only if a matching receive is 1516posted, and the receive operation has started to receive the message sent by the synchronous 17send. 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 18 19execution, 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 2021communication 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. 22

A send that uses the **ready** communication mode may be started *only* if the matching 23receive is already posted. Otherwise, the operation is erroneous and its outcome is unde- 24 25fined. On some systems, this allows the removal of a hand-shake operation that is otherwise 26required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer 27can be reused. A send operation that uses the ready mode has the same semantics as a 28standard send operation, or a synchronous send operation; it is merely that the sender 29provides additional information to the system (namely that a matching receive is already 30 posted), that can save some overhead. In a correct program, therefore, a ready send could 31 32 be replaced by a standard send with no effect on the behavior of the program other than performance. 33

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.

- 37
- 38
- 39 40
- 41
- 42

43

- 44
- 45
- 46
- 47
- 48

1

2

3

4

MPI_BSEND(buf, count, datatype, dest, tag, comm) ¹					
IN	buf	initial address of send buffer (choice)	2		
IN	count	number of elements in send buffer (non-negative inte- ger)	3 4 5		
IN	datatype	datatype of each send buffer element (handle)	6		
IN		rank of destination (non-negative integer)	7		
IN		message tag (integer)	8		
	0		9 10		
IN	comm	communicator (handle)	10		
Сb	inding		12		
	C binding int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)				
F08	binding		15 16		
	e e	type, dest, tag, comm, ierror)	10		
	<pre>TYPE(*), DIMENSION(),</pre>		18		
	<pre>INTEGER, INTENT(IN) ::</pre>		19		
	TYPE(MPI_Datatype), INT TYPE(MPI_Comm), INTENT(20		
	INTEGER, OPTIONAL, INTE		21		
			22 23		
	inding	VDE DECT TAC COMM TEDDOD)	24		
MP1_	<pre>desend(BOF, COUNT, DATA) <type> BUF(*)</type></pre>	TYPE, DEST, TAG, COMM, IERROR)	25		
	INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR				
	27				
	Send in buffered mode. 24				
	3				
MPI	_SSEND(buf, count, datatyp	e, dest, tag, comm)	31		
IN	buf	initial address of send buffer (choice)	32		
IN	count	number of elements in send buffer (non-negative integer)	33 34		
IN	datatype	datatype of each send buffer element (handle)	35 36		
IN	dest	rank of destination (non-negative integer)	37		
IN	tag	message tag (integer)	38		
IN	comm	communicator (handle)	39		
			40		
C binding					
int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest,			43		
int tag, MPI_Comm comm) 44					
F08 binding 45					
MPI_		zype, dest, tag, comm, ierror)	46		
	<pre>TYPE(*), DIMENSION(),</pre>		47		
	INTEGER, INTENT(IN) :: count, dest, tag 48				

```
1
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                  datatype
\mathbf{2}
          TYPE(MPI_Comm), INTENT(IN) :: comm
3
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
4
     F binding
5
     MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
6
          <type> BUF(*)
7
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
8
9
          Send in synchronous mode.
10
11
     MPI_RSEND(buf, count, datatype, dest, tag, comm)
12
13
       IN
                 buf
                                              initial address of send buffer (choice)
14
       IN
                 count
                                              number of elements in send buffer (non-negative inte-
15
                                              ger)
16
       IN
                 datatype
                                              datatype of each send buffer element (handle)
17
18
       IN
                 dest
                                              rank of destination (non-negative integer)
19
       IN
                                              message tag (integer)
                 tag
20
       IN
                 comm
                                              communicator (handle)
21
22
23
     C binding
^{24}
     int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest,
25
                     int tag, MPI_Comm comm)
26
     F08 binding
27
     MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
28
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
29
          INTEGER, INTENT(IN) :: count, dest, tag
30
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
^{31}
          TYPE(MPI_Comm), INTENT(IN) :: comm
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     F binding
35
     MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
36
          <type> BUF(*)
37
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
38
          Send in ready mode.
39
          There is only one receive operation, but it matches any of the send modes. The receive
40
     operation described in the last section is blocking: it returns only after the receive buffer
41
     contains the newly received message. A receive can complete before the matching send has
42
     completed (of course, it can complete only after the matching send has started).
43
          In a multithreaded implementation of MPI, the system may de-schedule a thread that
44
     is blocked on a send or receive operation, and schedule another thread for execution in
45
     the same address space. In such a case it is the user's responsibility not to modify a
46
     communication buffer until the communication completes. Otherwise, the outcome of the
47
     computation is undefined.
48
```

	Advice to implementors. Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation.	1 2
	It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the sender until a matching receive occurs by using the synchronous send mode.	3 4 5 6
	A possible communication protocol for the various communication modes is outlined below.	7 8
	ready send: The message is sent as soon as possible.	9
	<i>synchronous send</i> : The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.	10 11 12 13
	<i>standard send</i> : First protocol may be used for short messages, and second protocol for long messages.	14 15
	<i>buffered send</i> : The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).	16 17
	Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.	18 19 20
	Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.	20 21 22
	A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, users may expect some buffering.	23 24
	In a multithreaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. (<i>End of advice to implementors.</i>)	25 26 27 28
3.5	Semantics of Point-to-Point Communication	29 30
	alid MPI implementation guarantees certain general properties of point-to-point com- ication, which are described in this section.	31 32 33 34
secor and	e destination, and both match the same receive, then this operation cannot receive the nd message if the first one is still pending. If a receiver posts two receives in succession, both match the same message, then the second receive operation cannot be satisfied	35 36 37 38
sends single	his message, if the first one is still pending. This requirement facilitates matching of s to receives. It guarantees that message-passing code is deterministic, if processes are e-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the described later, such as MPI_CANCEL or MPI_WAITANY, are additional sources of	39 40 41
	leterminism.)	42 43
	If a process has a single thread of execution, then any two communications executed his process are ordered. On the other hand, if the process is multithreaded, then the	44 45

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 48

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

```
3
     order.
4
     Example 3.5
                      An example of non-overtaking messages.
5
6
     CALL MPI_COMM_RANK(comm, rank, ierr)
7
     IF (rank.EQ.0) THEN
8
          CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
9
          CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
10
     ELSE IF (rank.EQ.1) THEN
11
          CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
12
          CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
13
     END IF
14
15
     The message sent by the first send must be received by the first receive, and the message
16
     sent by the second send must be received by the second receive.
17
18
     Progress If a pair of matching send and receives have been initiated on two processes, then
19
     at least one of these two operations will complete, independently of other actions in the
20
     system: the send operation will complete, unless the receive is satisfied by another message,
21
     and completes; the receive operation will complete, unless the message sent is consumed by
22
     another matching receive that was posted at the same destination process.
23
^{24}
                      An example of two, intertwined matching pairs.
     Example 3.6
25
26
     CALL MPI_COMM_RANK(comm, rank, ierr)
27
     IF (rank.EQ.0) THEN
28
          CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
29
          CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
30
     ELSE IF (rank.EQ.1) THEN
^{31}
          CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
32
          CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
33
     END IF
34
     Both processes invoke their first communication call. Since the first send of process zero
35
     uses the buffered mode, it must complete, irrespective of the state of process one. Since
36
     no matching receive is posted, the message will be copied into buffer space. (If insufficient
37
     buffer space is available, then the program will fail.) The second send is then invoked. At
38
     that point, a matching pair of send and receive operation is enabled, and both operations
39
     must complete. Process one next invokes its second receive call, which will be satisfied by
40
     the buffered message. Note that process one received the messages in the reverse order they
41
     were sent.
42
43
^{44}
     Fairness MPI makes no guarantee of fairness in the handling of communication. Suppose
45
     that a send is posted. Then it is possible that the destination process repeatedly posts a
46
     receive that matches this send, yet the message is never received, because it is each time
47
     overtaken by another message, sent from another source. Similarly, suppose that a receive
```

⁴⁸ was posted by a multithreaded process. Then it is possible that messages that match this

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

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

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

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

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

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

Example 3.7 An exchange of messages.

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

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

Example 3.8 An errant attempt to exchange messages.

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

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

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

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

3.6 Buffer Allocation and Usage

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

MPI_BUFFER_ATTACH(buffer, size)

IN	buffer	initial buffer address (choice)
IN	size	buffer size, in bytes (non-negative integer)

C binding

int MPI_Buffer_attach(void *buffer, int size)

```
F08 binding
MPI_Buffer_attach(buffer, size, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time. In C, **buffer** is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 17.1.12).

MPI_BUFFER_DETACH(buffer_addr, size)			40
OUT	buffer_addr	initial buffer address	41
OUT	size	buffer size, in bytes (non-negative integer)	42
			43
C binding			
			45
<pre>int MPI_Buffer_detach(void *buffer_addr, int *size)</pre>			46
F08 binding			47
MPI_Buffer_detach(buffer_addr, size, ierror)			48

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

```
1
          USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
\mathbf{2}
          TYPE(C_PTR), INTENT(OUT) :: buffer_addr
3
          INTEGER, INTENT(OUT) :: size
4
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
5
     F binding
6
     MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
7
          INTEGER(KIND=MPI_ADDRESS_KIND) BUFFER_ADDR
8
          INTEGER SIZE, IERROR
9
10
          Detach the buffer currently associated with MPI. The call returns the address and the
11
     size of the detached buffer. This operation will block until all messages currently in the
12
      buffer have been transmitted. Upon return of this function, the user may reuse or deallocate
13
      the space taken by the buffer.
14
     Example 3.10 Calls to attach and detach buffers.
15
16
      #define BUFFSIZE 10000
17
      int size;
18
     char *buff;
19
     MPI_Buffer_attach(malloc(BUFFSIZE), BUFFSIZE);
20
      /* a buffer of 10000 bytes can now be used by MPI_Bsend */
21
     MPI_Buffer_detach(&buff, &size);
22
      /* Buffer size reduced to zero */
23
     MPI_Buffer_attach(buff, size);
24
     /* Buffer of 10000 bytes available again */
25
26
                              Even though the C functions MPI_Buffer_attach and
           Advice to users.
27
           MPI_Buffer_detach both have a first argument of type void<sup>*</sup>, these arguments are used
28
           differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address of the
29
           pointer is passed to MPI_Buffer_detach, so that this call can return the pointer value.
30
           In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument is
31
           wrongly defined and the argument is therefore unused. In Fortran with the mpi_f08
32
           module, the address of the buffer is returned as TYPE(C_PTR), see also Example 8.1
33
           about the use of C_PTR pointers. (End of advice to users.)
34
35
                        Both arguments are defined to be of type void* (rather than
           Rationale.
36
           void<sup>*</sup> and void<sup>**</sup>, respectively), so as to avoid complex type casts. E.g., in the last
37
           example, &buff, which is of type char**, can be passed as argument to
38
           MPI_Buffer_detach without type casting. If the formal parameter had type void**
39
           then we would need a type cast before and after the call. (End of rationale.)
40
41
          The statements made in this section describe the behavior of MPI for buffered-mode
42
     sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is
43
     associated with the process.
44
          MPI must provide as much buffering for outgoing messages as if outgoing message
45
      data were buffered by the sending process, in the specified buffer space, using a circular,
46
      contiguous-space allocation policy. We outline below a model implementation that defines
47
      this policy. MPI may provide more buffering, and may use a better buffer allocation algo-
48
      rithm than described below. On the other hand, MPI may signal an error whenever the
```

simple buffering allocator described below would run out of space. In particular, if no buffer is explicitly associated with the process, then any buffered send may cause an error.

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

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

3.6.1 Model Implementation of Buffered Mode

The model implementation uses the packing and unpacking functions described in Section 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.

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

1 2

3.7 Nonblocking Communication

One can improve performance on many systems by overlapping communication and com-3 putation. This is especially true on systems where communication can be executed au-4 tonomously by an intelligent communication controller. Light-weight threads are one mech-5anism for achieving such overlap. An alternative mechanism that often leads to better 6 performance is to use **nonblocking communication**. A nonblocking **send start** call ini-7 tiates the send operation, but does not complete it. The send start call can return before 8 the message was copied out of the send buffer. A separate send complete call is needed 9 to complete the communication, i.e., to verify that the data has been copied out of the send 10 buffer. With suitable hardware, the transfer of data out of the sender memory may proceed 11 concurrently with computations done at the sender after the send was initiated and before it 12completed. Similarly, a nonblocking **receive start call** initiates the receive operation, but 13 does not complete it. The call can return before a message is stored into the receive buffer. 14A separate **receive complete** call is needed to complete the receive operation and verify 15that the data has been received into the receive buffer. With suitable hardware, the transfer 16of data into the receiver memory may proceed concurrently with computations done after 17the receive was initiated and before it completed. The use of nonblocking receives may also 18 avoid system buffering and memory-to-memory copying, as information is provided early 19on the location of the receive buffer. 20

Nonblocking send start calls can use the same four modes as blocking sends: standard, 21buffered, synchronous and ready. These carry the same meaning. Sends of all modes, ready 22excepted, can be started whether a matching receive has been posted or not; a nonblocking 23ready send can be started only if a matching receive is posted. In all cases, the send start 24 call is local: it returns immediately, irrespective of the status of other processes. If the call 25causes some system resource to be exhausted, then it will fail and return an error code. 26Quality implementations of MPI should ensure that this happens only in "pathological" 27cases. That is, an MPI implementation should be able to support a large number of pending 28nonblocking operations. 29

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

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

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

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

 $45 \\ 46$

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.*)

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.

34

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15

16

17

18

19 20

1 MPI_ISEND(buf, count, datatype, dest, tag, comm, request) $\mathbf{2}$ IN buf initial address of send buffer (choice) 3 IN count number of elements in send buffer (non-negative inte-4 ger) 56 IN datatype datatype of each send buffer element (handle) 7 dest rank of destination (non-negative integer) IN 8 IN tag message tag (integer) 9 10 IN comm communicator (handle) 11 OUT request communication request (handle) 1213 C binding 14 int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, 15int tag, MPI_Comm comm, MPI_Request *request) 1617F08 binding 18MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) 19TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 20INTEGER, INTENT(IN) :: count, dest, tag 21TYPE(MPI_Datatype), INTENT(IN) :: datatype 22TYPE(MPI_Comm), INTENT(IN) :: comm 23TYPE(MPI_Request), INTENT(OUT) :: request 24 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25F binding 26MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 27<type> BUF(*) 28INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 29 30 Start a standard mode, nonblocking send. 31 32 MPI_IBSEND(buf, count, datatype, dest, tag, comm, request) 33 34IN buf initial address of send buffer (choice) 35 IN count number of elements in send buffer (non-negative inte-36 ger) 37 IN datatype datatype of each send buffer element (handle) 38 39 IN dest rank of destination (non-negative integer) 40IN message tag (integer) tag 41 42IN comm communicator (handle) 43 OUT communication request (handle) request 44 45C binding 46 int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest, 47int tag, MPI_Comm comm, MPI_Request *request) 48

F08 binding				
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)				
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag				
	(MPI_Datatype), INTENT(IN)	•	4 5	
	(MPI_Datatype), INTENI(IN) ::		6	
	(MPI_Request), INTENT(IN)		7	
	GER, OPTIONAL, INTENT(OUT)	-	8	
			9	
F binding			10	
		DEST, TAG, COMM, REQUEST, IERROR)	11	
• 1	e> BUF(*)		12	
INTE	ER CUUNT, DATATYPE, DEST	, TAG, COMM, REQUEST, IERROR	13	
Start	a buffered mode, nonblocking	send.	14	
			15	
	ND(buf, count, datatype, dest,	tag comm request)	16	
		c , ,	17	
IN	buf	initial address of send buffer (choice)	18 19	
IN	count	number of elements in send buffer (non-negative inte-	20	
		ger)	20	
IN	datatype	datatype of each send buffer element (handle) (handle)	22	
IN	dest	rank of destination (non-negative integer)	23	
			24	
IN	tag	message tag (integer)	25	
IN	comm	communicator (handle)	26	
OUT	request	communication request (handle)	27	
			28	
C binding			29	
int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest,			30	
int tag, MPI_Comm comm, MPI_Request *request)			31	
F08 bind	ing		32 33	
	0	dest, tag, comm, request, ierror)	34	
		T(IN), ASYNCHRONOUS :: buf	35	
	GER, INTENT(IN) :: count		36	
	(MPI_Datatype), INTENT(IN		37	
	(MPI_Comm), INTENT(IN) ::		38	
TYPE	(MPI_Request), INTENT(OUT)) :: request	39	
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	40	
F binding	o.		41	
		DEST, TAG, COMM, REQUEST, IERROR)	42	
	<pre>> BUF(*)</pre>		43	
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR			44	
			45	
Start a synchronous mode, nonblocking send.			46 47	
			47	
10				

1 MPI_IRSEND(buf, count, datatype, dest, tag, comm, request) $\mathbf{2}$ IN buf initial address of send buffer (choice) 3 IN number of elements in send buffer (non-negative intecount 4 ger) 56 IN datatype datatype of each send buffer element (handle) 7 IN dest rank of destination (non-negative integer) 8 IN message tag (integer) tag 9 10 IN comm communicator (handle) 11 OUT request communication request (handle) 1213 C binding 14int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest, 15int tag, MPI_Comm comm, MPI_Request *request) 1617F08 binding 18MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) 19TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf INTEGER, INTENT(IN) :: count, dest, tag 20TYPE(MPI_Datatype), INTENT(IN) :: datatype 21TYPE(MPI_Comm), INTENT(IN) :: comm 22 23TYPE(MPI_Request), INTENT(OUT) :: request 24 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 25F binding 26MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 27<type> BUF(*) 28INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 29 30 Start a ready mode nonblocking send. 31 32 MPI_IRECV(buf, count, datatype, source, tag, comm, request) 33 34OUT buf initial address of receive buffer (choice) 35 IN number of elements in receive buffer (non-negative incount 36 teger) 37 IN datatype datatype of each receive buffer element (handle) 38 IN rank of source or MPI_ANY_SOURCE (non-negative 39 source 40integer) 41 IN message tag or MPI_ANY_TAG (integer) tag 42IN communicator (handle) comm 43 OUT request communication request (handle) 44 4546C binding 47int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, 48 int tag, MPI_Comm comm, MPI_Request *request)

F08 binding
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count, source, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
F binding
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
<pre><type> BUF(*)</type></pre>
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
INTEGER COONT, DATATIFE, SOONCE, TAG, COMM, REQUEST, TERROR
Start a nonblocking receive.
These calls allocate a communication request object and associate it with the request

These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.

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

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10–17.1.20. (End of advice to users.)

3.7.3 Communication Completion

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

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

We shall use the following terminology: A **null handle** is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are **inactive** if the request is not associated with any ongoing communication (see Section 3.9). A handle is **active** if it is neither null nor inactive. An **empty** status is a status which is set to return **tag** = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and is also internally configured so that calls to MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X return count = 0 and MPI_TEST_CANCELLED returns false. We

 24

set a status variable to empty when the value returned by it is not significant. Status is set
 in this way so as to prevent errors due to accesses of stale information.
 The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any

of the other derived functions (MPI_{TEST|WAIT}{ALL|SOME|ANY}), where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI_ERR_IN_STATUS; and the returned status can be queried by the call MPI_TEST_CANCELLED.

⁸ Error codes belonging to the error class MPI_ERR_IN_STATUS should be returned only ⁹ by the MPI completion functions that take arrays of MPI_Status. For the functions that ¹⁰ take a single MPI_Status argument, the error code is returned by the function, and the value ¹¹ of the MPI_ERROR field in the MPI_Status argument is undefined (see 3.2.5).

```
<sup>13</sup>
14 MPI_WAIT(request, status)
```

12

```
15
       INOUT
                 request
                                            request (handle)
16
       OUT
                                            status object (Status)
                status
17
18
     C binding
19
     int MPI_Wait(MPI_Request *request, MPI_Status *status)
20
21
     F08 binding
22
     MPI_Wait(request, status, ierror)
23
         TYPE(MPI_Request), INTENT(INOUT) :: request
^{24}
         TYPE(MPI_Status) :: status
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     F binding
27
     MPI_WAIT(REQUEST, STATUS, IERROR)
28
         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
29
30
^{31}
```

³⁰ A call to MPI_WAIT returns when the operation identified by request is complete. If the ³¹ request is an active persistent request, it is marked inactive. Any other type of request is ³² deallocated and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local ³³ 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.*)

MPI_TEST(request, flag, status)				
INOUT	request	communication request (handle)		
OUT	flag	true if operation completed (logical)		
OUT	status	status object (Status)		

C binding

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

F08 binding

MPI_Test(request, flag, status, ierror)
TYPE(MPI_Request), INTENT(INOUT) :: request
LOGICAL, INTENT(OUT) :: flag
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

```
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
```

A call to MPI_TEST returns flag = true if the operation identified by request is complete. In such a case, the status object is set to contain information on the completed operation. If the request is an active persistent request, it is marked as inactive. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns flag = false if the operation identified by request is not complete. In this case, the value of the status object is undefined. MPI_TEST is a local operation.

The return status object for a receive operation carries information that can be accessed as described in Section 3.2.5. The status object for a send operation carries information that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).

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

The functions MPI_WAIT and MPI_TEST can be used to complete both sends and receives.

Advice to users. The use of the nonblocking MPI_TEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be emulated with periodic calls to MPI_TEST. (*End of advice to users.*)

Example 3.11 Simple usage of nonblocking operations and MPI_WAIT.

Unofficial Draft for Comment Only

 24

 31

```
1
     CALL MPI_COMM_RANK(comm, rank, ierr)
\mathbf{2}
     IF (rank.EQ.0) THEN
3
          CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
4
          **** do some computation to mask latency ****
5
          CALL MPI_WAIT(request, status, ierr)
6
     ELSE IF (rank.EQ.1) THEN
7
          CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
8
          **** do some computation to mask latency ****
9
          CALL MPI_WAIT(request, status, ierr)
10
     END IF
11
         A request object can be deallocated by using the following operation.
12
13
14
     MPI_REQUEST_FREE(request)
15
       INOUT
                                             communication request (handle)
                 request
16
17
     C binding
18
     int MPI_Request_free(MPI_Request *request)
19
20
     F08 binding
21
     MPI_Request_free(request, ierror)
22
          TYPE(MPI_Request), INTENT(INOUT) :: request
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     F binding
25
     MPI_REQUEST_FREE(REQUEST, IERROR)
26
          INTEGER REQUEST, IERROR
27
28
          MPI_REQUEST_FREE is a local operation that marks the request object for deallo-
29
     cation and sets request to MPI_REQUEST_NULL. Ongoing communication, if any, that is
30
     associated with the request will be allowed to complete. The request will be deallocated
^{31}
     only after its completion. Classes of operations described later in the standard, such as
32
     nonblocking collective and persistent collective (see Chapters 5 and 7), also use request ob-
33
     jects. In the case of nonblocking collective operations and persistent collective operations,
34
     it is erroneous to call MPI_REQUEST_FREE unless the request is inactive.
35
                       For point-to-point operations, the MPI_REQUEST_FREE mechanism is
36
           Rationale.
37
           provided for reasons of performance and convenience on the sending side. (End of
38
           rationale.)
39
           Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not
40
           possible to check for the successful completion of the associated communication with
41
           calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the
42
           communication, an error code cannot be returned to the user — such an error must
43
           be treated as fatal. An active receive request should never be freed as the receiver
44
           will have no way to verify that the receive has completed and the receive buffer can
45
           be reused. (End of advice to users.)
46
47
48
     Example 3.12
                        An example using MPI_REQUEST_FREE.
```

CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)	1
IF (rank.EQ.0) THEN	2
DO i=1, n	3
CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)	4
CALL MPI_REQUEST_FREE(req, ierr)	5
CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)	6
CALL MPI_WAIT(req, status, ierr)	7
END DO	8
ELSE IF (rank.EQ.1) THEN	9
CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	10
CALL MPI_WAIT(req, status, ierr)	11
DO I=1, n-1	12
CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	13
CALL MPI_REQUEST_FREE(req, ierr)	14
CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	15
CALL MPI_WAIT(req, status, ierr)	16
END DO	17
CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)	18
CALL MPI_WAIT(req, status, ierr)	19
END IF	20
	21

3.7.4 Semantics of Nonblocking Communications

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

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

Example 3.13 Message ordering for nonblocking operations.

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

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

46**Progress** A call to MPI_WAIT that completes a receive will eventually terminate and return 47if a matching send has been started, unless the send is satisfied by another receive. In 48 particular, if the matching send is nonblocking, then the receive should complete even if no

Unofficial Draft for Comment Only

22

23

24

2526

27

28

2930

 31 32

33

34

35

36

37

38

39

40

41

42

43

1call is executed by the sender to complete the send. Similarly, a call to MPI_WAIT that $\mathbf{2}$ completes a send will eventually return if a matching receive has been started, unless the 3 receive is satisfied by another send, and even if no call is executed to complete the receive. 4

```
Example 3.14
                An illustration of progress semantics.
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
      CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
      CALL MPI_WAIT(r, status, ierr)
END IF
```

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

20If an MPI_TEST that completes a receive is repeatedly called with the same arguments, 21and a matching send has been started, then the call will eventually return flag = true, unless 22the send is satisfied by another receive. If an MPI_TEST that completes a send is repeatedly 23called with the same arguments, and a matching receive has been started, then the call will 24 eventually return flag = true, unless the receive is satisfied by another send.

2526

27

3.7.5 **Multiple Completions**

It is convenient to be able to wait for the completion of any, some, or all the operations 28in a list, rather than having to wait for a specific message. A call to MPI_WAITANY or 29 MPI_TESTANY can be used to wait for the completion of one out of several operations. A 30 call to MPI_WAITALL or MPI_TESTALL can be used to wait for all pending operations in 31 a list. A call to MPI_WAITSOME or MPI_TESTSOME can be used to complete all enabled 32 operations in a list. 33

3435

36

MPI_WAITANY(count, array_of_requests, index, status)

37	IN	count	list length (integer)
38	INOUT	array_of_requests	array of requests (array of handles)
39 40	OUT	index	index of handle for operation that completed (integer)
41	OUT	status	status object (Status)
42			
43	C bindin	g	
44	int MPI_V	Vaitany(int count, MF	PI_Request array_of_requests[], int *index,
45		MDT Ctatua watat	+ua)

```
MPI_Status *status)
```

```
46
      F08 binding
47
```

MPI_Waitany(count, array_of_requests, index, status, ierror) 48

56

7

8

9

10

11

12

13

14

1516

17

18

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

F binding

```
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
IERROR
```

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

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

The execution of MPI_WAITANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_WAIT(&array_of_requests[i], status), where i is the value returned by index (unless the value of index is MPI_UNDEFINED). MPI_WAITANY with an array containing one active entry is equivalent to MPI_WAIT.

MPI_TESTANY(count, array_of_requests, index, flag, status)

IN	count	list length) (integer)	27
INOUT	array_of_requests	array of requests (array of handles)	28 29
OUT	index	index of operation that completed or	30
		MPI_UNDEFINED if none completed (integer)	31
OUT	flag	true if one of the operations is complete (logical)	32 33
OUT	status	status object (Status)	34

C binding

F08 binding

40 MPI_Testany(count, array_of_requests, index, flag, status, ierror) 41 INTEGER, INTENT(IN) :: count 42TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count) 43 INTEGER, INTENT(OUT) :: index 44LOGICAL, INTENT(OUT) :: flag 45TYPE(MPI_Status) :: status 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47

F binding

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

35

36

37

38 39

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

⁶ Tests for completion of either one or none of the operations associated with active ⁷ handles. In the former case, it returns flag = true, returns in index the index of this request ⁸ in the array, and returns in status the status of that operation. If the request is an active ⁹ persistent request, it is marked as inactive. Any other type of request is deallocated and ¹⁰ the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from ¹¹ one in Fortran.) In the latter case (no operation completed), it returns flag = false, returns ¹² a value of MPI_UNDEFINED in index and status is undefined.

The array may contain null or inactive handles. If the array contains no active handles then the call returns immediately with flag = true, $index = MPI_UNDEFINED$, and an empty status.

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

¹⁶ MPI_TESTANY(count, array_of_requests, index, status) has the same effect as the execution ¹⁷ of MPI_TEST(&array_of_requests[i], flag, status), for i=0, 1,..., count-1, in some arbitrary ¹⁹ order, until one call returns flag = true, or all fail. In the former case, index is set to the ²⁰ last value of i, and in the latter case, it is set to MPI_UNDEFINED. MPI_TESTANY with an ²¹ array containing one active entry is equivalent to MPI_TEST.

```
22
23
```

 31

32

5

MPI_WAITALL(count, array_of_requests, array_of_statuses)

```
    IN count
    INOUT array_of_requests
    OUT array_of_statuses
    array of status objects (array of Status)
```

```
^{29}_{30} C binding
```

```
<sup>33</sup> F08 binding
```

```
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

$_{40}^{33}$ F binding

```
    MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

⁴⁴ Blocks until all communication operations associated with active handles in the list ⁴⁵ complete, and return the status of all these operations (this includes the case where no ⁴⁶ handle in the list is active). Both arrays have the same number of valid entries. The ⁴⁷ i-th entry in array_of_statuses is set to the return status of the i-th operation. Active ⁴⁸ 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.*)

MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)			
IN	count	lists length (integer)	
INOUT	array_of_requests	array of requests (array of handles)	
OUT	flag	(logical)	
OUT	array_of_statuses	array of status objects (array of Status)	

C binding

F08 binding

```
MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*),
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
LOGICAL FLAG

Returns flag = true if all communications associated with active handles in the array have completed (this includes the case where no handle in the list is active). In this case, each $\frac{47}{48}$

Unofficial Draft for Comment Only

1 2 3 4 5 6 7 8 9	operation. deallocated Each statu Other entries are Errors	status entry that corresponds to an active request is set to the status of the corresponding operation. Active persistent requests are marked inactive. Requests of any other type are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. Each status entry that corresponds to a null or inactive handle is set to empty. Otherwise, flag = false is returned, no request is modified and the values of the status entries are undefined. This is a local operation. Errors that occurred during the execution of MPI_TESTALL are handled in the same manner as errors in MPI_WAITALL.			
10 11 12	MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)				
13	IN	incount	length of arry_of_requests (integer)		
14	INOUT	array_of_requests	array of requests (array of handles)		
15	OUT	outcount	number of completed requests (integer)		
16 17 18	OUT	array_of_indices	array of indices of operations that completed (array of integers)		
19 20 21	OUT	array_of_statuses	array of status objects for operations that completed (array of Status)		
23 24 25 26 27 28	<pre>C binding int MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>				
29 30	<pre>MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses, ierror)</pre>				
31		ER, INTENT(IN) :: incou MPI Request), INTENT(INO	nt UT) :: array_of_requests(count)		
32		-	ount, array_of_indices(*)		
33		MPI_Status) :: array_of			
34	INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror		
35 36	F binding	S			
37	MPI_WAITS		QUESTS, OUTCOUNT, ARRAY_OF_INDICES,		
38	ΤΝͲΕΟ	ARRAY_OF_STATUSES, I			
39 40			UESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), SIZE,*), IERROR		
41 42			ations associated with active handles in the list have		
43	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				
44	indices of these operations (index within the array array_of_requests; the array is indexed				
45	from zero in C and from one in Fortran). Returns in the first outcount locations of the				
46 47	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,				
48	requests are marked as inactive. Any other type or request that completed is deallocated, and the associated handle is set to MPL REQUEST NULL				

```
<sup>48</sup> 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)

			15
IN	incount	length of array_of_requests (integer)	16
INOUT	array_of_requests	array of requests (array of handles)	17
OUT	outcount	number of completed requests (integer)	18 19
OUT	array_of_indices	array of indices of operations that completed (array of integers)	20 21
OUT	array_of_statuses	array of status objects for operations that completed	22
		(array of Status)	23 24
			24 25
C bindin	g		25 26
<pre>int MPI_Testsome(int incount, MPI_Request array_of_requests[],</pre>			27
	<pre>int *outcount, int array_of_indices[], MPI_Status array_of_statuses[])</pre>		
F08 bind	ing		30
MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,			31
array_of_statuses, ierror)			32

INTEGER, INTENT(IN) :: incount
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has 44 completed it returns outcount = 0. If there is no active handle in the list it returns outcount 45 = MPI_UNDEFINED. 46

MPI_TESTSOME is a local operation, which returns immediately, whereas MPI_WAITSOME will block until a communication completes, if it was passed a list that

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 7

8 9

10

11

12 13

14

33

34

35

36

37 38

39

40

41

42

43

47

```
1
     contains at least one active handle. Both calls fulfill a fairness requirement: If a request
\mathbf{2}
     for a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or
3
     MPI_TESTSOME, and a matching send has been posted, then the receive will eventually
4
     succeed, unless the send is satisfied by another receive; and similarly for send requests.
5
          Errors that occur during the execution of MPI_TESTSOME are handled as for
6
     MPI_WAITSOME.
7
           Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use
8
9
           of MPI_TESTANY. The former returns information on all completed communications,
           with the latter, a new call is required for each communication that completes.
10
11
           A server with multiple clients can use MPI_WAITSOME so as not to starve any client.
12
           Clients send messages to the server with service requests. The server calls
13
           MPI_WAITSOME with one receive request for each client, and then handles all receives
14
           that completed. If a call to MPI_WAITANY is used instead, then one client could starve
15
           while requests from another client always sneak in first. (End of advice to users.)
16
17
           Advice to implementors. MPI_TESTSOME should complete as many pending com-
18
           munications as possible. (End of advice to implementors.)
19
20
     Example 3.15
                        Client-server code (starvation can occur).
21
22
23
     CALL MPI_COMM_SIZE(comm, size, ierr)
24
     CALL MPI_COMM_RANK(comm, rank, ierr)
25
     IF(rank .GT. 0) THEN
                                       ! client code
26
          DO WHILE(.TRUE.)
27
             CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
28
             CALL MPI_WAIT(request, status, ierr)
29
          END DO
30
     ELSE
                    ! rank=0 -- server code
^{31}
             DO i=1, size-1
32
                 CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
33
                           comm, request_list(i), ierr)
34
             END DO
35
             DO WHILE(.TRUE.)
36
                 CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
37
                 CALL DO_SERVICE(a(1, index)) ! handle one message
38
                 CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag,
39
                            comm, request_list(index), ierr)
40
             END DO
41
     END IF
42
43
                        Same code, using MPI_WAITSOME.
44
     Example 3.16
45
46
47
48
```

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 0) THEN
                             ! client code
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
ELSE
             ! rank=0 -- server code
    DO i=1, size-1
       CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                      comm, request_list(i), ierr)
                                                                                  12
    END DO
    DO WHILE(.TRUE.)
                                                                                  14
       CALL MPI_WAITSOME(size, request_list, numdone,
                         indices, statuses, ierr)
       DO i=1, numdone
          CALL DO_SERVICE(a(1, indices(i)))
                                                                                  19
          CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag,
                                                                                  20
                       comm, request_list(indices(i)), ierr)
                                                                                  21
       END DO
    END DO
                                                                                  22
END IF
                                                                                  23
```

Non-destructive Test of status 3.7.6

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

MPI_REQUEST_GET_STATUS(request, flag, status)

IN	request	request (handle)
OUT	flag	boolean flag, same as from MPI_TEST (logical)
OUT	status	status object if flag is true (Status)

C binding

```
int MPI_Request_get_status(MPI_Request request, int *flag,
             MPI_Status *status)
```

F08 binding

```
MPI_Request_get_status(request, flag, status, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

1 $\mathbf{2}$

3

4

56

7

8

9 10

11

13

15

16

17

18

2425

26

27

28

29

30 3132

39

40

41 42

43

44

45

46

47

1	F binding	у Э		
2	MPI_REQUE	EST_GET_STATUS(REQUEST, F	LAG, STATUS, IERROR)	
3		SER REQUEST, STATUS(MPI_S	TATUS_SIZE), IERROR	
4	LOGIC	CAL FLAG		
5	Sets f	lag = true if the operation is of	complete, and, if so, returns in status the request	
6		-		
7	status. However, unlike test or wait, it does not deallocate or inactivate the request; a subsequent call to test, wait or free should be executed with that request. It sets flag=false			
8	if the operation is not complete.			
9	One is allowed to call MPI_REQUEST_GET_STATUS with a null or inactive request			
10	argument. In such a case the operation returns with flag=true and empty status.			
11 12	0.0	·····		
13	3.8 Pro	be and Cancel		
14				
15	The MPI_	PROBE, MPI_IPROBE, MPI_N	MPROBE, and MPI_IMPROBE operations allow in-	
16	coming messages to be checked for, without actually receiving them. The user can then			
17	decide how to receive them, based on the information returned by the probe (basically, the			
18	information returned by status). In particular, the user may allocate memory for the receive			
19	buffer, according to the length of the probed message.			
20	The MPI_CANCEL operation allows pending communications to be cancelled. This is			
21	-		r a receive ties up user resources (send or receive	
22 23	· · ·	°	o free these resources gracefully. g MPI_CANCEL is deprecated.	
23 24	Cance	ening a send request by caning	g MFI_CANCEL is deprecated.	
25	3.8.1 Pro	ohe		
26	0.0.2			
27				
28 29	MPI_IPRO	BE(source, tag, comm, flag, st	atus)	
30	IN	source	rank of source or MPI_ANY_SOURCE (non-negative	
31			integer)	
32	IN	tag	message tag or MPI_ANY_TAG (integer)	
33 34	IN	comm	communicator (handle)	
35	OUT	flag	(logical)	
36	OUT	status	status object (Status)	
37				
38	C binding	0		
39	int MPI_1	-	g, MPI_Comm comm, int *flag,	
40 41		MPI_Status *status)		
42	F08 bind	ing		
43	MPI_Iprobe(source, tag, comm, flag, status, ierror)			
44	INTEGER, INTENT(IN) :: source, tag			
45	TYPE(MPI_Comm), INTENT(IN) :: comm			
46	LOGICAL, INTENT(OUT) :: flag			
47	TYPE(MPI_Status) :: status			
48	TNLEC	ER, OPTIONAL, INTENT(OUT) :: lerror	

F binding

```
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
LOGICAL FLAG
```

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.

INsourcerank of source or MPI_ANY_SOURCE (non-negative
integer)INtagmessage tag or MPI_ANY_TAG (integer)INcommcommunicator (handle)OUTstatusstatus object (Status)

C binding

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

F08 binding

```
MPI_Probe(source, tag, comm, status, ierror)
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

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

MPI_PROBE(source, tag, comm, status)

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

1	INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR			
2				
3	MPI_PROBE behaves like MPI_IPROBE except that it is a blocking call that returns			
4	only after a matching message has been found.			
5	The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress:			
6	if a call to MPI_PROBE has been issued by a process, and a send that matches the probe			
7	has been initiated by some process, then the call to MPI_PROBE will return, unless the message is received by another concurrent receive operation (that is executed by another			
8	thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and a			
9	matching message has been issued, then the call to MPI_IPROBE will eventually return flag			
10	= true unless the message is received by another concurrent receive operation or matched			
11	by a concurrent matched probe.			
12 13				
14	Example 3.17			
15	Use blocking probe to wait for an incoming message.			
16	CALL MPI_COMM_RANK(comm, rank, ierr)			
17	IF (rank.EQ.0) THEN			
18	CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)			
19	ELSE IF (rank.EQ.1) THEN			
20	CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)			
21	ELSE IF (rank.EQ.2) THEN			
22	DO i=1, 2			
23 24	CALL MPI_PROBE(MPI_ANY_SOURCE, 0,			
25	comm, status, ierr)			
26	IF (status(MPI_SOURCE) .EQ. 0) THEN			
27	100 CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)			
28	ELSE 200 CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)			
29	END IF			
30	END DO			
31	END IF			
32				
$\frac{33}{34}$	Each message is received with the right type.			
35	Example 3.18 A similar program to the previous example, but now it has a problem.			
36	Example 5.15 A similar program to the previous example, but now it has a problem.			
37	CALL MPI_COMM_RANK(comm, rank, ierr)			
38	IF (rank.EQ.0) THEN			
39	CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)			
40	ELSE IF (rank.EQ.1) THEN			
41	CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)			
42	ELSE IF (rank.EQ.2) THEN			
43	DO i=1, 2			
44	CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)			
45 46	IF (status(MPI_SOURCE) .EQ. 0) THEN			
40 47	100 CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,			
48	0, comm, status, ierr)			
	-,,,,			

ELSE

200

CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, 0, comm, status, ierr) END IF END DO END IF

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

In a multithreaded MPI program, MPI_PROBE and Advice to users. 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.

4647

48

1

2

3

4

5

6 7

8

9

10

11 12

13

14

15

16

17

18

19 20

21

22

23

24

25

26

27

28

29

30

31

32

33 34

35 36

37

38

39

40

41

42

43

44

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

```
2
       IN
                                            rank of source or MPI_ANY_SOURCE (non-negative
                 source
3
                                            integer)
4
       IN
                                            message tag or MPI_ANY_TAG (integer)
                 tag
5
6
       IN
                                            communicator (handle)
                 comm
7
       OUT
                 flag
                                             (logical)
8
       OUT
                 message
                                            returned message (handle)
9
10
       OUT
                 status
                                            status object (Status)
11
12
     C binding
13
     int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,
14
                    MPI_Message *message, MPI_Status *status)
15
     F08 binding
16
     MPI_Improbe(source, tag, comm, flag, message, status, ierror)
17
          INTEGER, INTENT(IN) :: source, tag
18
          TYPE(MPI_Comm), INTENT(IN) :: comm
19
          LOGICAL, INTENT(OUT) :: flag
20
          TYPE(MPI_Message), INTENT(OUT) :: message
21
          TYPE(MPI_Status) :: status
22
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
23
^{24}
     F binding
25
     MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
26
          INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
27
          LOGICAL FLAG
28
         MPI_IMPROBE(source, tag, comm, flag, message, status) returns flag = true if there is
29
     a message that can be received and that matches the pattern specified by the arguments
30
     source, tag, and comm. The call matches the same message that would have been received
^{31}
     by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point in the
32
     program and returns in status the same value that would have been returned by MPI_RECV.
33
     In addition, it returns in message a handle to the matched message. Otherwise, the call
34
     returns flag = false, and leaves status and message undefined.
35
          A matched receive (MPI_MRECV or MPI_IMRECV) executed with the message han-
36
     dle will receive the message that was matched by the probe. Unlike MPI_IPROBE, no
37
     other probe or receive operation may match the message returned by MPI_IMPROBE.
38
     Each message returned by MPI_IMPROBE must be received with either MPI_MRECV or
39
     MPI_IMRECV.
40
         The source argument of MPI_IMPROBE can be MPI_ANY_SOURCE, and the tag argu-
41
     ment can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source
42
     and/or with an arbitrary tag. However, a specific communication context must be provided
43
     with the comm argument.
44
          A synchronous send operation that is matched with MPI_IMPROBE or MPI_MPROBE
45
     will complete successfully only if both a matching receive is posted with MPI_MRECV or
46
     MPI_IMRECV, and the receive operation has started to receive the message sent by the
47
     synchronous send.
48
```

There is a special predefined message: MPI_MESSAGE_NO_PROC, which is a message which has MPI_PROC_NULL as its source process. The predefined constant MPI_MESSAGE_NULL is the value used for invalid message handles. A matching probe with MPI_PROC_NULL as source returns flag = true, message = MPI_MESSAGE_NO_PROC, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0; see Section 3.11. It is not necessary to call MPI_MRECV or MPI_IMRECV with MPI_MESSAGE_NO_PROC, but it is not erroneous to do so. <i>Rationale.</i> MPI_MESSAGE_NO_PROC was chosen instead of MPI_MESSAGE_PROC_NULL to avoid possible confusion as another null handle constant. (<i>End of rationale.</i>)			
MPI_MPF	OBE(source, tag, comm, messa	age, status)	14 15
IN	source	rank of source or MPI_ANY_SOURCE (non-negative integer)	16 17
IN	tag	message tag or MPI_ANY_TAG (integer)	18
IN	comm	communicator (handle)	19 20
OUT	message	returned message (handle)	21
OUT	status	status object (Status)	22
	Mprobe(int source, int ta MPI_Status *status)	g, MPI_Comm comm, MPI_Message *message,	23 24 25 26 27
F08 bind	6	and status immore)	28
-	be(source, tag, comm, mes GER, INTENT(IN) :: sourc	-	29 30
	(MPI_Comm), INTENT(IN) ::	0	31
	(MPI_Message), INTENT(OUT) :: message	32
	(MPI_Status) :: status GER, OPTIONAL, INTENT(OUT) :: ierror	33 34
			35
F bindin	g BE(SOURCE, TAG, COMM, MES	SAGE STATUS TERROR)	36
		SSAGE, STATUS(MPI_STATUS_SIZE), IERROR	37 38
only after The i	a matching message has been	$BE \text{ and } MPI_IMPROBE \text{ needs to guarantee progress}$	39 40 41 42 43
3.8.3 M	atched Receives		44
		MRECV receive messages that have been previously	45
	by a matching probe (Section)		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 11 C binding int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype, 12MPI_Message *message, MPI_Status *status) 13 14F08 binding 15MPI_Mrecv(buf, count, datatype, message, status, ierror) 16TYPE(*), DIMENSION(..) :: buf 17 INTEGER, INTENT(IN) :: count 18 TYPE(MPI_Datatype), INTENT(IN) :: datatype 19 TYPE(MPI_Message), INTENT(INOUT) :: message 20TYPE(MPI_Status) :: status 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 F binding 23 24 MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR) 25<type> BUF(*) 26INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR 27This call receives a message matched by a matching probe operation (Section 3.8.2). 28The receive buffer consists of the storage containing **count** consecutive elements of the 29 type specified by datatype, starting at address buf. The length of the received message must 30 be less than or equal to the length of the receive buffer. An overflow error occurs if all 31 incoming data does not fit, without truncation, into the receive buffer. 32 If the message is shorter than the receive buffer, then only those locations corresponding 33 to the (shorter) message are modified. 34 On return from this function, the message handle is set to MPI_MESSAGE_NULL. All 35 errors that occur during the execution of this operation are handled according to the error 36 handler set for the communicator used in the matching probe call that produced the message 37 handle. 38 If MPI_MRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the 39

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

- $45 \\ 46$
- 47
- 48

MPI_IMRECV(buf, count, datatype, message, request)			1	
OUT	buf	initial address of receive buffer (choice)	2	
IN	count	number of elements in receive buffer (non-negative in- teger)	3 4 5	
IN	datatype	datatype of each receive buffer element (handle)	6	
INOUT	message	message (handle)	7 8	
OUT	request	communication request (handle)	9	
			10	
C binding				
int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,			12	
MPI_Message *message, MPI_Request *request) 13			13	
F08 binding				
MPI_Imrecv(buf, count, datatype, message, request, ierror)				
	(*), DIMENSION(), ASYNCI	o i	16	
	ER, INTENT(IN) :: count		17	
	[MPI_Datatype), INTENT(IN]) :: datatype	18	
	(MPI_Message), INTENT(INO		19	
	(MPI_Request), INTENT(OUT)	0	20	
	ER, OPTIONAL, INTENT(OUT)	-	21	
infident, of flowel, infent(oof) feffor				
F binding 23				

MPI_IMRECV(BUF,	COUNT,	DATATYPE	, MESSAC	GE, REQUE	ST, II	ERROR)
<type> BUF(*</type>)					
INTEGER COUN	T, DAT	ATYPE, ME	SSAGE, F	REQUEST,	IERROF	1

MPI_IMRECV is the nonblocking variant of MPI_MRECV and starts a nonblocking receive of a matched message. Completion semantics are similar to MPI_IRECV as described in Section 3.7.2. On return from this function, the message handle is set to MPI_MESSAGE_NULL.

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

Advice to implementors. If reception of a matched message is started with 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 probe (MPI_PROBE, MPI_IPROBE, MPI_MPROBE, or MPI_IMPROBE), received by a subsequent receive operation or cancelled by the sender. See Section 3.8.4 for details about MPI_CANCEL. The cancellation of operations initiated with MPI_IMRECV may fail. (End of advice to implementors.)

> 4647

28

29

30

31

32

33

34

35

36 37

38

39

40

41

42

43

```
1
     3.8.4
            Cancel
2
3
4
     MPI_CANCEL(request)
5
       IN
                 request
                                            communication request (handle)
6
7
     C binding
8
     int MPI_Cancel(MPI_Request *request)
9
10
     F08 binding
11
     MPI_Cancel(request, ierror)
12
         TYPE(MPI_Request), INTENT(IN) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
14
     F binding
15
     MPI_CANCEL(REQUEST, IERROR)
16
         INTEGER REQUEST, IERROR
17
18
         A call to MPI_CANCEL marks for cancellation a pending, nonblocking communica-
19
```

tion operation (send or receive). Cancelling a send request by calling MPI_CANCEL is 20deprecated. The cancel call is local. It returns immediately, possibly before the communi-21cation is actually cancelled. It is still necessary to call MPI_REQUEST_FREE, MPI_WAIT or 22MPI_TEST (or any of the derived operations) with the cancelled request as argument after 23the call to MPI_CANCEL. If a communication is marked for cancellation, then a MPI_WAIT 24call for that communication is guaranteed to return, irrespective of the activities of other 25processes (i.e., MPI_WAIT behaves as a local function); similarly if MPI_TEST is repeatedly 26called in a busy wait loop for a cancelled communication, then MPI_TEST will eventually 27be successful. 28

MPI_CANCEL can be used to cancel a communication that uses a persistent request (see 29 Section 3.9), in the same way it is used for nonpersistent requests. Cancelling a persistent 30 send request by calling MPI_CANCEL is deprecated. A successful cancellation cancels the 31 active communication, but not the request itself. After the call to MPI_CANCEL and the 32 subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can be 33 activated for a new communication. 34

The successful cancellation of a buffered send frees the buffer space occupied by the 35 pending message. Cancelling a buffered send request by calling MPI_CANCEL is deprecated. 36

Either the cancellation succeeds, or the communication succeeds, but not both. If a 37 send is marked for cancellation, which is deprecated, then it must be the case that either 38 the send completes normally, in which case the message sent was received at the destination 39 process, or that the send is successfully cancelled, in which case no part of the message 40 was received at the destination. Then, any matching receive has to be satisfied by another send. If a receive is marked for cancellation, then it must be the case that either the receive 42completes normally, or that the receive is successfully cancelled, in which case no part of the 43 receive buffer is altered. Then, any matching send has to be satisfied by another receive.

44 If the operation has been cancelled, then information to that effect will be returned in 45the status argument of the operation that completes the communication. 46

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-

Unofficial Draft for Comment Only

41

47

			4
MPI_TEST_CANCELLED(status, flag)			
IVIPI_	TEST_CANCELLED(sta	itus, fiag)	6
IN	status	status object (Status)	7
OU	T flag	(logical)	8
00	1000	(1081011)	9
a 1.	1.		10
	nding		11
int	PI_Test_cancelled(c	const MPI_Status *status, int *flag)	12
F08 binding			
MPI_Test_cancelled(status, flag, ierror)			14
TYPE(MPI_Status), INTENT(IN) :: status			15
LOGICAL, INTENT(OUT) :: flag			16
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
			18
F binding			19
	FEST_CANCELLED(STATU		20
	. –	TATUS_SIZE), IERROR	21
	LOGICAL FLAG		22
]	Returns flag = true if the	communication associated with the status object was cancelled	23

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.)

If a send operation uses an "eager" protocol (data is Advice to implementors. 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 (with the exception of the buffer contents) is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a **persistent** communication request once and, then, repeatedly

Unofficial Draft for Comment Only

 24

1using the request to initiate and complete operations. In the case of point-to-point commu- $\mathbf{2}$ nication, the persistent request thus created can be thought of as a communication port or 3 a "half-channel." It does not provide the full functionality of a conventional channel, since 4 there is no binding of the send port to the receive port. This construct allows reduction 5of the overhead for communication between the process and communication controller, but 6 not of the overhead for communication between one communication controller and another. 7 It is not necessary that messages sent with a persistent point-to-point request be received 8 by a receive operation using a persistent point-to-point request, or vice versa.

⁹ There are also collective communication persistent operations defined in Section 5.13 and Section 7.8. The remainder of this section covers the point-to-point persistent initialization operations and the start routines, which are used for both point-to-point and collective persistent communication.

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

15 16 17

27

29

30

 31

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

18	IN	buf	initial address of send buffer (choice)
19 20	IN	count	number of elements sent (non-negative integer)
21	IN	datatype	type of each element (handle)
22	IN	dest	rank of destination (non-negative integer)
23 24	IN	tag	message tag (integer)
25	IN	comm	communicator (handle)
26	OUT	request	communication request (handle)

```
<sup>28</sup> C binding
```

```
32 F08 binding
```

```
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
33
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
34
         INTEGER, INTENT(IN) :: count, dest, tag
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         TYPE(MPI_Request), INTENT(OUT) :: request
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     F binding
41
     MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
42
         <type> BUF(*)
43
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
44
         Creates a persistent communication request for a standard mode send operation, and
45
     binds to it all the arguments of a send operation.
46
```

MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request) ¹				
IN	buf	initial address of send buffer (choice)	2	
IN	count	number of elements sent (non-negative integer)	3 4	
IN	datatype	type of each element (handle)	5	
IN	dest	rank of destination (non-negative integer)	6	
IN	tag	message tag (integer)	7	
IN	comm	communicator (handle)	9	
OUT	request	communication request (handle)	10	
	·	· ()	11 12	
C bindin	•		13	
int MPI_		(DI Commonweak MDI Demonstration and MDI Demonstration)	14	
	-	<pre>MPI_Comm comm, MPI_Request *request)</pre>	15	
F08 bind	0	me deat ter comm request ierner)	16 17	
	(*), DIMENSION(), INTEN	pe, dest, tag, comm, request, ierror) T(IN), ASYNCHRONOUS :: buf	18	
	GER, INTENT(IN) :: count		19	
	(MPI_Datatype), INTENT(IN		20	
	(MPI_Comm), INTENT(IN) ::		21 22	
	(MPI_Request), INTENT(OUT	-	23	
	GER, OPTIONAL, INTENT(OUT) :: lerror	24	
F bindin	0		25	
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 24				
• 1	<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type>			
Creates a persistent communication request for a buffered mode send.				
Creat	es a persistent communicatio.	n request for a bunered mode send.	30	
			31	
MPI_SSEI	ND_INIT(buf, count, datatype,	dest, tag, comm, request)	32	
IN	buf	initial address of send buffer (choice)	33 34	
IN	count	number of elements sent (non-negative integer)	35	
IN	datatype	type of each element (handle)	36	
IN	dest	rank of destination (non-negative integer)	37	
IN	tag	message tag (integer)	38 39	
IN	comm	communicator (handle)	40	
OUT	request	communication request (handle)	41	
			42	
C bindin	0		43 44	
<pre>int MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype,</pre>				
	<pre>int dest, int tag, MPI_Comm comm, MPI_Request *request) 46</pre>			
	F08 binding 47			
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror) 48				

1 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 2 INTEGER, INTENT(IN) :: count, dest, tag 3 TYPE(MPI_Datatype), INTENT(IN) :: datatype 4 TYPE(MPI_Comm), INTENT(IN) :: comm 5TYPE(MPI_Request), INTENT(OUT) :: request 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 7 F binding 8 MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 9 <type> BUF(*) 10 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 11 12Creates a persistent communication object for a synchronous mode send operation. 13 14MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request) 1516IN buf initial address of send buffer (choice) 17IN count number of elements sent (non-negative integer) 18 IN datatype type of each element (handle) 19 20IN dest rank of destination (non-negative integer) 21IN message tag (integer) tag 22 IN comm communicator (handle) 23 24 OUT communication request (handle) request 2526C binding 27int MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype, 28int dest, int tag, MPI_Comm comm, MPI_Request *request) 29 F08 binding 30 MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror) 31 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 32 INTEGER, INTENT(IN) :: count, dest, tag 33 34 TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm 35 TYPE(MPI_Request), INTENT(OUT) :: request 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 38 F binding 39 MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 40 <type> BUF(*) 41 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 42Creates a persistent communication object for a ready mode send operation. 43 44 4546 47 48

MPI_REC	√_INIT(buf, count, datatype, s	ource, tag, comm, request)	1
OUT	buf	initial address of receive buffer (choice)	2 3
IN	count	number of elements received (non-negative integer)	4
IN	datatype	type of each element (handle)	5
IN	source	rank of source or MPI_ANY_SOURCE (non-negative	6
		integer)	7 8
IN	tag	message tag or MPI_ANY_TAG (integer)	9
IN	comm	communicator (handle)	10
OUT	request	communication request (handle)	11
		- `` `	12 13
C bindin	0		14
int MPI_H		count, MPI_Datatype datatype, int source,	15
	C	mm, MPI_Request *request)	16
F08 bind	0		17 18
	_init(buf, count, datatyp (*), DIMENSION(), ASYNC	e, source, tag, comm, request, ierror) HRONOUS ·· buf	19
	GER, INTENT(IN) :: count		20
TYPE	(MPI_Datatype), INTENT(IN) :: datatype	21
	(MPI_Comm), INTENT(IN) ::		22 23
IPE(MPI_Request), INTENI(UUI) :: request			23 24
			25
F binding	•	E, SOURCE, TAG, COMM, REQUEST, IERROR)	26
	= BUF(*)	E, SUURCE, ING, CUMM, REQUESI, IERRUR/	27 28
• 1		CE, TAG, COMM, REQUEST, IERROR	28 29
Creat	es a persistent communication	a request for a receive operation. The argument buf	30
	-	permission to write on the receive buffer by passing	31
0	ent to MPI_RECV_INIT.		32 33
-	-	t is inactive after it was created — no active com-	34
	n is attached to the request. nmunication (send or receive)) that uses a persistent request is initiated by the	35
	IPI_START.		36
			37 38
MPI STAF	RT(request)		30 39
INOUT	request	communication request (handle)	40
	request	communeation request (namine)	41
C bindin	g		42
int MPI_S	Start(MPI_Request *reques	t)	43 44
F08 bind	ing		45
MPI_Start	t(request, ierror)		46
	(MPI_Request), INTENT(INO	-	47
INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	48

1	F binding	r.		
2	MPI_START(REQUEST, IERROR)			
3	INTEGER REQUEST, IERROR			
4				
5		с , . ,	adle returned by one of the previous five calls. The	
6		-	The request becomes active once the call is made.	
7		-	eady mode, then a matching receive should be posted	
8		before the call is made. The communication buffer should not be modified after the call,		
9	and until the operation completes.			
10	The call is local, with similar semantics to the nonblocking communication operations described in Section 3.7. That is, a call to MPI_START with a request created by			
11			ation in the same manner as a call to MPI_ISEND; a	
12			reated by MPI_BSEND_INIT starts a communication	
13		e manner as a call to MPI		
14	III the sam	e manner as a can to wir i	IDSEND, and so on.	
15				
16	MPI_STAF	RTALL(count, array_of_requ	ests)	
17 18	IN	count	list length (integer)	
19	INOUT	array_of_requests	array of requests (array of handles)	
20	moor	anay_or_requests	array of requests (array of handles)	
21	C binding	n,		
22			_Request array_of_requests[])	
23			_nequebt array_or_requebts[])	
24	F08 bind	•		
25		all(count, array_of_re	-	
26		ER, INTENT(IN) :: cou		
27		-	NOUT) :: array_of_requests(count)	
28	INTEG	ER, OPTIONAL, INTENT(O	UI) :: lerror	
29	F binding			
30		CALL(COUNT, ARRAY_OF_RE		
31 32	INTEG	ER COUNT, ARRAY_OF_REQ	UESTS(*), IERROR	
33	Start	all communications assoc	iated with requests in array_of_requests. A call to	
34			uests) has the same effect as calls to	
35			executed for $i=0$,, count-1, in some arbitrary order.	
36			call to MPI_START or MPI_STARTALL is completed	
37	by a call	to MPI_WAIT, MPI_TEST	, or one of the derived functions described in Sec-	
38	tion $3.7.5$.	The request becomes inac	tive after successful completion of such call. The re-	
39	quest is no	t deallocated and it can be	activated anew by an MPI_START or MPI_STARTALL	
40	call.			
41	-	-	ed by a call to $MPI_REQUEST_FREE$ (Section 3.7.3).	
42			E can occur at any point in the program after the per-	
43	-		the request will be deallocated only after it becomes	
44		_	ild not be freed. Otherwise, it will not be possible to	
45		_	Collective operation requests (defined in Section 5.12	
46		_	ctive operations, and Section 5.13 and Section 7.8 for	
47	persistent	conective operations) must	not be freed while active. It is preferable, in general,	
48				

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 17.1.10–17.1.20. (End of advice to users.)

3.10 Send-Receive

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

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation. $\mathbf{2}$

 24

12	MPI_SEND	DRECV(sendbuf, sendcount, se source, recvtag, comm, s	ndtype, dest, sendtag, recvbuf, recvcount, recvtype, tatus)
3	IN	sendbuf	initial address of send buffer (choice)
4 5 6	IN	sendcount	number of elements in send buffer (non-negative integer)
7	IN	sendtype	type of elements in send buffer (handle)
8	IN	dest	rank of destination (non-negative integer)
9	IN	sendtag	send tag (integer)
10 11	OUT	recvbuf	initial address of receive buffer (choice)
12 13	IN	recvcount	number of elements in receive buffer (non-negative in- teger)
14	IN	recvtype	type of elements receive buffer element (handle)
15 16 17	IN	source	rank of source or MPI_ANY_SOURCE (non-negative integer)
18	IN	recvtag	receive tag or MPI_ANY_TAG (integer)
19	IN	comm	communicator (handle)
20 21	OUT	status	status object (Status)
24 25 26 27	int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)		
28	F08 bind	0	
29 30	MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror)		
31	TYPE((*), DIMENSION(), INTEN	-
32			ount, dest, sendtag, recvcount, source,
33	recvt	0	
$\frac{34}{35}$		<pre>[MPI_Datatype), INTENT(IN] [*), DIMENSION() :: reference</pre>	VI VI
36		(MPI_Comm), INTENT(IN) ::	
37		MPI_Status) :: status	
38	INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror
39 40	F binding	r	
41	MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,		
42	RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)		
43	• 1	> SENDBUF(*), RECVBUF(*) FR SENDCOUNT SENDTYPE	DEST, SENDTAG, RECVCOUNT, RECVTYPE,
44			(MPI_STATUS_SIZE), IERROR
$45 \\ 46$			ve operation. Both send and receive use the same
47		0	ags. The send buffer and receive buffers must be
48	disjoint, and may have different lengths and datatypes.		

 24

 $45 \\ 46$

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

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, sta-

	tus)		7
INOUT	buf	initial address of send and receive buffer (choice)	8
IN	count	number of elements in send and receive buffer (non-negative integer)	9 10
IN	datatype	type of elements in send and receive buffer (handle)	11 12
IN	dest	rank of destination (non-negative integer)	13
IN	sendtag	send message tag (integer)	14
IN	source	rank of source or MPI_ANY_SOURCE (non-negative integer)	15 16 17
IN	recvtag	receive message tag or MPI_ANY_TAG (integer)	18
IN	comm	communicator (handle)	19 20
OUT	status	status object (Status)	20 21

C binding

F08 binding

F binding

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

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

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

3.11 Null Processes

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

The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI_PROC_NULL has no effect. A send to MPI_PROC_NULL succeeds and returns as soon as possible. A receive from MPI_PROC_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with $source = MPI_PROC_NULL$ is executed then the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0. A probe or matching probe with source $= MPI_PROC_NULL$ succeeds and returns as soon as possible, and the status object returns source = MPI_PROC_NULL , tag = MPI_ANY_TAG and count = 0. A matching probe (cf. Section 3.8.2) with MPI_PROC_NULL as source returns $flag = true, message = MPI_MESSAGE_NO_PROC, and the status object returns source =$ MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0.

Chapter 4

Datatypes

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

4.1**Derived** Datatypes

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

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

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

+ +bot • •

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

7 8 9

15

16

17

18 19

2021

22

23

 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

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

6 7

1

 $\mathbf{2}$

3

4

5

8 9

10

11 12

13

14

15

 $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

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

Let

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

¹⁶ Most datatype constructors have replication count or block length arguments. Allowed ¹⁷ values are non-negative integers. If the value is zero, no elements are generated in the type ¹⁸ map and there is no effect on datatype bounds or extent.

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

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

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

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

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

then

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)

 $41 \\ 42$

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

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

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. In Fortran, structures can be expressed with several language features, e.g., common blocks, SEQUENCE derived types, or BIND(C) derived types. The compiler may use different alignments, and therefore, it is recommended to use MPI_TYPE_CREATE_RESIZED for arrays of structures if an alignment may cause an alignment-gap at the end of a structure as described in Section 4.1.6 and in Section 17.1.15. (End of rationale.)

4.1.1 Type Constructors with Explicit Addresses

INTEGER, OPTIONAL, INTENT(OUT) ::

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 (integer)	
IN	oldtype	old datatype (handle)	
OUT	newtype	new datatype (handle)	
C binding int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)			
F08 binding			
MPI_Type_contiguous(count, oldtype, newtype, ierror)			
INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: oldtype			
TYPE(MPI_Datatype), INTENT(OUT) :: newtype			

ierror

 $\overline{7}$

 24

1	D 1 · 1 ·			
1 2 3	F binding MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)			
4	INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR			
5 6	newtype is the datatype obtained by concatenating count copies of oldtype. Concatenation is defined using <i>extent</i> as the size of the concatenated copies.			
7 8 9	Example 4.2 Let oldtype have type map $\{(double, 0), (char, 8)\}$, with extent 16, and let $count = 3$. The type map of the datatype returned by newtype is			
10 11	{(do	uble, 0), (char, 8), (double, 16)	$), (char, 24), (double, 32), (char, 40) \};$	
12	i.e., altern	ating double and char elemen	nts, with displacements $0, 8, 16, 24, 32, 40$.	
13 14	In ger	neral, assume that the type ma	ap of oldtype is	
15	$\{(typ)\}$	$(type_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$-1)\},$	
16 17	with exten	t ex . Then newtype has a type	e map with $count \cdot n$ entries defined by:	
18 19	$\{(type_0$	$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots, (type_{n-1}, disp_{n-1}), \dots, (type_{n-1}, disp_{n-1}), \dots, (type_{n-1}, disp_{n-1} + ex), \dots, (type_{n$		
20 21	$\ldots, (ty$	$\ldots, (type_0, disp_0 + ex \cdot (count - 1)), \ldots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}.$		
23 24 25 26 27	Vector The function MPI_TYPE_VECTOR is a more general constructor that allows repli- cation of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.			
28	MPI_TYPI	E_VECTOR(count, blocklength	, stride, oldtype, newtype)	
29 30	IN	count	number of blocks (integer)	
31 32	IN	blocklength	number of elements in each block (non-negative integer)	
33 34	IN	stride	number of elements between start of each block (integer)	
35	IN	oldtype	old datatype (handle)	
$\frac{36}{37}$	OUT	newtype	new datatype (handle)	
38 39 40 41	C binding int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)			
42	F08 bind	F08 binding		
43	MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)			
44	INTEGER, INTENT(IN) :: count, blocklength, stride			
45	TYPE(MPI_Datatype), INTENT(IN) :: oldtype			
46 47	TYPE(MPI_Datatype), INTENT(OUT) :: newtype			
48	INTEC	ER, OPTIONAL, INTENT(OUT)) :: ierror	

F binding MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR	1 2 3 4
Example 4.3 Assume, again, that oldtype has type map {(double, 0), (char, 8)}, with extent 16. A call to MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) will create the datatype with type map,	5 6 7 8
$\{(\texttt{double},0),(\texttt{char},8),(\texttt{double},16),(\texttt{char},24),(\texttt{double},32),(\texttt{char},40),$	9 10
$(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104) \}.$	11
That is, two blocks with three copies each of the old type, with a stride of 4 elements $(4 \cdot 16 \text{ bytes})$ between the the start of each block.	12 13 14
Example 4.4 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,	15 16 17
$\{(\texttt{double}, 0), (\texttt{char}, 8), (\texttt{double}, -32), (\texttt{char}, -24), (\texttt{double}, -64), (\texttt{char}, -56)\}.$	18 19
In general, assume that oldtype has type map,	20
$\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\$	21 22
with extent ex . Let bl be the blocklength. The newly created datatype has a type map with count \cdot bl \cdot n entries:	23 24 25
$\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1}),$	26
$(type_0, disp_0 + ex), \ldots, (type_{n-1}, disp_{n-1} + ex), \ldots,$	27 28
$(type_0, disp_0 + (bl - 1) \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),$	29 30
$(type_0, disp_0 + stride \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + stride \cdot ex), \ldots,$	31 32
$(type_0, disp_0 + (stride + bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex), \dots,$	33 34
$(type_0, disp_0 + stride \cdot (count - 1) \cdot ex), \ldots,$	35 36
$(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) \cdot ex), \dots,$	37
$(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots,$	38 39
	40
$(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}.$	41 42
A call to $MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)$ is equivalent to a call to	
MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1,	44
count, n, oldtype, newtype), n arbitrary.	45 46

```
1
      Hvector The function MPI_TYPE_CREATE_HVECTOR is identical to
\mathbf{2}
      MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The
3
      use for both types of vector constructors is illustrated in Section 4.1.14. (H stands for
4
      "heterogeneous").
5
6
      MPI_TYPE_CREATE_HVECTOR(count, blocklength, stride, oldtype, newtype)
\overline{7}
8
        IN
                                                  number of blocks (integer)
                   count
9
        IN
                   blocklength
                                                  number of elements in each block (non-negative inte-
10
                                                  ger)
11
        IN
                   stride
                                                  number of bytes between start of each block (integer)
12
13
        IN
                   oldtype
                                                  old datatype (handle)
14
        OUT
                                                  new datatype (handle)
                   newtype
15
16
      C binding
17
      int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
18
                       MPI_Datatype oldtype, MPI_Datatype *newtype)
19
20
      F08 binding
21
      MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
22
                       ierror)
23
           INTEGER, INTENT(IN) :: count, blocklength
^{24}
           INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
25
           TYPE(MPI_Datatype), INTENT(IN) :: oldtype
26
           TYPE(MPI_Datatype), INTENT(OUT) :: newtype
27
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
      F binding
29
      MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
30
                       IERROR)
^{31}
           INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
32
           INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
33
34
           Assume that oldtype has type map,
35
            \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\
36
37
      with extent ex. Let bl be the blocklength. The newly created datatype has a type map with
38
      \operatorname{count} \cdot \operatorname{bl} \cdot n entries:
39
40
            \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1}), 
41
42
            (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots,
43
            (type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),
44
45
            (type_0, disp_0 + \mathsf{stride}), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{stride}), \ldots,
46
47
            (type_0, disp_0 + stride + (bl - 1) \cdot ex), \ldots,
48
```

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

Indexed The function MPI_TYPE_INDEXED allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, new-

	type)		10
	type)		16
IN	count	number of blocks – also number of entries in	17
		<code>array_of_displacements</code> and <code>array_of_blocklengths</code> (in-	18
		teger)	19
IN	array_of_blocklengths	number of elements per block (array of non-negative	20
	,	integers)	21
	amou of disale concerts		22
IN	array_of_displacements	displacement for each block, in multiples of oldtype	23
		(array of integers)	24
IN	oldtype	old datatype (handle)	25
OUT	newtype	new datatype (handle)	26

C binding

F08 binding

F binding

Example 4.5

1	Lot of	dtype have type man ((doub	le, 0), (char, 8), with extent 16. Let $B = (3, 1)$
2			$PE_{INDEXED(2, B, D, oldtype, newtype) returns a$
3		with type map, $(4, 0)$. A can to 100 mm^{-1}	L_{1} L_{2} L_{2} L_{2} L_{3} L_{3
4			
5	$\{(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104), \\$		
6	$(\texttt{double}, 0), (\texttt{char}, 8)\}.$		
7	That is, th	aree copies of the old type sta	rting at displacement 64, and one copy starting at
8 9	displaceme	ent 0.	
9 10	In gen	neral, assume that oldtype has	type map,
11	$\{(typ)\}$	$(be_0, disp_0), \dots, (type_{n-1}, disp_n)$	$_{n-1})\},$
12 13			locklengths argument and D be the ewly created datatype has $n \cdot \sum_{i=0}^{\text{count}-1} B[i]$ entries:
14 15			$pe_{n-1}, disp_{n-1} + D[0] \cdot ex), \dots,$
16 17	(type	$B_0, disp_0 + (D[0] + B[0] - 1) \cdot \mathbf{e}_0$	$ex),\ldots,$
18	(type	$e_{n-1}, disp_{n-1} + (D[0] + B[0] - $	$(1) \cdot ex), \ldots,$
19 20	(type	$e_0, disp_0 + D[count-1] \cdot ex), \dots$	$, (type_{n-1}, disp_{n-1} + D[count-1] \cdot ex), \ldots,$
20	(type	$B_0, disp_0 + (D[count-1] + B[count-1])$	$[int-1] - 1) \cdot ex), \ldots,$
22 23	(type	$e_{n-1}, disp_{n-1} + (D[count-1] +$	$B[count-1] - 1) \cdot ex)\}.$
24 25	A call to MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent to a call to MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) where		
26	$D[j] = j \cdot stride, \ j = 0, \dots, count - 1,$		
27	and		
28 29	B[j] =	= blocklength, $j = 0, \ldots,$ coun	t – 1.
30	Hindexed	The function MPL TVPE C	REATE_HINDEXED is identical to
31 32			k displacements in array_of_displacements are spec-
33		tes, rather than in multiples of	
34		r i i i i i i i i i i i i i i i i i i i	
35			
36 37	MPI_TYPE	type, newtype)	, array_of_blocklengths, array_of_displacements, old-
38	IN	count	number of blocks – also number of entries in
39			<code>array_of_displacements</code> and <code>array_of_blocklengths</code> (in-
40			teger)
41	IN	array_of_blocklengths	number of elements in each block (array of non-negative
42			integers)
43 44	IN	array_of_displacements	byte displacement of each block (array of integers)
44 45	IN	oldtype	old datatype (handle)
46	OUT	newtype	new datatype (handle)
47			
48	C binding	r 5	

1 int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[], 2 const MPI_Aint array_of_displacements[], MPI_Datatype oldtype, 3 MPI_Datatype *newtype) 4 F08 binding 5MPI_Type_create_hindexed(count, array_of_blocklengths, 6 array_of_displacements, oldtype, newtype, ierror) 7 INTEGER, INTENT(IN) :: count, array_of_blocklengths(count) 8 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: 9 array_of_displacements(count) 10 TYPE(MPI_Datatype), INTENT(IN) :: oldtype 11 TYPE(MPI_Datatype), INTENT(OUT) :: newtype 12INTEGER, OPTIONAL, INTENT(OUT) :: ierror 13 14F binding 15MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, 16ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) 17INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR 18 INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*) 19 Assume that oldtype has type map, 2021 $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$ 22 with extent ex. Let B be the array_of_blocklengths argument and D be the 23array_of_displacements argument. The newly created datatype has a type map with $n \cdot$ 24 $\sum_{i=0}^{\text{count}-1} B[i]$ entries: 2526 $\{(type_0, disp_0 + D[0]), \dots, (type_{n-1}, disp_{n-1} + D[0]), \dots, \}$ 2728 $(type_0, disp_0 + D[0] + (B[0] - 1) \cdot ex), \dots,$ 29 30 $(type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1) \cdot ex), \dots,$ 3132 $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}]), \dots, (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}]), \dots,$ 33 34 $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex), \dots,$ 35 $(type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}] + (\mathsf{B}[\mathsf{count-1}] - 1) \cdot ex)\}.$ 36 37 38

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

48

39

40

41

```
1
     MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements, oldtype,
\mathbf{2}
                     newtype)
3
       IN
                                             length of array of displacements (integer)
                 count
4
       IN
                 blocklength
                                             size of block (non-negative integer)
5
6
       IN
                 array_of_displacements
                                             array of displacements (array of integers)
7
       IN
                 oldtype
                                             old datatype (handle)
8
       OUT
                 newtype
                                             new datatype (handle)
9
10
     C binding
11
     int MPI_Type_create_indexed_block(int count, int blocklength,
12
                     const int array_of_displacements[], MPI_Datatype oldtype,
13
                    MPI_Datatype *newtype)
14
15
     F08 binding
16
     MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
17
                    oldtype, newtype, ierror)
18
          INTEGER, INTENT(IN) :: count, blocklength,
19
          array_of_displacements(count)
20
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
21
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
22
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
23
     F binding
^{24}
     MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
25
26
                    OLDTYPE, NEWTYPE, IERROR)
          INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
27
          NEWTYPE, IERROR
28
29
30
     Hindexed_block The function MPI_TYPE_CREATE_HINDEXED_BLOCK is identical to
^{31}
     MPI_TYPE_CREATE_INDEXED_BLOCK, except that block displacements in
32
     array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.
33
34
35
     MPI_TYPE_CREATE_HINDEXED_BLOCK(count, blocklength, array_of_displacements, old-
36
                    type, newtype)
37
       IN
                 count
                                             length of array of displacements (integer)
38
       IN
                 blocklength
                                             size of block (non-negative integer)
39
40
       IN
                 array_of_displacements
                                             byte displacement of each block (array of integers)
41
       IN
                 oldtype
                                             old datatype (handle)
42
       OUT
                 newtype
                                             new datatype (handle)
43
44
45
     C binding
46
     int MPI_Type_create_hindexed_block(int count, int blocklength,
47
                     const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
48
                     MPI_Datatype *newtype)
```

<pre>F08 binding MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,</pre>				
TYPE(TYPE(array_of_displacements(count) TYPE(MPI_Datatype), INTENT(IN) :: oldtype TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
<pre>F binding MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,</pre>				
generalizes		is the most general type constructor. It further XED in that it allows each block to consist of repli-	17 18 19 20 21	
MPI_TYPE	_CREATE_STRUCT(count, ar newtype)	ray_of_blocklengths, array_of_displacements, array_of	_types, 23	
IN	count	<pre>number of blocks - also number of entries in arrays array_of_types, array_of_displacements, and array_of_blocklengths (integer)</pre>	24 25 26	
IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)	27 28 29	
IN	array_of_displacements	byte displacement of each block (array of integers)	30	
IN	array_of_types	types of elements in each block (array of handles)	31	
OUT	newtype	new datatype (handle)	32 33	
C binding			33 34 35	
<pre>int MPI_Type_create_struct(int count, const int array_of_blocklengths[],</pre>			36 37 38	
F08 bindi	F08 binding			
	create_struct(count, arra	ay_of_blocklengths,	40 41	
array_of_displacements, array_of_types, newtype, ierror)				
		, array_of_blocklengths(count)	42 43	
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::				
array_of_displacements(count)				
TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count) TYPE(MPI_Datatype), INTENT(OUT) :: newtype				
	ER, OPTIONAL, INTENT(OUT)	• =	47	
			48	

1 F binding $\mathbf{2}$ MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, 3 ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR) 4 INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE, 5IERROR 6 INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*) 7 8 **Example 4.6** Let type1 have type map, 9 10 $\{(double, 0), (char, 8)\},\$ 11 with extent 16. Let B = (2, 1, 3), D = (0, 16, 26), and $T = (MPI_FLOAT, type1, MPI_CHAR)$. 12Then a call to MPI_TYPE_CREATE_STRUCT(3, B, D, T, newtype) returns a datatype with 13 type map, 1415 $\{(\texttt{float}, 0), (\texttt{float}, 4), (\texttt{double}, 16), (\texttt{char}, 24), (\texttt{char}, 26), (\texttt{char}, 27), (\texttt{char}, 28)\}.$ 1617That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 18 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies 19four bytes.) 20In general, let T be the array_of_types argument, where T[i] is a handle to, 21 $typemap_i = \{(type_0^i, disp_0^i), \dots, (type_{n_i-1}^i, disp_{n_i-1}^i)\},\$ 2223with extent ex_i . Let B be the array_of_blocklength argument and D be the 24 array_of_displacements argument. Let c be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{\mathsf{c}-1} \mathsf{B}[\mathsf{i}] \cdot n_i$ entries: 2526 $\{(type_0^0, disp_0^0 + \mathsf{D}[0]), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0]), \dots, \}$ 2728 $(type_0^0, disp_0^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots,$ 29 30 $(type_0^{\mathsf{C}-1}, disp_0^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1]), \dots, (type_{n_{\mathsf{C}-1}-1}^{\mathsf{C}-1}, disp_{n_{\mathsf{C}-1}-1}^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1]), \dots,$ 3132 $(type_0^{\mathsf{C}-1}, disp_0^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1] + (\mathsf{B}[\mathsf{c}-1] - 1) \cdot ex_{\mathsf{C}-1}), \dots,$ 33 34 $(type_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1] + (B[c-1]-1) \cdot ex_{C-1})\}.$ 35 36 A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent 37 to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T 38 is equal to oldtype. 39 4041 4243 44454647 48

4.1. DERIVED DATATYPES

4.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

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

C binding

<pre>int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],</pre>
<pre>const int array_of_subsizes[], const int array_of_starts[],</pre>
int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

F08 binding

<pre>MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,</pre>
array_of_starts, order, oldtype, newtype, ierror)
<pre>INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),</pre>
<pre>array_of_subsizes(ndims), array_of_starts(ndims), order</pre>
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

```
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.

Unofficial Draft for Comment Only

```
1
           The number of elements of type oldtype in each dimension of the n-dimensional ar-
\mathbf{2}
      ray and the requested subarray are specified by array_of_sizes and array_of_subsizes, re-
3
      spectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] < 1 or
4
      array_of_subsizes[i] > array_of_sizes[i].
5
           The array_of_starts contains the starting coordinates of each dimension of the subarray.
6
      Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to
\overline{7}
      specify array_of_starts[i] < 0 or array_of_starts[i] > (array_of_sizes[i] - array_of_subsizes[i]).
8
             Advice to users. In a Fortran program with arrays indexed starting from 1, if the
9
             starting coordinate of a particular dimension of the subarray is n, then the entry in
10
             array_of_starts for that dimension is n-1. (End of advice to users.)
11
12
           The order argument specifies the storage order for the subarray as well as the full array.
13
      It must be set to one of the following:
14
15
      MPI_ORDER_C The ordering used by C arrays, (i.e., row-major order)
16
17
      MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-major order)
18
           A ndims-dimensional subarray (newtype) with no extra padding can be defined by the
19
      function Subarray() as follows:
20
21
             newtype = Subarray(ndims, {size_0, size_1, ..., size_{ndims-1}},
22
                            \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\
23
                            \{start_0, start_1, \ldots, start_{ndims-1}\}, oldtype\}
^{24}
25
           Let the typemap of oldtype have the form:
26
27
             \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}
28
      where type_i is a predefined MPI datatype, and let e_x be the extent of oldtype. Then we define
29
      the Subarray() function recursively using the following three equations. Equation 4.2 defines
30
      the base step. Equation 4.3 defines the recursion step when order = MPI_ORDER_FORTRAN,
^{31}
32
      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.
33
34
35
             Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \}
                                                                                                       (4.2)
36
                      \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}\}
37
38
                  \{(\mathsf{lb}_\mathsf{marker}, 0),
               =
39
                   (type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex),
40
                   (tupe_0, disp_0 + (start_0 + 1) \times ex), \ldots, (tupe_{n-1}), \ldots
41
                            disp_{n-1} + (start_0 + 1) \times ex), \ldots
42
                   (tupe_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots,
43
44
                            (type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),
45
                   (ub_marker, size_0 \times ex)
46
47
             Subarray(ndims, {size_0, size_1, \ldots, size_{ndims-1}},
48
                                                                                                       (4.3)
```

$\{subsize_0, subsize_1, \dots, subsize_{ndims-1}\},\$	1
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	2
= Subarray($ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\},\$	3
$\{subsize_1, subsize_2, \dots, subsize_{ndims-1}\},\$	5
$\{start_1, start_2, \ldots, start_{ndims-1}\},\$	6
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$	7
	8
Subarray($ndims$, { $size_0, size_1, \dots, size_{ndims-1}$ }, (4.4)	9 10
$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$	11
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype)$	12
= Subarray($ndims - 1, \{size_0, size_1, \dots, size_{ndims-2}\},\$	13
$\{subsize_0, subsize_1, \dots, subsize_{ndims-2}\},\$	14
$\{start_0, start_1, \dots, start_{ndims-2}\},\$	15
	16
$Subarray(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, oldtype))$	17
	18

For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 13.9.2.

4.1.4 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [42] data distributions. However, unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call this constructor with identical arguments (with the exception of rank which should be set appropriately). These filetypes (along with identical disp and etype) are then used to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 13.1.1 and Section 13.3. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. (*End of advice to users.*)

 24

 $25 \\ 26$

```
1
     MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs, array_of_dargs,
\mathbf{2}
                     array_of_psizes, order, oldtype, newtype)
3
       IN
                 size
                                              size of process group (non-negative integer)
4
       IN
                 rank
                                              rank in process group (non-negative integer)
5
6
       IN
                 ndims
                                              number of array dimensions as well as process grid
7
                                              dimensions (integer)
8
                 array_of_gsizes
       IN
                                              number of elements of type oldtype in each dimension
9
                                              of global array (array of non-negative integers)
10
       IN
                 array_of_distribs
                                              distribution of array in each dimension (array of non-
11
                                              negative integers)
12
13
       IN
                 array_of_dargs
                                              distribution argument in each dimension (array of non-
14
                                              negative integers)
15
       IN
                 array_of_psizes
                                              size of process grid in each dimension (array of non-
16
                                              negative integers)
17
       IN
                 order
                                              array storage order flag (state)
18
19
       IN
                 oldtype
                                              old datatype (handle)
20
       OUT
                 newtype
                                              new datatype (handle)
21
22
     C binding
23
     int MPI_Type_create_darray(int size, int rank, int ndims,
24
                     const int array_of_gsizes[], const int array_of_distribs[],
25
                     const int array_of_dargs[], const int array_of_psizes[],
26
                     int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
27
28
     F08 binding
     MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
29
30
                     array_of_distribs, array_of_dargs, array_of_psizes, order,
^{31}
                     oldtype, newtype, ierror)
          INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
32
33
          array_of_distribs(ndims), array_of_dargs(ndims),
34
          array_of_psizes(ndims), order
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
35
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
36
37
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
38
     F binding
39
     MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
40
                     ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
41
                     OLDTYPE, NEWTYPE, IERROR)
42
          INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
43
          ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR
44
45
          MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding
46
     to the distribution of an ndims-dimensional array of oldtype elements onto an
47
     ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be
48
     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.

Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 7. (*End of advice to users.*)

Each dimension of the array can be distributed in one of three ways:

- MPI_DISTRIBUTE_BLOCK Block distribution
- MPI_DISTRIBUTE_CYCLIC Cyclic distribution
- MPI_DISTRIBUTE_NONE Dimension not distributed.

The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify array_of_dargs[i] * array_of_psizes[i] < array_of_gsizes[i].

For example, the HPF layout ARRAY(CYCLIC(15)) corresponds to MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout AR-RAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of MPI_DISTRIBUTE_DFLT_DARG.

The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the storage order. Therefore, arrays described by this type constructor may be stored in Fortran (column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN and MPI_ORDER_C.

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

Without loss of generality, it suffices to define the typemap for the MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.

MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_NONE can be reduced to the MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.

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

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

If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and MPI_DISTRIBUTE_CYCLIC are equivalent.

MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to array_of_gsizes[i].

Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to 1.

For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined by the following code fragment:

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

	1
$(type_0, disp_0 + (r \times darg + 1) \times ex), \dots,$	2
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex),$	3
	4
$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \dots,$	5
$(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex),$	6
	7
$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots,$	8
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex),$	9 10
$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \dots,$	11
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),$	12
$(g_{p} \circ n = 1, u \circ p_{n} = 1 + (r + u u \circ g + 1) + (v + p \circ n \circ r + u \circ n \circ g + v \circ u),$	13
$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \dots,$	14
$(type_0, usp_0 + ((t+1) \land uurg - 1) \land ex + psize \land uurg \land ex), \dots, $ $(type_{n-1}, disp_{n-1} + ((r+1) \land darg - 1) \land ex + psize \land darg \land ex),$	15
$(type_{n-1}, aisp_{n-1} + ((r+1) \times aarg - 1) \times ex + psize \times aarg \times ex),$	16
	17 18
$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots,$	19
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),$	20
$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \dots,$	21
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex$	22
$+psize \times darg \times ex \times (count - 1)),$	23
(b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b b a b c (b a b a b c (b b a b c (b a b a b a b a b a b a b a b a b a b	24
$(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex$	25
	26 27
$+psize \times darg \times ex \times (count - 1)), \dots,$	28
$(type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex$	29
$+psize \times darg \times ex \times (count - 1)),$	30
$(ub_marker, gsize * ex) \}$	31
where <i>count</i> is defined by this code fragment:	32
	33
<pre>nblocks = (gsize + (darg - 1)) / darg; count = pblocks (psize;</pre>	34
count = nblocks / psize; left_over = nblocks - count * psize;	35 36
if (r < left_over)	37
count = count + 1;	38
	39
Here, <i>nblocks</i> is the number of blocks that must be distributed among the processors. Finally, $darg_{last}$ is defined by this code fragment:	40
I many, aur glast is defined by this code magnetic.	41
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)	42
<pre>darg_last = darg;</pre>	43

```
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
    darg_last = darg;
else {
    darg_last = num_in_last_cyclic - darg * r;
    if (darg_last > darg)
        darg_last = darg;
    if (darg_last <= 0)</pre>
```

Unofficial Draft for Comment Only

```
1
                   darg_last = darg;
\mathbf{2}
              }
3
4
     Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:
5
6
            <oldtype> FILEARRAY(100, 200, 300)
7
     !HPF$ PROCESSORS PROCESSES(2, 3)
8
     !HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
9
     This can be achieved by the following Fortran code, assuming there will be six processes
10
     attached to the run:
11
12
          ndims = 3
13
          array_of_gsizes(1) = 100
14
          array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
15
          array_of_dargs(1) = 10
16
          array_of_gsizes(2) = 200
17
          array_of_distribs(2) = MPI_DISTRIBUTE_NONE
18
          \operatorname{array_of_dargs}(2) = 0
19
          array_of_gsizes(3) = 300
20
          array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
21
          array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
22
          array_of_psizes(1) = 2
23
          array_of_psizes(2) = 1
24
          array_of_psizes(3) = 3
25
          call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
26
          call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
27
          call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
28
               array_of_distribs, array_of_dargs, array_of_psizes,
                                                                                   &
29
                MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
30
^{31}
            Address and Size Functions
     4.1.5
32
33
     The displacements in a general datatype are relative to some initial buffer address. Abso-
34
     lute addresses can be substituted for these displacements: we treat them as displacements
35
     relative to "address zero," the start of the address space. This initial address zero is in-
36
     dicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address
37
     of the entries in the communication buffer, in which case the buf argument is passed the
38
     value MPI_BOTTOM. Note that in Fortran MPI_BOTTOM is not usable for initialization or
39
     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
```

```
^{46} pages 16 and 121.
```

```
47
48
```

Rationale. 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.*)

MPI_GET_ADDRESS(location, address)

IN	location	location in caller memory (choice)
OUT	address	address of location (integer)

C binding

int MPI_Get_address(const void *location, MPI_Aint *address)

F08 binding

MPI_Get_address(location, address, ierror)
 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location
 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

Returns the (byte) address of location.

Rationale. In the mpi_f08 module, the location argument is not defined with INTENT(IN) because existing applications may use MPI_GET_ADDRESS as a substitute for MPI_F_SYNC_REG that was not defined before MPI-3.0. (*End of rationale.*)

Example 4.8 Using MPI_GET_ADDRESS for an array.

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

Advice to users. C users may be tempted to avoid the usage of ⁴³ MPI_GET_ADDRESS and rely on the availability of the address operator &. Note, ⁴⁴ however, that & cast-expression is a pointer, not an address. ISO C does not require ⁴⁵ that the value of a pointer (or the pointer cast to int) be the absolute address of the ⁴⁶ object pointed at — although this is commonly the case. Furthermore, referencing ⁴⁷ may not have a unique definition on machines with a segmented address space. The ⁴⁸

Unofficial Draft for Comment Only

 24

us	e of MPI_GET_ADD	ORESS to "reference" C variables guarantees portability to such
ma	achines as well. (En	nd of advice to users.)
op		To prevent problems with the argument copying and register Fortran compilers, please note the hints in Sections $17.1.10-$ ce to users.)
	· · · · · · · · · · · · · · · · · · ·	arithmetic on MPI addresses must be performed using the AINT_DIFF functions.
1PI_AII	NT_ADD(base, disp)	
IN	base	base address (integer)
IN	disp	displacement (integer)
PI_Ain 08 bir		PI_Aint base, MPI_Aint disp)
NTEGER	(KIND=MPI_ADDRES	S_KIND) MPI_Aint_add(base, disp) DRESS_KIND), INTENT(IN) :: base, disp
	(KIND=MPI_ADDRES	S_KIND) MPI_AINT_ADD(BASE, DISP) DRESS_KIND) BASE, DISP
he base MPI_GE lress is n the sa ormed i	and disp argument $T_ADDRESS$ and d valid only at the prame object reference n a manner that res	uces a new MPI_Aint value that is equivalent to the sum of ts, where base represents a base address returned by a call to lisp represents a signed integer displacement. The resulting ad- ocess that generated base, and it must correspond to a location ed by base, as described in Section 4.1.12. The addition is per- ults in the correct MPI_Aint representation of the output address, ally produced base had called:
MPI_Get	_address((char *) base + disp, &result);
MPI_AII	NT_DIFF(addr1, add	lr2)
IN	addr1	minuend address (integer)
IN	addr2	subtrahend address (integer)
C bindi PI_Ain	0	MPI_Aint addr1, MPI_Aint addr2)
	(KIND=MPI_ADDRES	S_KIND) MPI_Aint_diff(addr1, addr2) DRESS_KIND), INTENT(IN) :: addr1, addr2
F bindi	ng	

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

MPI_AINT_DIFF produces a new MPI_Aint value that is equivalent to the difference between addr1 and addr2 arguments, where addr1 and addr2 represent addresses returned by calls to MPI_GET_ADDRESS. The resulting address is valid only at the process that generated addr1 and addr2, and addr1 and addr2 must correspond to locations in the same object in the same process, as described in Section 4.1.12. The difference is calculated in a manner that results in the signed difference from addr1 to addr2, as if the process that originally produced the addresses had called (char *) addr1 - (char *) addr2 on the addresses initially passed to MPI_GET_ADDRESS.

The following auxiliary functions provide useful information on derived datatypes.

MPI_TYPE_SIZE(datatype, size)

IN	datatype	datatype (handle)
OUT	size	datatype size (non-negative integer)

C binding

int MPI_Type_size(MPI_Datatype datatype, int *size)

F08 binding

MPI_Type_size(datatype, size, ierror)							
TYPE(MPI_Datatype), INTENT(IN) ::	datatype						
INTEGER, INTENT(OUT) :: size							
INTEGER, OPTIONAL, INTENT(OUT) ::	ierror						

F binding

MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, SIZE, IERROR

MPI_TYPE_SIZE_X(datatype, size)

IN	datatype	datatype (handle)
OUT	size	datatype size (integer)

C binding

```
int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)
```

F08 binding

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

F binding

```
MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
INTEGER DATATYPE, IERROR
```

 $\mathbf{2}$

1 2

3

4

5

6

7

8 9

10

INTEGER(KIND=MPI_COUNT_KIND) SIZE

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.

4.1.6 Lower-Bound and Upper-Bound Markers

11It is often convenient to define explicitly the lower bound and upper bound of a type map, 12and override the definition given on page 110. This allows one to define a datatype that has 13"holes" at its beginning or its end, or a datatype with entries that extend above the upper 14bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. 15Also, the user may want to overide the alignment rules that are used to compute upper 16bounds and extents. E.g., a C compiler may allow the user to overide default alignment 17rules for some of the structures within a program. The user has to specify explicitly the 18 bounds of the datatypes that match these structures.

¹⁹ To achieve this, we add two additional conceptual datatypes, **lb_marker** and ²⁰ **ub_marker**, that represent the lower bound and upper bound of a datatype. These con-²¹ ceptual datatypes occupy no space (*extent*(**lb_marker**) = *extent*(**ub_marker**) = 0). They do ²² not affect the size or count of a datatype, and do not affect the content of a message created ²³ with this datatype. However, they do affect the definition of the extent of a datatype and, ²⁴ therefore, affect the outcome of a replication of this datatype by a datatype constructor.

26**Example 4.9** A call to MPI_TYPE_CREATE_RESIZED(MPI_INT, -3, 9, type1) creates a 27new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer 28 at displacement 0. This is the datatype defined by the typemap $\{(\mathsf{lb}_\mathsf{marker}, -3), (int, 0), \}$ 29(ub_marker, 6)}. If this type is replicated twice by a call to MPI_TYPE_CONTIGUOUS(2, 30 type1, type2) then the newly created type can be described by the typemap {(lb_marker, 31 -3), (int, 0), (int,9), (ub_marker, 15)}. (An entry of type ub_marker can be deleted if there 32 is another entry of type ub_marker with a higher displacement; an entry of type lb_marker 33 can be deleted if there is another entry of type lb_marker with a lower displacement.)

In general, if

34

35 36

37

43

44 45

48

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

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

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

 $_{42}$ Similarly, the **upper bound** of *Typemap* is defined to be

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

46 47 Then

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

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

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

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

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

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

Example: Instead of

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

one should define

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

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 2 3 4 5	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.				
6 7 8 9 10 11	In principle, a compiler may define an additional alignment rule for structures, e.g., to use at least 4 or 8 byte alignment, although the content may have a max_ik_i alignment less than this structure alignment. To maintain portability, users should always resize structure derived datatype handles if used in an array of structures, see the Example in Section 17.1.15. (End of advice to users.)				
12 13 14	4.1.7 E	xtent and Bounds of Dataty	rpes		
15 16	MPI_TYF	PE_GET_EXTENT(datatype,	lb, extent)		
17	IN	datatype	datatype to get information on (handle)		
18	OUT	lb	lower bound of datatype (integer)		
19 20	OUT	extent	extent of datatype (integer)		
21					
22	C bindi	•			
23 24	int MPI_		atype datatype, MPI_Aint *1b,		
24 25		MPI_Aint *extent)			
26	F08 binding MPI_Type_get_extent(datatype, lb, extent, ierror)				
27		e_get_extent(datatype, 1) 2(MPI_Datatype), INTENT(2			
28		GER(KIND=MPI_ADDRESS_KI			
29 30	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
31	F binding				
32	MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)				
33	INTEGER DATATYPE, IERROR				
34 35	INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT				
36					
37	MPI_TYF	PE_GET_EXTENT_X(datatyp	pe, lb, extent)		
38 39	IN	datatype	datatype to get information on (handle)		
40	OUT	lb	lower bound of datatype (integer)		
41	OUT	extent	extent of datatype (integer)		
42					
43 44	C bindin	ng			
44 45	int MPI_		atatype datatype, MPI_Count *1b,		
46		MPI_Count *extent)			
47	F08 bine	0			
48	МРІ_Туре	e_get_extent_x(datatype,	lb, extent, ierror)		

TYPE(MPI_Datatype), INTENT(IN)) :: datatype	1	
INTEG	ER(KIND=MPI_COUNT_KIND),	INTENT(OUT) :: lb, extent	2	
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	3	
F binding	r		4	
C C	GET_EXTENT_X(DATATYPE, LI	3 EXTENT TERROR)	5	
	ER DATATYPE, IERROR	, <u>minu</u> , <u>inivit</u> ,	6	
	ER(KIND=MPI_COUNT_KIND) 1	LB. EXTENT	7	
			8	
		xtent of datatype (as defined in Equation 4.1).	9	
	, _	parameter cannot express the value to be returned	10	
(0 /	-	ld the output value), it is set to MPI_UNDEFINED.	11	
	8	of a datatype, using lower bound and upper bound	12	
markers. 7	This provides control over the	e stride of successive datatypes that are replicated	13	
by datatyp	be constructors, or are replicat	ed by the count argument in a send or receive call.	14	
	E_CREATE_RESIZED(oldtype,	In extent neutrine)	16	
		ib, extent, newtype)	17	
IN	oldtype	input datatype (handle)	18	
IN	lb	new lower bound of datatype (integer)	19	
IN	outopt		20	
IIN	extent	new extent of datatype (integer)	21	
OUT	newtype	output datatype (handle)	22	
			23	

```
C binding
```

<pre>int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,</pre>
MPI_Aint extent, MPI_Datatype *newtype)

```
F08 binding
```

```
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
    F binding
```

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
INTEGER OLDTYPE, NEWTYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
```

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

 24

 31

```
1
     4.1.8
           True Extent of Datatypes
\mathbf{2}
     Suppose we implement gather (see also Section 5.5) as a spanning tree implemented on
3
     top of point-to-point routines. Since the receive buffer is only valid on the root pro-
4
     cess, one will need to allocate some temporary space for receiving data on intermedi-
5
     ate nodes. However, the datatype extent cannot be used as an estimate of the amount
6
     of space that needs to be allocated, if the user has modified the extent, for example
7
     by using MPI_TYPE_CREATE_RESIZED. The functions MPI_TYPE_GET_TRUE_EXTENT
8
     and MPI_TYPE_GET_TRUE_EXTENT_X are provided which return the true extent of the
9
     datatype.
10
11
12
     MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)
13
       IN
                datatype
                                            datatype to get information on (handle)
14
       OUT
                true_lb
                                            true lower bound of datatype (integer)
15
16
       OUT
                true_extent
                                            true size of datatype (integer)
17
18
     C binding
19
     int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
20
                    MPI_Aint *true_extent)
21
22
     F08 binding
23
     MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
^{24}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     F binding
28
     MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
29
         INTEGER DATATYPE, IERROR
30
         INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
^{31}
32
33
     MPI_TYPE_GET_TRUE_EXTENT_X(datatype, true_lb, true_extent)
34
35
       IN
                datatype
                                            datatype to get information on (handle)
36
       OUT
                true_lb
                                            true lower bound of datatype (integer)
37
       OUT
                true_extent
                                            true size of datatype (integer)
38
39
     C binding
40
41
     int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
42
                    MPI_Count *true_extent)
43
     F08 binding
44
     MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
47
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

F binding

```
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(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 lower bound of the corresponding typemap, ignoring explicit lower bound markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring explicit lower bound and upper bound markers, and performing no rounding for alignment. If the typemap associated with datatype is

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

Then

$$true_lb(Typemap) = min_j \{ disp_j : type_j \neq \mathsf{lb_marker}, \mathsf{ub_marker} \},$$

$$true_ub(Typemap) = max_j \{ disp_j + size of(type_j) : type_j \neq \mathsf{lb_marker}, \mathsf{ub_marker} \},$$

and

$$true_extent(Typemap) = true_ub(Typemap) - true_lb(typemap).$$

(Readers should compare this with the definitions in Section 4.1.6 and Section 4.1.7, which describe the function MPI_TYPE_GET_EXTENT.)

The true_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed.

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

4.1.9 Commit and Free

A datatype object has to be **committed** before it can be used in a communication. As an argument in datatype constructors, uncommitted and also committed datatypes can be used. There is no need to commit basic datatypes. They are "pre-committed."

MPI_TYPE_COMMIT(datatype)

datatype

datatype that is committed (handle)

C binding

INOUT

int MPI_Type_commit(MPI_Datatype *datatype)

F08 binding
MPI_Type_commit(datatype, ierror)

```
TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

MPI_TYPE_COMMIT(DATATYPE, IERROR) INTEGER DATATYPE, IERROR $\mathbf{2}$

1 The commit operation commits the datatype, that is, the formal description of a com- $\mathbf{2}$ munication buffer, not the content of that buffer. Thus, after a datatype has been commit-3 ted, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, 4 the content of different buffers, with different starting addresses. 5The system may "compile" at commit time an internal 6 Advice to implementors. representation for the datatype that facilitates communication, e.g., change from a 7 compacted representation to a flat representation of the datatype, and select the most 8 convenient transfer mechanism. (End of advice to implementors.) 9 10 MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent 11 to a no-op.

```
12
13
14
```

Example 4.10 The following code fragment gives examples of using MPI_TYPE_COMMIT.

```
15
     INTEGER type1, type2
16
     CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
17
                     ! new type object created
18
     CALL MPI_TYPE_COMMIT(type1, ierr)
19
                     ! now type1 can be used for communication
20
     type2 = type1
21
                     ! type2 can be used for communication
22
                     ! (it is a handle to same object as type1)
23
     CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
^{24}
                     ! new uncommitted type object created
25
     CALL MPI_TYPE_COMMIT(type1, ierr)
26
                    ! now type1 can be used anew for communication
27
28
29
     MPI_TYPE_FREE(datatype)
30
^{31}
       INOUT
                datatype
                                           datatype that is freed (handle)
32
33
     C binding
34
     int MPI_Type_free(MPI_Datatype *datatype)
35
     F08 binding
36
     MPI_Type_free(datatype, ierror)
37
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     F binding
41
     MPI_TYPE_FREE(DATATYPE, IERROR)
42
         INTEGER DATATYPE, IERROR
43
         Marks the datatype object associated with datatype for deallocation and sets datatype
44
     to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will
45
```

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.

4.1. DERIVED DATATYPES

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

 OUT
 newtype

 copy of oldtype (handle)

C binding

int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)

F08 binding

<pre>MPI_Type_dup(oldtype, newtype, ierror)</pre>	
TYPE(MPI_Datatype), INTENT(IN) ::	oldtype
TYPE(MPI_Datatype), INTENT(OUT) ::	newtype
INTEGER, OPTIONAL, INTENT(OUT) ::	ierror

F binding

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

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.

4.1.11 Use of General Datatypes in Communication

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

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm)
MPI_TYPE_FREE(newtype).
```

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

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

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10

11

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

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

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

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

17 18 19

16

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

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

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

Example 4.11 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
32
     . . .
33
     CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2, ...)
34
     CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4, ...)
35
     CALL MPI_TYPE_CONTIGUOUS(2, type2, type22, ...)
36
     . . .
37
     CALL MPI_SEND(a, 4, MPI_REAL, ...)
38
     CALL MPI_SEND(a, 2, type2, ...)
39
     CALL MPI_SEND(a, 1, type22, ...)
40
     CALL MPI_SEND(a, 1, type4, ...)
41
     . . .
42
     CALL MPI_RECV(a, 4, MPI_REAL, ...)
43
     CALL MPI_RECV(a, 2, type2, ...)
^{44}
     CALL MPI_RECV(a, 1, type22, ...)
45
     CALL MPI_RECV(a, 1, type4, ...)
46
     Each of the sends matches any of the receives.
47
48
```

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_TYPE_GET_ELEMENTS(status, datatype, count)

IN	status	return status of receive operation (Status)	1
IN	datatype	datatype used by receive operation (handle)	1
OUT	count	number of received basic elements (non-negative inte-	1
		$\operatorname{ger})$	1

C binding

F08 binding

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

F binding

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

MPI_TYPE_GET_ELEMENTS_X(status, datatype, count)

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

C binding

F08 binding

MPI_Type_get_elements_x(status, datatype, count, ierror)
 TYPE(MPI_Status), INTENT(IN) :: status

Unofficial Draft for Comment Only

 24

 $45 \\ 46$

1 2 3	TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4 5 6 7 8	<pre>F binding MPI_TYPE_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) COUNT</pre>
9 10 11 12	The datatype argument should match the argument provided by the receive call that set the status variable. For both functions, if the OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.
13 14 15 16 17 18 19 20 21	The previously defined function MPI_GET_COUNT (Section 3.2.5), has a different behavior. It returns the number of "top-level entries" received, i.e. the number of "copies" of type datatype. In the previous example, MPI_GET_COUNT may return any integer value k , where $0 \le k \le \text{count}$. If MPI_GET_COUNT returns k , then the number of basic elements received (and the value returned by MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X) is $n \cdot k$. If the number of basic elements received is not a multiple of n , that is, if the receive operation has not received an integral number of datatype "copies," then MPI_GET_COUNT sets the value of count to MPI_UNDEFINED.
22 23 24 25	<pre>Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)</pre>
26 27 28 29 30 31	CALL MPI_TYPE_COMMIT(Type2, ierr) CALL MPI_COMM_RANK(comm, rank, ierr) IF (rank.EQ.0) THEN CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr) CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr) ELSE IF (rank.EQ.1) THEN
32 33 34 35 36 37 38 39	CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr) CALL MPI_GET_COUNT(stat, Type2, i, ierr) ! returns i=1 CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=2 CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr) CALL MPI_GET_COUNT(stat, Type2, i, ierr) ! returns i=MPI_UNDEFINED CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr) ! returns i=3 END IF
40 41 42 43 44 45	The functions MPI_GET_ELEMENTS and MPI_GET_ELEMENTS_X can also be used after a probe to find the number of elements in the probed message. Note that the MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X return the same values when they are used with basic datatypes as long as the limits of their respective count arguments are not exceeded.
46 47 48	<i>Rationale.</i> The extension given to the definition of MPI_GET_COUNT seems natural: one would expect this function to return the value of the count argument, when the receive buffer is filled. Sometimes datatype represents a basic unit of data one wants

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

The definition implies that a receive cannot change the Advice to implementors. 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 41 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.

4748

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 $\overline{7}$

8

9

10

11

12

13

14

151617

18

19

20

21

22

23

 24

25

2627

28

2930

 31

32

33

34 35

36

37

38

39

40

42

43

44

45

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

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

10 11 12

1

 $\mathbf{2}$

3

4

5

6

7

8

9

4.1.13 Decoding a Datatype

13 MPI datatype objects allow users to specify an arbitrary layout of data in memory. There 14are several cases where accessing the layout information in opaque datatype objects would 15be useful. The opaque datatype object has found a number of uses outside MPI. Further-16 more, a number of tools wish to display internal information about a datatype. To achieve 17this, datatype decoding functions are provided. The two functions in this section are used 18 together to decode datatypes to recreate the calling sequence used in their initial defini-19 tion. These can be used to allow a user to determine the type map and type signature of a 20datatype. 21

```
22
23
```

 24

25

37

```
MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, com-
biner)
```

20			
26	IN	datatype	datatype to access (handle)
27 28	OUT	num_integers	number of input integers used in call constructing combiner (integer)
29 30 31	OUT	num_addresses	number of input addresses used in call constructing combiner (integer)
32 33	OUT	num_datatypes	number of input datatypes used in call constructing combiner (integer) $% \left(\left({{{\left({{{\left({{{\left({{{\left({{{c}}} \right)}} \right.} \right.} \right)}}}} \right)$
34 35	OUT	combiner	combiner (state)

```
<sup>36</sup> C binding
```

```
_{40} F08 binding
```

```
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
```

```
^{47}_{48} F binding
```

MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR) INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR

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

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

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

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

	a named predefined deteture	25
MPI_COMBINER_NAMED	a named predefined datatype	26
MPI_COMBINER_DUP	MPI_TYPE_DUP	27
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS	28
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR	
MPI_COMBINER_HVECTOR	MPI_TYPE_CREATE_HVECTOR	29
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED	30
MPI_COMBINER_HINDEXED	MPI_TYPE_CREATE_HINDEXED	31
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK	32
MPI_COMBINER_HINDEXED_BLOCK	MPI_TYPE_CREATE_HINDEXED_BLOCK	33
MPI_COMBINER_STRUCT	MPI_TYPE_CREATE_STRUCT	34
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY	35
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY	36
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL	37
	MPI_TYPE_CREATE_F90_COMPLEX	38
MPI_COMBINER_F90_COMPLEX		39
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER	40
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED	41
		42
		42

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.

 24

MPI_TYF	· · ·	/pe, max_integers, max_addresses, max_datatypes, ar- /_of_addresses, array_of_datatypes)			
IN	datatype	datatype to access (handle)			
IN	max_integers	number of elements in array_of_integers (integer)			
	C C				
IN	max_addresses	number of elements in array_of_addresses (integer)			
IN	max_datatypes	number of elements in array_of_datatypes (integer)			
Ουτ	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 constructing datatype (array of handles)			
C bindiı	no				
	•	Datatype datatype, int max_integers,			
		, int max_datatypes, int array_of_integers[],			
	MPI_Aint array_of				
	MPI_Datatype arra	y_of_datatypes[])			
F08 binding					
<pre>MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,</pre>					
	• •	<pre>, array_of_addresses, array_of_datatypes,</pre>			
TVDE	ierror) E(MPI_Datatype), INTENT((IN) ··· datatuma			
	• -	<pre>x_integers, max_addresses, max_datatypes</pre>			
		rray_of_integers(max_integers)			
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::					
array_of_addresses(max_addresses)					
TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)					
INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
F bindir	ıg				
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,					
		, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,			
IERROR)					
		GERS, MAX_ADDRESSES, MAX_DATATYPES, Y_OF_DATATYPES(*), IERROR			
	-	I_OF_DATATIFES(*), TERROR IND) ARRAY_OF_ADDRESSES(*)			
	51 X	nnamed or a derived datatype; the call is erroneous if			
datatype is a predefined named datatype. The values given for max_integers, max_addresses, and max_datatypes must be at least as					
		egers, num_addresses, and num_datatypes must be at least as			
0		OPE for the same datatype argument.			
	0	x_integers, max_addresses, and max_datatypes allow for			
erro	or checking in the call. (<i>End</i>	a of rationale.)			

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

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

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

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

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

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

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

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

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

Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$

6 7

8

9

10

11 12

13

14

15

16

17 18

19

20

21

22

23 24

25

26

27

28 29

30

 31

32

33

34

35

36

37 38

39

40

41

42

43

1	PARAMETER	(LARGE = 1000)			
2	INTEGER T	YPE, NI, NA, ND, COMBI	NER. I	(LARGE). D(LARG	E). IERROR
3		KIND=MPI_ADDRESS_KIND)			
4					
		DATATYPE TYPE (NOT SH			
5	CALL MPI_	TYPE_GET_ENVELOPE(TYPE	, NI,	NA, ND, COMBINE	R, IERROR)
6	IF ((NI .	GT. LARGE) .OR. (NA .G	T. LAF	GE) .OR. (ND .G	Γ. LARGE)) THEN
7	WRITE (*, *) "NI, NA, OR ND =	". NI	. NA. ND. &	
8		NED BY MPI_TYPE_GET_EN			IARCE = " IARCE
9					LANGE - , LANGE
		I_ABORT(MPI_COMM_WORLD	, 99,	IERROR)	
10	ENDIF				
11	CALL MPI_	TYPE_GET_CONTENTS (TYPE	, NI,	NA, ND, I, A, D	, IERROR)
12					
13	or in C the analog	gous calls of:			
14					
	#define LARGE 1	000			
15	int ni, na, nd,	<pre>combiner, i[LARGE];</pre>			
16	MPI_Aint a[LARG	E]:			
17	MPI_Datatype ty				
18		-	× . /		
19		tatype type (not shown			
		velope(type, ∋, &na,			
20	if ((ni > LARGE	L) (na > LARGE) (:	nd > I	.ARGE)) {	
21	fprintf(std	err, "ni, na, or nd = '	%d %d	%d returned by '	", ni, na, nd);
22	-	err, "MPI_Type_get_env		•	
23	-	GE);	oropo	10 101801 011011	Jiiitold 7,64 (ii)
24					
25		PI_COMM_WORLD, 99);			
26	};				
	MPI_Type_get_co	ntents(type, ni, na, n	d, i,	a, d);	
27				<u>^</u>	
28	In the descrip	ptions that follow, the lower	r case i	name of arguments	is used.
29	If combiner i	s MPI_COMBINER_NAMED	then i	t is erroneous to	call
30	MPI_TYPE_GET_	CONTENTS.			
31		MPI_COMBINER_DUP ther	h		
32			-		
		Constructor argument	С	Fortran location	
33		oldtype	d[0]	D(1)	
34		olutype	u[0]	D(1)	
35	and $ni = 0$, $na =$	$0 \mathrm{nd} = 1$			
36	/	B MPI_COMBINER_CONTIGU	0115 +1	nen	
37	II COMDINEL 18		005 11	1611	
		Constructor argument	С	Fortran location	
38					
39		count	i[0]	I(1)	
40		oldtype	d[0]	D(1)	
41	1 • •				
42	and $ni = 1$, $na =$	·			
43	If combiner is	MPI_COMBINER_VECTOR	then		
44		Constructor argument	\mathbf{C}	Fortran location	
45		count	i[0]	I(1)	•
46		blocklength	i[1]	I(2)	
47		stride			
48			i[2]	I(3)	
-10		oldtype	d[0]	D(1)	

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

If combiner is $\mathsf{MPI_COMBINER_HVECTOR}$ then

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

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

If combiner is MPI_COMBINER_INDEXED then

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

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

If combiner is MPI_COMBINER_HINDEXED then

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

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]	$\mathrm{D}(1)$

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]	$\mathrm{D}(1)$

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

If combiner is MPI_COMBINER_STRUCT then

 $44 \\ 45$

	Constructor and		0	Fortron loost	ion
-	Constructor arg	gument	C ;[0]	Fortran locat	.1011
	count	onotha	i[0]	I(1) $I(2) \neq I(I(1))$	+ 1)
	array_of_blockl array_of_displa	-	i[1] to i[i[0]]		,
	array_of_displa array_of_types		a[0] to a[i[0]- d[0] to d[i[0]-		
-	array_or_types	((1))
	t+1, na = count	,			
If combine	er is MPI_COMBI	NER_SUBAH	RAY then		
Constru	uctor argument		С	Fortran lo	cation
ndims			i[0]	I(1)	
array_c	of_sizes	i[1] t	o i[i[0]]	I(2) to $I(I($	(1)+1)
array_c	of_subsizes	i[i[0]+1]	to $i[2*i[0]]$	I(I(1)+2) to $I($	(2*I(1)+1)
array_c	of_starts	i[2*i[0]+1]] to $i[3*i[0]]$	I(2*I(1)+2) to 1	I(3*I(1)+1)
order			i[0]+1]	I(3*I(1))	+2]
oldtype	e	-	1[0]	D(1)	-
	er is MPI_COMBI		C	Fortran l	ocation
size	argument		<u>c</u> i[0]	I(1	
rank			i[1]	I(1 I(2	,
ndims			i[1]	I(2 I(3	,
array_of	rsizes		i[2] i[i[2]+2]	I(4) to $I(4)$,
	_gsizes _distribs		i[1[2]+2] o $i[2*i[2]+2]$		
-				I(I(3)+4) to I $I(2*I(3)+4)$ to	
arrav of		1 2 1 2 + 3	UU 10 14 T4		
array_of array_of					
array_of		i[3*i[2]+3]	to $i[4*i[2]+2$] $I(3*I(3)+4)$ to	I(4*I(3)+3)
array_of order		i[3*i[2]+3] i[4*i]	to $i[4*i[2]+2$ i[2]+3]] $I(3*I(3)+4)$ to I(4*I(3)+4)	I(4*I(3)+3)+4)
array_of		i[3*i[2]+3] i[4*i]	to $i[4*i[2]+2$] $I(3*I(3)+4)$ to	I(4*I(3)+3)+4)
array_of order oldtype and ni = 4^* ndi	psizes ms+4, $na = 0$, $na = 0$	i[3*i[2]+3] i[4*2] i[4*2] nd = 1.	to $i[4*i[2]+2$ i[2]+3] i[0]] $I(3*I(3)+4)$ to I(4*I(3)+4)	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4^* ndi	_psizes	i[3*i[2]+3] i[4*2] i[4*2] nd = 1.	to $i[4*i[2]+2$ i[2]+3] i[0]] $I(3*I(3)+4)$ to I(4*I(3)+4)	I(4*I(3)+4)
array_of order oldtype and ni = 4^* ndi	prizes $prims+4, na = 0, no er is MPI_COMBIN Construct$	i[3*i[2]+3] i[4*2] i[4*2] nd = 1.	to i[4*i[2]+2 i[2]+3] i[0] EAL then ent C Fo] I(3*I(3)+4) to I(4*I(3 D(3	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4^* ndi	$\frac{1}{p}$ psizes $\frac{1}{p}$ ims+4, na = 0, no er is MPI_COMBIN $\frac{1}{p}$	i[3*i[2]+3] i[4*2] nd = 1. NER_F90_RI	to $i[4*i[2]+2$ i[2]+3] l[0] EAL then ent C Fe i[0]	$\frac{I(3*I(3)+4) \text{ to}}{I(4*I(3)+4) \text{ to}}$	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4^* ndi	prizes $prims+4, na = 0, no er is MPI_COMBIN Construct$	i[3*i[2]+3] i[4*2] nd = 1. NER_F90_RI	to i[4*i[2]+2 i[2]+3] i[0] EAL then ent C Fo] I(3*I(3)+4) to I(4*I(3 D(3	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4*ndi If combine	$\frac{1}{p}$ prizes $\frac{1}{p}$	i[3*i[2]+3] i[4*2] nd = 1. NER_F90_RI	to $i[4*i[2]+2$ i[2]+3] l[0] EAL then ent C Fe i[0]	$\frac{I(3*I(3)+4) \text{ to}}{I(4*I(3)+4) \text{ to}}$	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4 *ndi If combine and ni = 2, na	$\frac{1}{1} \text{psizes}$ $\frac{1}{1} \text{psizes}$ $\frac{1}{1} \text{psizes}$ $\frac{1}{1} \text{Construct}$ $\frac{1}{1} \text{construct}$ $\frac{1}{1} \text{construct}$ $\frac{1}{1} \text{psizes}$ $\frac{1}{1} \text{construct}$ $\frac{1}{1}$	i[3*i[2]+3] i[4*] nd = 1. NER_F90_RI	to $i[4*i[2]+2$ i[2]+3] l[0] EAL then ent C For $i[0]i[1]$	$\frac{I(3*I(3)+4) \text{ to}}{I(4*I(3)+4) \text{ to}}$	I(4*I(3)+4)
array_of order oldtype and ni = 4 *ndi If combine and ni = 2, na	$\frac{1}{p}$ prizes $\frac{1}{p}$	i[3*i[2]+3] i[4*] nd = 1. NER_F90_RI	to $i[4*i[2]+2$ i[2]+3] l[0] EAL then ent C For $i[0]i[1]$	$\frac{I(3*I(3)+4) \text{ to}}{I(4*I(3)+4) \text{ to}}$	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4 *ndi If combine and ni = 2, na	$\frac{1}{p} = 0, \text{ nd} = 0.$ $\frac{1}{p} = 0, \text{ nd} = 0.$ $\frac{1}{p} = 0.$	i[3*i[2]+3] i[4*] nd = 1. NER_F90_RI	to i[4*i[2]+2 i[2]+3] i[0] EAL then ent C Fo i[0] i[1]	$\frac{I(3*I(3)+4) \text{ to}}{I(4*I(3)+4) \text{ to}}$	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4 *ndi If combine and ni = 2, na	$\frac{1}{p}$ prizes ims+4, na = 0, n er is MPI_COMBI $\frac{1}{p}$ r = 0, nd = 0. er is MPI_COMBI Construct	i[3*i[2]+3] i[4*i nd = 1. NER_F90_R etor argume	to $i[4*i[2]+2$ i[2]+3] l[0] EAL then mt C For $i[0]i[1]OMPLEX thenent C For$	I(3*I(3)+4) to I(4*I(3)-4) to I(4*I(4*I(3)-4) to I(4*I(4*I(4)-4)) to I(4*I(4*I(4*I(4)-4)) to I(4*I(4*I(4*I(4*I(4*I(4*I(4*I(4*I(4*I(4*	I(4*I(3)+3) (3)+4)
array_of order <u>oldtype</u> and ni = 4 *ndi If combine and ni = 2, na	$\frac{1}{p} = 0, \text{ nd} = 0.$ $\frac{1}{p} = 0, \text{ nd} = 0.$ $\frac{1}{p} = 0.$	i[3*i[2]+3] i[4*i nd = 1. NER_F90_R etor argume	to i[4*i[2]+2 i[2]+3] i[0] EAL then ent C Fo i[0] i[1]	I $I(3*I(3)+4)$ to I(4*I(3)-4) to I(4*I(3)-4) to I(1) I(1) I(2)	I(4*I(3)+4)
array_of order oldtype and ni = 4*ndi If combine and ni = 2, na If combine	$\frac{1}{p}$	i[3*i[2]+3] i[4*i nd = 1. NER_F90_R etor argume	to $i[4*i[2]+2$ i[2]+3] l[0] EAL then ent C Formula $i[0]i[1]DMPLEX thenent C Formulai[0]$	$\frac{I(3*I(3)+4) \text{ to}}{I(4*I(3)-4) \text{ to}}$	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4*ndi If combine and ni = 2, na If combine and ni = 2, na	$\frac{1}{p} = 0, \text{ nd} = 0.$	i[3*i[2]+3] i[4*i nd = 1. NER_F90_RI etor argume	to $i[4*i[2]+2$ i[2]+3] l[0] EAL then ent C Formation $i[0]$ i[1] DMPLEX then ent C Formation $i[0]$ i[1]	$\frac{I(3*I(3)+4) \text{ to}}{I(4*I(3)-4) \text{ to}}$	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4*ndi If combine and ni = 2, na If combine and ni = 2, na	$\frac{1}{p}$	i[3*i[2]+3] i[4*i nd = 1. NER_F90_RI etor argume	to $i[4*i[2]+2$ i[2]+3] l[0] EAL then ent C Formation $i[0]$ i[1] DMPLEX then ent C Formation $i[0]$ i[1]	$\frac{I(3*I(3)+4) \text{ to}}{I(4*I(3)-4) \text{ to}}$	I(4*I(3)+3) (3)+4)
array_of order oldtype and ni = 4*ndi If combine and ni = 2, na If combine and ni = 2, na	$\frac{1}{1} \text{psizes}$ $\frac{1}{1} \text{ms} + 4, \text{ na} = 0, \text{ n}$ $\frac{1}{1} \text{construct}}$ $\frac{1}{1} \text{Construct}$ $\frac{1}{$	i[3*i[2]+3] i[4*i nd = 1. NER_F90_RI etor argume	to $i[4*i[2]+2$ i[2]+3] l[0] EAL then ent C For $i[0]i[1]DMPLEX thenent C For i[0]i[1]ITEGER then$	$\frac{I(3*I(3)+4) \text{ to}}{I(4*I(3)-4) \text{ to}}$	I(4*I(3)+3) (3)+4)

	ni = 1, na = 0,		. 1		1 2
	lf combiner is N	IPI_COMBINER_RESIZED	then		3
		Constructor argument	С	Fortran location	4
		lb	a[0]	A(1)	5
		extent	a[0] $a[1]$	A(2)	6
		oldtype	d[0]	D(1)	7
			[°]	_ (-)	8
and 1	ni = 0, na = 2,	nd = 1.			9
					10
4.1.1	4 Examples				11
The	following exam	oles illustrate the use of d	lerived	datatypes	12
Inc	tonowing examp	nes mustrate the use of e		r dataty pes.	13
Exai	mple 4.13 Sen	d and receive a section o	f a 3D) array.	14
	_			-	15
		100,100), e(9,9,9)			16 17
		slice, twoslice, three			18
		ND=MPI_ADDRESS_KIND)	lb, s	sizeofreal	19
	INTEGER sta	tus(MPI_STATUS_SIZE)			20
a	++ +1		. 4 . 4	2.10)	21
C C		section a(1:17:2, 3:	11, 2	2:10)	22
C	and store i	t in e(:,:,:).			23
	CATT MPT CO	MM_RANK(MPI_COMM_WORI	D ma	vrank jerr)	24
	OALL IN I_00		ш, шу	(iank, ieii)	25
	CALL MPI TY	PE_GET_EXTENT(MPI_REA	AL. 11	o. sizeofreal. ierr)	26
	_		,		27
С	create data	type for a 1D sectior	ı		28
		PE_VECTOR(9, 1, 2, MF		AL, oneslice, ierr)	29
					30
С		type for a 2D sectior			31
	CALL MPI_TY	PE_CREATE_HVECTOR(9,	1, 10	00*sizeofreal, oneslice,	32 33
		two	oslice	e, ierr)	34
~					35
С		type for the entire s			36
	CALL MPI_TY			00*100*sizeofreal, twoslice,	37
		thi	reesti	ice, ierr)	38
	CATT MDT TV	PE_COMMIT(threeslice,	iorr	-)	39
				-/ slice, myrank, 0, e, 9*9*9,	40
), MPI_COMM_WORLD, status, ierr)	41
		·····, ···	, .	,,,,,	42
					43
Exai	mple 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
ſ
                                                                                     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
              i, dest, tag;
int
                                                                                     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
        zdisp[j] = i;
                                                                                      4
        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
              zzblock[2] = \{1,1\};
                                                                                        14
int
                                                                                        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
                                                                                        36
zztype[1] = Zparticles;
                                                                                        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
     int
              utype;
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
```

```
4.2
            Pack and Unpack
1
\mathbf{2}
      Some existing communication libraries provide pack/unpack functions for sending noncon-
3
      tiguous data. In these, the user explicitly packs data into a contiguous buffer before sending
4
      it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are
5
      described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking.
6
      The user specifies the layout of the data to be sent or received, and the communication
7
      library directly accesses a noncontiguous buffer. The pack/unpack routines are provided
8
      for compatibility with previous libraries. Also, they provide some functionality that is not
9
      otherwise available in MPI. For instance, a message can be received in several parts, where
10
      the receive operation done on a later part may depend on the content of a former part.
11
      Another use is that outgoing messages may be explicitly buffered in user supplied space,
12
13
      thus overriding the system buffering policy. Finally, the availability of pack and unpack
      operations facilitates the development of additional communication libraries layered on top
14
      of MPI.
15
16
17
      MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)
18
       IN
                 inbuf
                                              input buffer start (choice)
19
20
       IN
                 incount
                                               number of input data items (non-negative integer)
21
       IN
                 datatype
                                              datatype of each input data item (handle)
22
       OUT
                 outbuf
                                              output buffer start (choice)
23
^{24}
       IN
                 outsize
                                              output buffer size, in bytes (non-negative integer)
25
       INOUT
                 position
                                              current position in buffer, in bytes (integer)
26
       IN
                                              communicator for packed message (handle)
                 comm
27
28
29
      C binding
30
      int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,
^{31}
                     void *outbuf, int outsize, int *position, MPI_Comm comm)
32
     F08 binding
33
     MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
34
          TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
35
          INTEGER, INTENT(IN) :: incount, outsize
36
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
          TYPE(*), DIMENSION(...)
                                      :: outbuf
38
          INTEGER, INTENT(INOUT) :: position
39
          TYPE(MPI_Comm), INTENT(IN) :: comm
40
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
      F binding
43
     MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
44
          <type> INBUF(*), OUTBUF(*)
45
          INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
46
          Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer
47
      space specified by outbuf and outsize. The input buffer can be any communication buffer
```

allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize bytes, starting at the address outbuf (length is counted in *bytes*, not elements, as if it were a communication buffer for a message of type MPI_PACKED).

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

			10
MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)			11
IN	inbuf	input buffer start (choice)	12
IN	insize	size of input buffer, in bytes (non-negative integer)	13 14
INOUT	position	current position in bytes (integer)	15
OUT	outbuf	output buffer start (choice)	16
IN	outcount	number of items to be unpacked (non-negative integer)	17 18
IN	datatype	datatype of each output data item (handle)	18
IN	comm	communicator for packed message (handle)	20
			21
C binding	.		22
			23

int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

F08 binding

MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,	27
ierror)	28
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf	29
INTEGER, INTENT(IN) :: insize, outcount	30
INTEGER, INTENT(INOUT) :: position	31
TYPE(*), DIMENSION() :: outbuf	32
TYPE(MPI_Datatype), INTENT(IN) :: datatype	33
TYPE(MPI_Comm), INTENT(IN) :: comm	34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
E binding	36
F binding	37

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 42the buffer space specified by inbuf and insize. The output buffer can be any communication 43 buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize 44bytes, starting at address inbuf. The input value of position is the first location in the input 45buffer occupied by the packed message. position is incremented by the size of the packed 46message, so that the output value of **position** is the first location in the input buffer after 47

1

 $\mathbf{2}$

3

4

5

6 $\overline{7}$

8 9

 24

25

26

38

39

40

41

the locations occupied by the message that was unpacked. **comm** is the communicator used to receive the packed message.

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

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

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

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

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

³⁵ MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the ³⁶ value of position that was output by the previous call, and the same values for inbuf, insize ³⁷ 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.)

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10

11

12

13

32

33

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.

			derer ist		· · · · · ·
MPI_PACK_	SIZE()	ncount.	datatype.	comm.	size)
		,			

IN	incount	count argument to packing call (non-negative integer)	6
IN	datatype	datatype argument to packing call (handle)	7
IN	comm	communicator argument to packing call (handle)	9
OUT	size	upper bound on size of packed message, in bytes (non-	10
		negative integer)	11

C binding

F08 binding

MPI_Pack_size(incount, datatype, comm, s	size, ierror)
INTEGER, INTENT(IN) :: incount	
TYPE(MPI_Datatype), INTENT(IN) :: c	latatype
TYPE(MPI_Comm), INTENT(IN) :: comm	
INTEGER, INTENT(OUT) :: size	
INTEGER, OPTIONAL, INTENT(OUT) :: i	lerror

F binding

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm). If the packed size of the datatype cannot be expressed by the size parameter, then MPI_PACK_SIZE sets the value of size to MPI_UNDEFINED.

Rationale. The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). (*End of rationale.*)

Example 4.21 An example using MPI_PACK.

```
int position, i, j, a[2];
char buff[1000];
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
    /* SENDER CODE */
    position = 0;
    MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
```

1

 $\mathbf{2}$

3 4 5

12 13

14

15

16

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

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

```
/* gather at root packed messages */ 45
MPI_Gatherv(lbuf, position, MPI_PACKED, NULL, 46
NULL, NULL, MPI_DATATYPE_NULL, root, comm); 47
48
```

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

4.3 Canonical MPI_PACK and MPI_UNPACK

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

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

The buffer will contain exactly the packed data, without headers. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.

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

29

30 31

32

33

34

35 36

37

38

39

40 41

42

43

44

MPI	_PACK_	_EXTERNAL(datarep, inbuf, in	count, 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 (non-negative integer)	5
IN		datatype	datatype of each input data item (handle)	6 7
Ol	JT	outbuf	output buffer start (choice)	8
IN		outsize	output buffer size, in bytes (integer)	9
IN	OUT	position	current position in buffer, in bytes (integer)	10 11
				11

C binding

int MPI_Pack_external(const char *datarep, const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, MPI_Aint outsize, MPI_Aint *position)

F08 binding

MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize, position, ierror) CHARACTER(LEN=*), INTENT(IN) :: datarep TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf INTEGER, INTENT(IN) :: incount TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(*), DIMENSION(..) :: outbuf INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, IERROR) CHARACTER*(*) DATAREP <type> INBUF(*), OUTBUF(*) INTEGER INCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION

 31

```
1
     MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outsize, datatype)
\mathbf{2}
       IN
                 datarep
                                             data representation (string)
3
       IN
                 inbuf
                                             input buffer start (choice)
4
5
       IN
                 insize
                                             input buffer size, in bytes (integer)
6
       INOUT
                 position
                                             current position in buffer, in bytes (integer)
7
       OUT
                 outbuf
                                             output buffer start (choice)
8
9
       IN
                 outsize
                                             number of output data items (non-negative integer)
10
       IN
                 datatype
                                             datatype of output data item (handle)
11
12
     C binding
13
     int MPI_Unpack_external(const char *datarep, const void *inbuf,
14
                    MPI_Aint insize, MPI_Aint *position, void *outbuf,
15
                     int outsize, MPI_Datatype datatype)
16
17
     F08 binding
18
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outsize,
19
                    datatype, ierror)
20
          CHARACTER(LEN=*), INTENT(IN) :: datarep
21
          TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
22
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                                insize
23
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
24
          TYPE(*), DIMENSION(..) :: outbuf
25
          INTEGER, INTENT(IN) :: outsize
26
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     F binding
29
     MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTSIZE,
30
                    DATATYPE, IERROR)
^{31}
          CHARACTER*(*) DATAREP
32
          <type> INBUF(*), OUTBUF(*)
33
          INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
34
          INTEGER OUTSIZE, DATATYPE, IERROR
35
36
37
     MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)
38
39
       IN
                 datarep
                                             data representation (string)
40
       IN
                 incount
                                             number of input data items (non-negative integer)
41
       IN
42
                 datatype
                                             datatype of each input data item (handle)
43
       OUT
                 size
                                             output buffer size, in bytes (integer)
44
45
     C binding
46
     int MPI_Pack_external_size(const char *datarep, int incount,
47
                    MPI_Datatype datatype, MPI_Aint *size)
48
```

F08 bind	ling		1
		, incount, datatype, size, ierror)	2
	ACTER(LEN=*), INTENT(I	_	3 4
	GER, INTENT(IN) :: in (MPI_Datatype), INTENT	count (IN) ·· datatupe	5
		IND), INTENT(OUT) :: size	6
	GER, OPTIONAL, INTENT(7
F bindin			8
	•	, INCOUNT, DATATYPE, SIZE, IERROR)	9
	ACTER*(*) DATAREP	, 1000001, 5000011, 5022, 1200000	10 11
INTE	GER INCOUNT, DATATYPE,	IERROR	11
INTE	GER(KIND=MPI_ADDRESS_K	IND) SIZE	13
			14
			15
MPI_PACI	K_EXTERNAL_SIZE(datare	ep, incount, datatype, size)	16
IN	datarep	data representation (string)	17
IN	incount	number of input data items (integer)	18 19
IN	datatype	datatype of each input data item (handle)	20
OUT	size	output buffer size, in bytes (integer)	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
			47
			10

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).

3

4

• MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN: Scan across all members of a group (also called prefix) (Section 5.11, Section 5.11.2, Section 5.12.11, and Section 5.12.12).

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

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

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

Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. The collective operations do not have a message tag argument. A more detailed discussion of correct use of collective routines is found in Section 5.14.

35 36

37

38

39

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

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

(End of rationale.)

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

Unofficial Draft for Comment Only

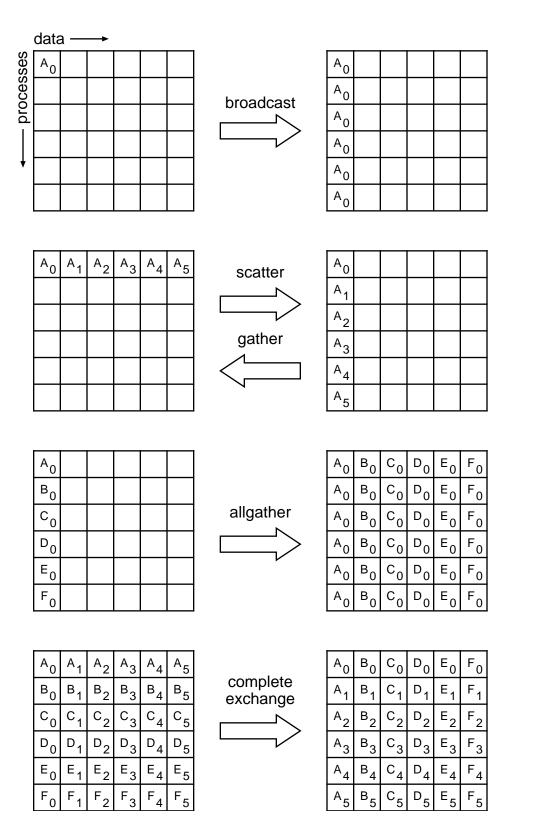


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

Unofficial Draft for Comment Only

 $\mathbf{2}$

 31

 41

 $44 \\ 45$

On the other hand, a correct, portable program must allow for the fact that a collective call *may* be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it. These issues are discussed further in Section 5.14. (*End of advice to users.*)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden, special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 5.14. (End of advice to implementors.)

¹³ Many of the descriptions of the collective routines provide illustrations in terms of ¹⁴ blocking MPI point-to-point routines. These are intended solely to indicate what data is ¹⁵ sent or received by what process. Many of these examples are *not* correct MPI programs; ¹⁶ 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.

⁴⁷ Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its ⁴⁸ 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	
To understand how collective operations apply to intercommunicators, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [55]):	3 4
All-To-All All processes contribute to the result. All processes receive the result.	
 MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV 9 	8
 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	.3
• 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	9
One-To-All One process contributes to the result. All processes receive the result.	
 MPI_BCAST, MPI_IBCAST MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV 22 	4
Other Collective operations that do not fit into one of the above categories.	
• MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN	
The data movement patterns of MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, and MPI_IEXSCAN do not fit this taxonomy. The application of collective communication to intercommunicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 5.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 5.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior. The following collective operations also apply to intercommunicators: • MPI_BARRIER, MPI_IBARRIER	30 31 32 33 34 35 36 37 38 39 10 11 12 12
44	
• MPI_BCAST, MPI_IBCAST	
• MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV,	
• MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV, 48	8

Unofficial Draft for Comment Only

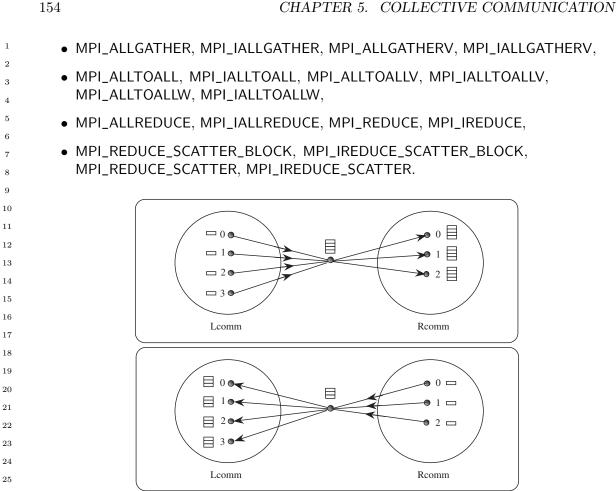


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

2930 31

2627

28

5.2.3 Specifics for Intercommunicator Collective Operations

32 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

Unofficial Draft for Comment Only

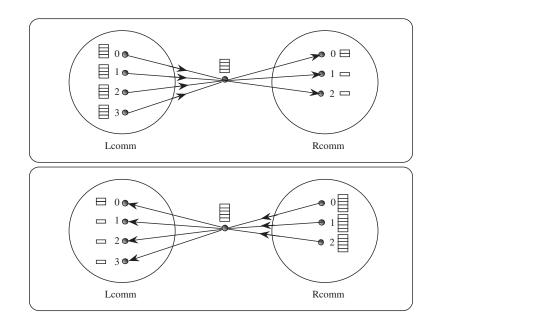


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

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 Synchronization

```
MPI_BARRIER(comm)
```

IN comm

C binding

int MPI_Barrier(MPI_Comm comm)

```
F08 binding
MPI_Barrier(comm, ierror)
```

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

F binding

```
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.

⁴⁶ 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 ⁴⁸ ⁴⁷ ⁴⁸

Unofficial Draft for Comment Only

```
\mathbf{2}
     the call before all processes in its own group have entered the call.
3
4
     5.4
            Broadcast
5
6
7
8
     MPI_BCAST(buffer, count, datatype, root, comm)
9
       INOUT
                 buffer
                                              starting address of buffer (choice)
10
       IN
                                              number of entries in buffer (non-negative integer)
                 count
11
12
       IN
                 datatype
                                              data type of buffer (handle)
13
       IN
                 root
                                              rank of broadcast root (integer)
14
       IN
                 comm
                                              communicator (handle)
15
16
17
     C binding
18
     int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
19
                     MPI_Comm comm)
20
     F08 binding
21
     MPI_Bcast(buffer, count, datatype, root, comm, ierror)
22
          TYPE(*), DIMENSION(..) :: buffer
23
          INTEGER, INTENT(IN) :: count, root
24
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                   datatype
25
          TYPE(MPI_Comm), INTENT(IN) :: comm
26
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
27
     F binding
28
     MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
29
30
          <type> BUFFER(*)
^{31}
          INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
32
          If comm is an intracommunicator, MPI_BCAST broadcasts a message from the process
33
     with rank root to all processes of the group, itself included. It is called by all members of
34
     the group using the same arguments for comm and root. On return, the content of root's
35
     buffer is copied to all other processes.
36
```

General, derived datatypes are allowed for datatype. The type signature of count, 37 datatype on any process must be equal to the type signature of count, datatype at the root. 38 This implies that the amount of data sent must be equal to the amount received, pairwise 39 between each process and the root. MPI_BCAST and all other data-movement collective 40 routines make this restriction. Distinct type maps between sender and receiver are still 41 allowed. 42

The "in place" option is not meaningful here.

43 If comm is an intercommunicator, then the call involves all processes in the intercom-44 municator, but with one group (group A) defining the root process. All processes in the 45 other group (group B) pass the same value in argument root, which is the rank of the root 46 in group A. The root passes the value MPI_ROOT in root. All other processes in group A 47pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes 48

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

		integer, significant only at root)	00
IN	recvtype	data type of recv buffer elements (significant only at	34
		root) (handle)	35
		100t) (hahdie)	36
IN	root	rank of receiving process (integer)	37
IN	comm	communicator (handle)	38
			39

C binding

F08 binding

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf

Unofficial Draft for Comment Only

40

1	
2	TYPE(*), DIMENSION() :: recvbuf
3	INTEGER, INTENT(IN) :: sendcount, recvcount, root
4	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
	TYPE(MPI_Comm), INTENT(IN) :: comm
5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6 7	F binding
8	MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
9	ROOT, COMM, IERROR)
	<type> SENDBUF(*), RECVBUF(*)</type>
10	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
11	
12	If comm is an intracommunicator, each process (root process included) sends the con-
13	tents of its send buffer to the root process. The root process receives the messages and stores
14	them in rank order. The outcome is <i>as if</i> each of the n processes in the group (including
15	the root process) had executed a call to
16	
17	MPI_Send(sendbuf, sendcount, sendtype, root ,),
18 19	and the root had executed n calls to
20	
20 21	MPI_Recv(recvbuf+i· recvcount· extent(recvtype), recvcount, recvtype, i,),
21	
22	where extent(recvtype) is the type extent obtained from a call to MPI_Type_get_extent.
23	An alternative description is that the n messages sent by the processes in the group
24	are concatenated in rank order, and the resulting message is received by the root as if by a
26	call to MPI_RECV (recvbuf, recvcount·n, recvtype,).
20	The receive buffer is ignored for all non-root processes.
28	General, derived datatypes are allowed for both sendtype and recvtype. The type signa-
29	ture of sendcount, sendtype on each process must be equal to the type signature of recvcount,
30	recvtype at the root. This implies that the amount of data sent must be equal to the amount
31	of data received, pairwise between each process and the root. Distinct type maps between
32	sender and receiver are still allowed.
33	All arguments to the function are significant on process root, while on other processes,
34	only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments
	root and comm must have identical values on all processes.
35	The specification of counts and types should not cause any location on the root to be
36 27	written more than once. Such a call is erroneous.
37	Note that the recvcount argument at the root indicates the number of items it receives
38	from <i>each</i> process, not the total number of items it receives.
39 40	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
41	the contribution of the root to the gathered vector is assumed to be already in the correct
42 43	place in the receive buffer.
	If comm is an intercommunicator, then the call involves all processes in the intercom-
44	municator, but with one group (group A) defining the root process. All processes in the
45	other group (group B) pass the same value in argument root, which is the rank of the root
46	in group A. The root passes the value MPI_ROOT in root. All other processes in group A
47	pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to
48	

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

			4
MPI_GATHE	ERV(sendbuf, sendcount, sen comm)	dtype, recvbuf, recvcounts, displs, recvtype, root,	5 6
IN	sendbuf	starting address of send buffer (choice)	7
IN	sendcount	number of elements in send buffer (non-negative inte-	8 9
		ger)	9 10
IN	sendtype	data type of send buffer elements (handle)	11
OUT	recvbuf	address of receive buffer (choice, significant only at	12
		root)	13 14
IN	recvcounts	non-negative integer array (of length group size) con-	14 15
		taining the number of elements that are received from each process (significant only at root)	16
		,	17
IN	displs	integer array (of length group size). Entry i specifies	18
		the displacement relative to recvbuf at which to place the incoming data from process i (significant only at	19
		root)	20 21
IN	recvtype	data type of recv buffer elements (significant only at	22
	leeveype	root) (handle)	23
IN	root	rank of receiving process (integer)	24
IN			25
IIN	comm	communicator (handle)	26

C binding

```
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)
```

F08 binding

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                34
             recvtype, root, comm, ierror)
                                                                                35
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                36
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                37
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
                                                                                38
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                39
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                41
                                                                                42
F binding
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                                                                                43
             RECVTYPE, ROOT, COMM, IERROR)
                                                                                44
```

1 $\mathbf{2}$

3

27

28

29

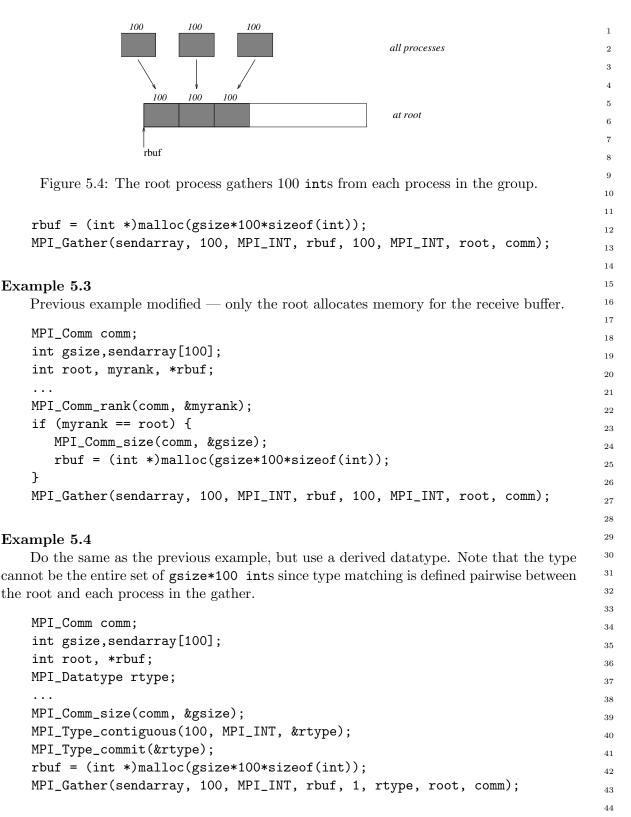
30

 31 32

33

40

1 MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count $\mathbf{2}$ of data from each process, since recvcounts is now an array. It also allows more flexibility 3 as to where the data is placed on the root, by providing the new argument, displs. 4 If comm is an intracommunicator, the outcome is as if each process, including the root $\mathbf{5}$ process, sends a message to the root, 6 MPI_Send(sendbuf, sendcount, sendtype, root, ...), 7 8 and the root executes **n** receives. 9 10 MPI_Recv(recvbuf+displs[i]· extent(recvtype), recvcounts[i], recvtype, i, ...). 11 12The data received from process j is placed into recvbuf of the root process beginning at 13 offset displs[j] elements (in terms of the recvtype). 14The receive buffer is ignored for all non-root processes. 15The type signature implied by sendcount, sendtype on process i must be equal to the 16type signature implied by recvcounts[i], recvtype at the root. This implies that the amount 17of data sent must be equal to the amount of data received, pairwise between each process 18 and the root. Distinct type maps between sender and receiver are still allowed, as illustrated 19in Example 5.6. 20All arguments to the function are significant on process root, while on other processes, 21only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments 22 root and comm must have identical values on all processes. 23The specification of counts, types, and displacements should not cause any location on 24 the root to be written more than once. Such a call is erroneous. 25The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as 26the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and 27the contribution of the root to the gathered vector is assumed to be already in the correct 28place in the receive buffer. 29 If comm is an intercommunicator, then the call involves all processes in the intercom-30 municator, but with one group (group A) defining the root process. All processes in the 31 other group (group B) pass the same value in argument root, which is the rank of the root 32 in group A. The root passes the value MPI_ROOT in root. All other processes in group A 33 pass the value MPI_PROC_NULL in root. Data is gathered from all processes in group B to 34 the root. The send buffer arguments of the processes in group B must be consistent with 35 the receive buffer argument of the root. 36 37 Examples using MPI_GATHER, MPI_GATHERV 5.5.138 39 The examples in this section use intracommunicators. 40Example 5.2 41 Gather 100 ints from every process in group to root. See Figure 5.4. 4243 MPI_Comm comm; 44 int gsize,sendarray[100]; 45int root, *rbuf; 46 47 . . . MPI_Comm_size(comm, &gsize);



Example 5.5

Now have each process send 100 ints to root, but place each set (of 100) stride ints ⁴⁶ apart at receiving end. Use MPI_GATHERV and the displs argument to achieve this effect. ⁴⁷ Assume stride \geq 100. See Figure 5.5.⁴⁸

Unofficial Draft for Comment Only

```
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>
```

Unofficial Draft for Comment Only

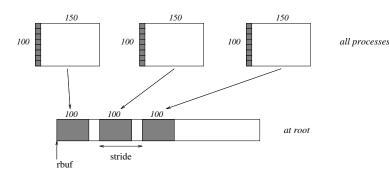


Figure 5.6: The root process gathers column 0 of a 100×150 C array, and each set is placed stride ints apart.

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

Example 5.7

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

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

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3 4 5

6

7

9 10

11

12 13

14

15

16

17

18

19

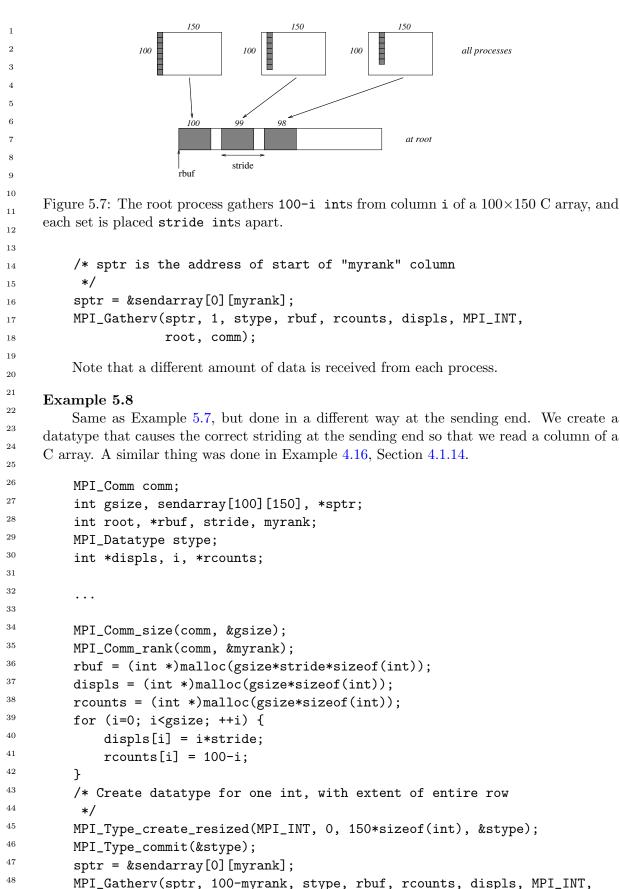
20

21

22 23 24

25

26



```
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

1

2 3

4

5

6 7

46

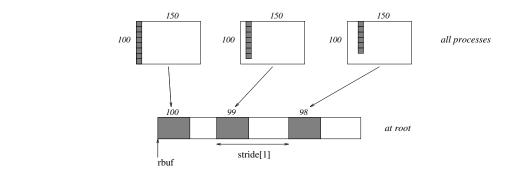


Figure 5.8: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed stride[i] ints apart (a varying stride).

18

1

2

7

8

9 10

11

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

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

Unofficial Draft for Comment Only

```
1
    MPI_Type_commit(&stype);
                                                                                           2
    sptr = &sendarray[0][myrank];
                                                                                           3
    MPI_Gatherv(sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
                  root, comm);
                                                                                           4
                                                                                           5
                                                                                           6
5.6
      Scatter
                                                                                           7
                                                                                           9
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
                                                                                           10
                                                                                           11
            sendbuf
                                        address of send buffer (choice, significant only at root)
 IN
                                                                                           12
 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
                                                                                          21
 IN
                                        data type of receive buffer elements (handle)
            recvtype
                                                                                          22
 IN
            root
                                       rank of sending process (integer)
                                                                                          23
 IN
                                        communicator (handle)
            comm
                                                                                           24
                                                                                           25
C binding
                                                                                           26
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                                                                                          27
               void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                                                                                           28
               MPI_Comm comm)
                                                                                          29
                                                                                           30
F08 binding
                                                                                           31
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                           32
               root, comm, ierror)
                                                                                           33
    TYPE(*), DIMENSION(...), INTENT(IN) ::
                                                 sendbuf
                                                                                          34
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                          35
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                          36
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                          37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                           38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                           39
F binding
                                                                                           40
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
                                                                                          41
                                                                                           42
               ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
                                                                                           43
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
                                                                                           44
                                                                                           45
    MPI_SCATTER is the inverse operation to MPI_GATHER.
                                                                                           46
    If comm is an intracommunicator, the outcome is as if the root executed n send oper-
                                                                                           47
ations.
                                                                                           48
```

MPI_Send(sendbuf+i· sendcount· extent(sendtype), sendcount, sendtype, i,...),

- and each process executed a receive,
 - MPI_Recv(recvbuf, recvcount, recvtype, i,...).

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

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.

¹⁹ The specification of counts and types should not cause any location on the root to be ²⁰ read more than once.

21 22 23

> 24 25

26

27

28

29

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.

³⁰ If comm is an intercommunicator, then the call involves all processes in the intercom-³¹ municator, but with one group (group A) defining the root process. All processes in the ³² other group (group B) pass the same value in argument root, which is the rank of the root ³³ in group A. The root passes the value MPI_ROOT in root. All other processes in group A ³⁴ pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in ³⁵ group B. The receive buffer arguments of the processes in group B must be consistent with ³⁶ the send buffer argument of the root.

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

1

2

3 4

5 6

7

8

9

10

16

17

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)			
IN	sendbuf	address of send buffer (choice, significant only at root)	$\frac{3}{4}$
IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	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)	11
IN	recvcount	number of elements in receive buffer (non-negative in-teger)	13 14
IN	recvtype	data type of receive buffer elements (handle)	15 16
IN	root	rank of sending process (integer)	17
IN	comm	communicator (handle)	18
			19
C binding		-	20 21
int MPI_Sc		ouf, const int sendcounts[],	21
<pre>const int displs[], MPI_Datatype sendtype, void* recvbuf,</pre>			23
	int recvcount, MP1_D	atatype recvtype, int root, MPI_Comm comm)	24
F08 binding			

MPI_SCATTERV is the inverse operation to MPI_GATHERV.

MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying count of data to be sent to each process, since sendcounts is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing an additional argument, displs.

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

 31

1 MPI_Send(sendbuf+displs[i] extent(sendtype), sendcounts[i], sendtype, i,...), 2 3 and each process executed a receive, 4 MPI_Recv(recvbuf, recvcount, recvtype, i,...). 56 The send buffer is ignored for all non-root processes. 7 The type signature implied by sendcount[i], sendtype at the root must be equal to the 8 type signature implied by recvcount, recvtype at process i (however, the type maps may be 9 different). This implies that the amount of data sent must be equal to the amount of data 10 received, pairwise between each process and the root. Distinct type maps between sender 11 and receiver are still allowed. 12All arguments to the function are significant on process root, while on other processes, 13 only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments 14 root and comm must have identical values on all processes. 15The specification of counts, types, and displacements should not cause any location on 16the root to be read more than once. 17The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as 18 the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and 19 root "sends" no data to itself. The scattered vector is still assumed to contain n segments, 20where n is the group size; the *root*-th segment, which root should "send to itself," is not 21moved. 22 If comm is an intercommunicator, then the call involves all processes in the intercom-23municator, but with one group (group A) defining the root process. All processes in the 24other group (group B) pass the same value in argument root, which is the rank of the root 25in group A. The root passes the value MPI_ROOT in root. All other processes in group A 26pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in 27group B. The receive buffer arguments of the processes in group B must be consistent with 28the send buffer argument of the root. 29 30 5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV 31 32 The examples in this section use intracommunicators. 33 34Example 5.11 35 The reverse of Example 5.2. Scatter sets of 100 ints from the root to each process in 36 the group. See Figure 5.9. 37 38 MPI_Comm comm; int gsize,*sendbuf; 39 int root, rbuf[100]; 40 41 . . . 42MPI_Comm_size(comm, &gsize); sendbuf = (int *)malloc(gsize*100*sizeof(int)); 43 44. . . MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm); 4546 47 Example 5.12 48

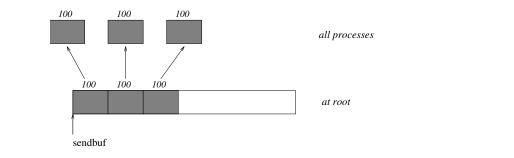


Figure 5.9: The root process scatters sets of 100 ints to each process in the group.

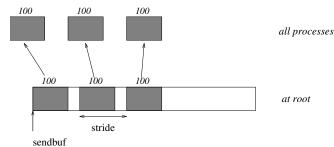


Figure 5.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

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

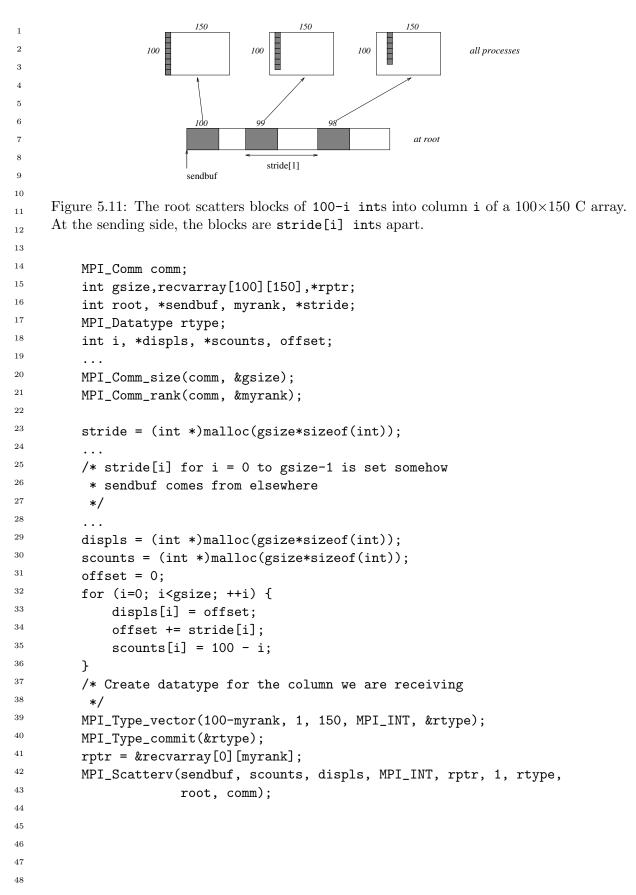
```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;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

Example 5.13

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

Unofficial Draft for Comment Only

 $45 \\ 46$



5.7 Gather-to-all

			2
			3
MPI ALI	_GATHER(sendbuf, sendo	count, sendtype, recvbuf, recvcount, recvtype, comm)	4
IN –	sendbuf	starting address of send buffer (choice)	5 6
			7
IN	sendcount	number of elements in send buffer (non-negative inte- ger)	8
IN	sendtype	data type of send buffer elements (handle)	9
OUT	recvbuf	address of receive buffer (choice)	10 11
			12
IN	recvcount	number of elements received from any process (non-negative integer)	13
IN	recvtype	data type of receive buffer elements (handle)	14 15
IN	comm	communicator (handle)	16
			17
C bindi	ng		18
int MPI	-	d* sendbuf, int sendcount,	19
	01	sendtype, void* recvbuf, int recvcount,	20 21
	MPI_Datatype r	recvtype, MPI_Comm comm)	21
F08 bin	ding		22
MPI_All	gather(sendbuf, send	count, sendtype, recvbuf, recvcount, recvtype,	24
	comm, ierror)		25
TYP	E(*), DIMENSION(),	INTENT(IN) :: sendbuf	26
	E(*), DIMENSION()		27
		sendcount, recvcount	28
	• 1	ENT(IN) :: sendtype, recvtype	29
	E(MPI_Comm), INTENT(30
TN.I.	EGER, OPTIONAL, INTE	NT(UUT) :: ierror	31
F bindi	ng		32
MPI_ALL	GATHER(SENDBUF, SEND	COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	33
	COMM, IERROR)		34
<ty< td=""><td><pre>pe> SENDBUF(*), RECV</pre></td><td>BUF(*)</td><td>35</td></ty<>	<pre>pe> SENDBUF(*), RECV</pre>	BUF(*)	35
INT	EGER SENDCOUNT, SEND	TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	36
MPI	_ALLGATHER can be th	nought of as MPI_GATHER, but where all processes receive	37 38
		ot. The block of data sent from the j-th process is received	39
		he j-th block of the buffer recvbuf.	40
		ed with sendcount, sendtype, at a process must be equal to	41
		th recvcount, recvtype at any other process.	42
If co	omm is an intracommuni	cator, the outcome of a call to MPI_ALLGATHER() is as	43
if all pro	cesses executed \boldsymbol{n} calls t	0	44
мпт	Cothom (conduct cond-	ount conditions require a contract	45
MPT_	Gauner (Senabul, Senac	ount,sendtype,recvbuf,recvcount,	

recvtype,root,comm)

1			correct usage of $MPI_ALLGATHER$ are easily found	
2	from the corresponding rules for MPI_GATHER.			
3	The "in place" option for intracommunicators is specified by passing the value			
4	MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored.			
5			ssumed to be in the area where that process would	
6 7		own contribution to the recei		
8			nen each process of one group (group A) contributes neatenated and the result is stored at each process	
9		,	sely the concatenation of the contributions of the	
10		0 1 (0 1)	process in group A. The send buffer arguments in	
11	-	0	ceive buffer arguments in group B, and vice versa.	
12	0 - 1			
13	Advice	te to users. The communication	ation pattern of $MPI_ALLGATHER$ executed on an	
14			not be symmetric. The number of items sent by	
15	-	、 _	by the arguments sendcount, sendtype in group A	
16		0	vtype in group B), need not equal the number of	
17			(as specified by the arguments sendcount, sendtype	
18 19			vcount, recvtype in group A). In particular, one can y specifying sendcount $= 0$ for the communication	
20		e reverse direction. (End of a		
21				
22				
23				
24	MPI_ALLG	ATHERV(senabut, senacount,	sendtype, recvbuf, recvcounts, displs, recvtype, comm)	
25	IN	sendbuf	starting address of sond buffer (choice)	
26 27			starting address of send buffer (choice)	
28	IN	sendcount	number of elements in send buffer (non-negative integer)	
29 30	IN	sendtype	data type of send buffer elements (handle)	
31	OUT	recvbuf	address of receive buffer (choice)	
32	IN	recvcounts	non-negative integer array (of length group size) con-	
33			taining the number of elements that are received from	
34			each process	
35	IN	displs	integer array (of length group size). Entry i specifies	
36 37			the displacement (relative to $recvbuf)$ at which to place	
38			the incoming data from process i	
39	IN	recvtype	data type of receive buffer elements (handle)	
40	IN	comm	communicator (handle)	
41				
42	C binding			
43 44	int MPI_A	llgatherv(const void* se		
45			<pre>we, void* recvbuf, const int recvcounts[],</pre>	
46		<pre>const int displs[],</pre>	MPI_Datatype recvtype, MPI_Comm comm)	
47	F08 bindi	ng		
48				

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

```
1
          MPI_Comm comm;
\mathbf{2}
          int gsize, sendarray[100];
3
          int *rbuf;
4
          . . .
5
          MPI_Comm_size(comm, &gsize);
6
          rbuf = (int *)malloc(gsize*100*sizeof(int));
          MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
7
8
          After the call, every process has the group-wide concatenation of the sets of data.
9
10
11
           All-to-All Scatter/Gather
     5.8
12
13
14
     MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
15
16
       IN
                  sendbuf
                                             starting address of send buffer (choice)
17
       IN
                  sendcount
                                             number of elements sent to each process (non-negative
18
                                             integer)
19
       IN
                  sendtype
20
                                             data type of send buffer elements (handle)
21
       OUT
                  recvbuf
                                             address of receive buffer (choice)
22
       IN
                  recvcount
                                             number of elements received from any process (non-
23
                                             negative integer)
^{24}
25
       IN
                                             data type of receive buffer elements (handle)
                  recvtype
26
       IN
                                             communicator (handle)
                  comm
27
28
     C binding
29
     int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
30
                    void* recvbuf, int recvcount, MPI_Datatype recvtype,
^{31}
                    MPI_Comm comm)
32
33
     F08 binding
34
     MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
35
                    comm, ierror)
36
          TYPE(*), DIMENSION(..), INTENT(IN) ::
                                                      sendbuf
37
          TYPE(*), DIMENSION(..) :: recvbuf
          INTEGER, INTENT(IN) :: sendcount, recvcount
38
39
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40
          TYPE(MPI_Comm), INTENT(IN) :: comm
41
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     F binding
43
     MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
44
                    COMM, IERROR)
45
          <type> SENDBUF(*), RECVBUF(*)
46
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
47
48
```

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

If comm is an intracommunicator, the outcome is as if each process executed a send to each process (itself included) with a call to,

MPI_Send(sendbuf+i· sendcount· extent(sendtype),sendcount,sendtype,i, ...),

and a receive from every other process with a call to,

MPI_Recv(recvbuf+i· recvcount· 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.*)

 $44 \\ 45$

 $\mathbf{5}$

 24

 31

1 2	MPI_ALLT(DALLV(sendbuf, sendcounts, s recvtype, comm)	displs, sendtype, recvbuf, recvcounts, rdispls,	
3 4	IN	sendbuf	starting address of send buffer (choice)	
5 6	IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	
7 8 9	IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	
10 11	IN	sendtype	data type of send buffer elements (handle)	
12	OUT	recvbuf	address of receive buffer (choice)	
13 14 15	IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each rank	
16 17 18 19	IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	
20	IN	recvtype	data type of receive buffer elements (handle)	
21	IN	comm	communicator (handle)	
25 26 27 28	<pre>int MPI_Alltoallv(const void* sendbuf, const int sendcounts[],</pre>			
29 30 31 32 33 34 35 36 37 38	TYPE(; TYPE(; INTEG] rdisp: TYPE(] TYPE(]	allv(sendbuf, sendcounts rdispls, recvtype, c *), DIMENSION(), INTEN *), DIMENSION() :: re ER, INTENT(IN) :: sendco	<pre>I(IN) :: sendbuf ecvbuf ounts(*), sdispls(*), recvcounts(*),) :: sendtype, recvtype comm</pre>	
39 40 41 42 43 44 45	<pre>F binding MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,</pre>			
46 47 48	MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls			

⁴⁸ side is specified by rdispls.

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], 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,

 $\mathsf{MPI}_\mathsf{Send}(\mathsf{sendbuf}+\mathsf{sdispls}[i] \cdot \mathsf{extent}(\mathsf{sendtype}), \mathsf{sendcounts}[i], \mathsf{sendtype}, \mathsf{i}, \ldots),$

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.*)

 $\mathbf{2}$

 $\mathbf{5}$

 24

1 2	MPI_ALL	TOALLW(sendbuf, sendcounts recvtypes, comm)	s, 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 data types (of length group size). Entry j specifies 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	C bindin int MPI_	Alltoallw(const void* se const int sdispls[void* recvbuf, con	<pre>endbuf, const int sendcounts[],], const MPI_Datatype sendtypes[], st int recvcounts[], const int rdispls[], recvtypes[], MPI_Comm comm)</pre>
34	F08 bind	ling	
35 36 37 38	MPI_Allt TYPE	oallw(sendbuf, sendcount rdispls, recvtypes (*), DIMENSION(), INTH	ENT(IN) :: sendbuf
39		(*), DIMENSION() ::	
40 41	1 N I E	GER, INTENT(IN) :: send rdispls(*)	<pre>dcounts(*), sdispls(*), recvcounts(*),</pre>
42		(MPI_Datatype), INTENT(V-1
43		(MPI_Datatype), INTENT(]	V-1
44		(MPI_Comm), INTENT(IN) = GER, OPTIONAL, INTENT(OU	
45			
46 47	F bindin	0	IS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
48	TH I_ALLI	RDISPLS, RECVTYPES	

<type> SENDBUF(*), RECVBUF(*)</type>	1
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),</pre>	2
RDISPLS(*), RECVTYPES(*), COMM, IERROR	3
MPI_ALLTOALLW is the most general form of complete exchange. Like	4 5
MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW al-	6
lows separate specification of count, displacement and datatype. In addition, to allow max-	7
imum flexibility, the displacement of blocks within the send and receive buffers is specified	8
in bytes.	9
If comm is an intracommunicator, then the j-th block sent from process i is received by	10
process j and is placed in the i-th block of recvbuf. These blocks need not all have the same	11
size. The type signature associated with sendcounts[j], sendtypes[j] at process i must be equal	12
to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that	13
the amount of data sent must be equal to the amount of data received, pairwise between	14
every pair of processes. Distinct type maps between sender and receiver are still allowed.	15 16
The outcome is as if each process sent a message to every other process with	10
	18
MPI_Send(sendbuf+sdispls[i],sendcounts[i],sendtypes[i] ,i,),	19
and received a message from every other process with a call to	20
and received a message from every other process with a can to	21
MPI_Recv(recvbuf+rdispls[i],recvcounts[i],recvtypes[i] ,i,).	22
	23
All arguments on all processes are significant. The argument comm must describe the	24
same communicator on all processes.	25 26
Like for MPI_ALLTOALLV, the "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at <i>all</i> processes. In such a case, sendcounts,	20
sdispls and sendtypes are ignored. The data to be sent is taken from the recvbuf and replaced	28
by the received data. Data sent and received must have the same type map as specified	29
by the recvcounts and recvtypes arrays, and is taken from the locations of the receive buffer	30
specified by rdispls.	31
If comm is an intercommunicator, then the outcome is as if each process in group A	32
sends a message to each process in group B, and vice versa. The j-th send buffer of process	33
i in group A should be consistent with the i-th receive buffer of process j in group B, and	34
vice versa.	35
Rationale. The MPI_ALLTOALLW function generalizes several MPI functions by	36 37
carefully selecting the input arguments. For example, by making all but one process	38
have sendcounts[i] = 0, this achieves an MPI_SCATTERW function. (<i>End of rationale.</i>)	39
	40
5.9 Global Reduction Operations	41
5.9 Global Reduction Operations	42
The functions in this section perform a global reduce operation (for example sum, maximum,	43
and logical and) across all members of a group. The reduction operation can be either one of	44
a predefined list of operations, or a user-defined operation. The global reduction functions	45
come in several flavors: a reduce that returns the result of the reduction to one member of a	46
group, an all-reduce that returns this result to all members of a group, and two scan (parallel	47

```
1
     prefix) operations. In addition, a reduce-scatter operation combines the functionality of a
\mathbf{2}
     reduce and of a scatter operation.
3
4
     5.9.1
             Reduce
5
6
\overline{7}
      MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
8
       IN
                  sendbuf
                                               address of send buffer (choice)
9
10
       OUT
                  recvbuf
                                               address of receive buffer (choice, significant only at
11
                                               root)
12
       IN
                  count
                                               number of elements in send buffer (non-negative inte-
13
                                               ger)
14
                                               data type of elements of send buffer (handle)
       IN
                  datatype
15
16
       IN
                                               reduce operation (handle)
                  ор
17
       IN
                                               rank of root process (integer)
                  root
18
       IN
                                               communicator (handle)
                  comm
19
20
21
      C binding
      int MPI_Reduce(const void* sendbuf, void* recvbuf, int count,
22
                     MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
23
^{24}
      F08 binding
25
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
26
          TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
27
          TYPE(*), DIMENSION(..) :: recvbuf
28
          INTEGER, INTENT(IN) :: count, root
29
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
          TYPE(MPI_Op), INTENT(IN) :: op
31
          TYPE(MPI_Comm), INTENT(IN) :: comm
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
      F binding
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
35
36
          <type> SENDBUF(*), RECVBUF(*)
37
          INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
38
          If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the
39
      input buffer of each process in the group, using the operation op, and returns the combined
40
      value in the output buffer of the process with rank root. The input buffer is defined by
41
      the arguments sendbuf, count and datatype; the output buffer is defined by the arguments
42
      recvbuf, count and datatype; both have the same number of elements, with the same type.
43
      The routine is called by all group members using the same arguments for count, datatype, op,
44
      root and comm. Thus, all processes provide input buffers of the same length, with elements
45
      of the same type as the output buffer at the root. Each process can provide one element, or a
46
      sequence of elements, in which case the combine operation is executed element-wise on each
47
      entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains
48
```

two elements that are floating point numbers (count = 2 and $datatype = MPI_FLOAT$), then $recvbuf(1) = global \max(sendbuf(1))$ and $recvbuf(2) = global \max(sendbuf(2))$.

Section 5.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes to which each operation can be applied.

In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 5.9.5.

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

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

Advice to users. Some applications may not be able to ignore the non-associative nature of floating-point operations or may use user-defined operations (see Section 5.9.5) that require a special reduction order and cannot be treated as associative. Such applications should enforce the order of evaluation explicitly. For example, in the case of operations that require a strict left-to-right (or right-to-left) evaluation order, this could be done by gathering all operands at a single process (e.g., with MPI_GATHER), applying the reduction operation in the desired order (e.g., with MPI_REDUCE_LOCAL), and if needed, broadcast or scatter the result to the other processes (e.g., with MPI_BCAST). (End of advice to users.)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 5.9.2 and Section 5.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI_REDUCE in each process. MPI does not define which operations are used on which operands in this case. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described by such a datatype, which may contain several basic values. This is further explained in Section 5.9.5.

Advice to users. Users should make no assumptions about how MPI_REDUCE is implemented. It is safest to ensure that the same function is passed to MPI_REDUCE by each process. (*End of advice to users.*)

Overlapping datatypes are permitted in "send" buffers. Overlapping datatypes in "receive" buffers are erroneous and may give unpredictable results.

The "in place" option for intracommunicators is specified by passing the value 46 MPI_IN_PLACE to the argument sendbuf at the root. In such a case, the input data is taken 47 at the root from the receive buffer, where it will be replaced by the output data. 48

Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$

6 7

8 9

10 11

12

13

14

15

16

17

18

19 20

21

22

23

 24

25

26

27

28

29 30

 31

32

33

34

35

36

37

38

39 40

41

42

43

44

¹ If comm is an intercommunicator, then the call involves all processes in the intercom-² municator, but with one group (group A) defining the root process. All processes in the ³ other group (group B) pass the same value in argument root, which is the rank of the root ⁴ in group A. The root passes the value MPI_ROOT in root. All other processes in group A ⁵ pass the value MPI_PROC_NULL in root. Only send buffer arguments are significant in group ⁶ B and only receive buffer arguments are significant at the root.

7 8 9

14

29

34

5.9.2 Predefined Reduction Operations

¹⁰ The following predefined operations are supplied for MPI_REDUCE and related functions ¹¹ MPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER,

¹² MPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 5.12), and

¹³ MPI_REDUCE_LOCAL. These operations are invoked by placing the following in **op**.

15		
16	Name	Meaning
17		
18	MPI_MAX	maximum
19	MPI_MIN	minimum
	MPI_SUM	sum
20	MPI_PROD	product
21	MPI_LAND	logical and
22	MPI_BAND	bit-wise and
23	MPI_LOR	logical or
24	MPI_BOR	bit-wise or
25	MPI_LXOR	logical exclusive or (xor)
26	MPI_BXOR	bit-wise exclusive or (xor)
27	MPI_MAXLOC	max value and location
28	MPI_MINLOC	min value and location

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Section 5.9.4. For the other predefined operations, we enumerate below the allowed combinations of **op** and **datatype** arguments. First, define groups of MPI basic datatypes in the following way.

35MPI_INT, MPI_LONG, MPI_SHORT, C integer: 36 MPI_UNSIGNED_SHORT, MPI_UNSIGNED, 37 MPI_UNSIGNED_LONG, 38 MPI_LONG_LONG_INT, 39 MPI_LONG_LONG (as synonym), 40 MPI_UNSIGNED_LONG_LONG, 41 MPI_SIGNED_CHAR, 42MPI_UNSIGNED_CHAR, 43 MPI_INT8_T, MPI_INT16_T, 44MPI_INT32_T, MPI_INT64_T, 45MPI_UINT8_T, MPI_UINT16_T, MPI_UINT32_T, and MPI_UINT64_T 4647Fortran integer: MPI_INTEGER and handles returned from 48

		MPI_TYPE_CREATE_F90_INTEGER	1
		and, if available, MPI_INTEGER1,	2
		MPI_INTEGER2, MPI_INTEGER4,	3
		MPI_INTEGER8, and MPI_INTEGER16	4
	Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL,	5
		MPI_DOUBLE_PRECISION,	6
		MPI_LONG_DOUBLE,	7
		and handles returned from	8
		MPI_TYPE_CREATE_F90_REAL	9
		and, if available, MPI_REAL2,	
		MPI_REAL4, MPI_REAL8, and MPI_REAL16	10
	Logical:	MPI_LOGICAL, MPI_C_BOOL,	11
		and MPI_CXX_BOOL	12
	Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	13
	complex.	MPI_C_FLOAT_COMPLEX (as synonym),	14
		MPI_C_DOUBLE_COMPLEX,	15
		MPI_C_LONG_DOUBLE_COMPLEX,	16
		MPI_CXX_FLOAT_COMPLEX,	17
		/	18
		MPI_CXX_DOUBLE_COMPLEX,	19
		MPI_CXX_LONG_DOUBLE_COMPLEX,	20
		and handles returned from	21
		MPI_TYPE_CREATE_F90_COMPLEX	22
		and, if available, MPI_DOUBLE_COMPLEX,	23
		MPI_COMPLEX4, MPI_COMPLEX8,	24
		MPI_COMPLEX16, and MPI_COMPLEX32	25
	Byte:	MPI_BYTE	26
	Multi-language types:	MPI_AINT, MPI_OFFSET, and MPI_COUNT	
	Now, the valid datatypes for each ope	ration are specified below.	27
	, , , , , , , , , , , , , , , , , , , ,	1	28
			29
	Ор	Allowed Types	30
			31
	MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point,	32
	_) _	Multi-language types	33
	MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex,	34
		Multi-language types	35
	MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical	36
	MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte, Multi-language types	37
	, , , , , , , , , , , , , , , , , , ,		38
		ed datatypes are valid in all supported program-	39
n	ning languages, see also Reduce Operation	ns on page 674 in Section $17.2.6$.	40
	The following examples use intracomm	nunicators.	41
			42
E	Example 5.15		43
	A routine that computes the dot prod	luct of two vectors that are distributed across a	40
g	roup of processes and returns the answer	at node zero.	44 45
			45 46
			47
			48

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

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

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

47

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

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

5.9.4 MINLOC and MAXLOC

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

The operation that defines MPI_MAXLOC is:

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

where

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

and

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

MPI_MINLOC is defined similarly:

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

where

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

and

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

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

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

1		air, and ties are resolved according to the second		
3	component.			
4	The reduce operation is defined to operate on arguments that consist of a pair: value			
5	and index. For both Fortran and C, types are provided to describe the pair. The potentially			
6	mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, for Fortran, by having the MPI-provided type consist of a pair of the same type as value,			
7		· · · · · · · · · · · · · · · · · · ·		
8	and coercing the index to this type also. In C, the MPI-provided pair type has distinct			
9	types and the index is an int.			
10		MAXLOC in a reduce operation, one must provide		
11		air (value and index). MPI provides nine such		
12		PI_MAXLOC and MPI_MINLOC can be used with		
13	each of the following datatypes.			
13	Fortran:			
14	Name	Description		
	MPI_2REAL	pair of REALs		
16	MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISION variables		
17 18	MPI_2INTEGER	pair of INTEGERS		
18				
20				
20 21	C.			
21	C: Name	Description		
23	MPI_FLOAT_INT	Description float and int		
23	MPI_PLOAT_INT MPI_DOUBLE_INT	double and int		
24 25	MPI_LONG_INT	long and int		
25 26	MPI_2INT	pair of int		
	MPI_SHORT_INT	short and int		
27	MPI_LONG_DOUBLE_INT	long double and int		
28		-		
29 30	The datatype MPI_2REAL is as if defi	ned by the following (see Section 4.1).		
31 32	MPI_Type_contiguous(2, MPI_REAL, MP	I_2REAL);		
33	Cimilar statements apply for MDL 2007			
34		FEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.		
35	The datatype MPI_SHORT_INT is as if	defined by the following sequence of instructions.		
36	<pre>struct mystruct {</pre>			
37	short val;			
38	int rank;			
39	};			
40	<pre>type[0] = MPI_SHORT;</pre>			
41	<pre>type[1] = MPI_INT;</pre>			
42	disp[0] = 0;			
43	<pre>disp[1] = offsetof(struct mystruct,</pre>	rank);		
44	block[0] = 1;			
45	block[1] = 1;			
46	MPI_Type_create_struct(2, block, di	<pre>sp, type, MPI_SHORT_INT);</pre>		
47	••			
48				
-				

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

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

MPI_OP_CREATE(user_fn, commute, op)

if (myrank == root) {	1
/* read answer out	2
*/	3
<pre>minval = out.value;</pre>	4
<pre>minrank = out.index / LEN;</pre>	5
<pre>minindex = out.index % LEN;</pre>	6
}	7
	8
Rationale. The definition of MPI_MINLOC and MPI_MAXLOC given here has the	9
advantage that it does not require any special-case handling of these two operations:	10
they are handled like any other reduce operation. A programmer can provide his or	11
her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage	12
is that values and indices have to be first interleaved, and that indices and values have	13
to be coerced to the same type, in Fortran. (End of rationale.)	14
	15
5.9.5 User-Defined Reduction Operations	16
·	17
	18
MDL OD $CPEATE(user fr commute on)$	19

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

C binding

int MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)
F08 binding

```
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
```

F binding

MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR) EXTERNAL USER_FN LOGICAL COMMUTE INTEGER OP, IERROR

40 MPI_OP_CREATE binds a user-defined reduction operation to an 41 op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, 42MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 5.12), and 43 44MPI_REDUCE_LOCAL. The user-defined operation is assumed to be associative. If commute 45= true, then the operation should be both commutative and associative. If commute = false, 46then the order of operands is fixed and is defined to be in ascending, process rank order, 47beginning with process zero. The order of evaluation can be changed, talking advantage of 48

Unofficial Draft for Comment Only

26

27

28 29

30

31

32

33

34

35

36

37

38

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

The argument user_fn is the user-defined function, which must have the following four arguments: invec, inoutvec, len, and datatype. The ISO C protection for the function is the following

The ISO C prototype for the function is the following.

```
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
TYPE(C_PTR), VALUE :: invec, inoutvec
INTEGER :: len
TYPE(MPI_Datatype) :: datatype
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
```

<type> INVEC(LEN), INOUTVEC(LEN)

```
INTEGER LEN, DATATYPE
```

The datatype argument is a handle to the data type that was passed into the call to 20MPI_REDUCE. The user reduce function should be written such that the following holds: 21Let $u[0], \ldots, u[len-1]$ be the len elements in the communication buffer described by the 22 arguments invec, len and datatype when the function is invoked; let $v[0], \ldots, v[len-1]$ be len 23elements in the communication buffer described by the arguments inoutvec, len and datatype 24when the function is invoked; let $w[0], \ldots, w[len-1]$ be len elements in the communication 25buffer described by the arguments inoutvec, len and datatype when the function returns; 26then $w[i] = u[i] \circ v[i]$, for i=0, ..., len-1, where \circ is the reduce operation that the function 27computes. 28

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

- Rationale. The len argument allows MPI_REDUCE to avoid calling the function for
 each element in the input buffer. Rather, the system can choose to apply the function
 to chunks of input. In C, it is passed in as a reference for reasons of compatibility
 with Fortran.
- By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different data types. (*End of rationale.*)
 - General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.
- ⁴⁴ No MPI communication function may be called inside the user function. MPI_ABORT ⁴⁵ may be called inside the function in case of an error.
- Advice to users. Suppose one defines a library of user-defined reduce functions that
 are overloaded: the datatype argument is used to select the right execution path at each

6 7

8

9

10

11

12

13

14

15 16

17

18

19

42

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

The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a datatype handle. 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) {</pre>
    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.

$45 \\ 46$

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

1 The predefined reduce operations can be implemented as a library of user-defined $\mathbf{2}$ operations. However, better performance might be achieved if MPI_REDUCE handles 3 these functions as a special case. (End of advice to implementors.) 4 56 MPI_OP_FREE(op) 7 8 INOUT ор operation (handle) 9 10 C binding 11int MPI_Op_free(MPI_Op *op) 12F08 binding 13 MPI_Op_free(op, ierror) 14 TYPE(MPI_Op), INTENT(INOUT) :: op 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617F binding 18 MPI_OP_FREE(OP, IERROR) 19INTEGER OP, IERROR 20Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL. 2122 Example of User-defined Reduce 23 24 It is time for an example of user-defined reduction. The example in this section uses an 25intracommunicator. 2627**Example 5.20** Compute the product of an array of complex numbers, in C. 28typedef struct { 29double real, imag; 30 } Complex; 31 32 /* the user-defined function 33 34 */ void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr) 35 { 36 int i; 37 Complex c; 38 Complex *in = (Complex *)inP, *inout = (Complex *)inoutP; 39 40 for (i=0; i< *len; ++i) {</pre> 41 c.real = inout->real*in->real -42inout->imag*in->imag; 43 c.imag = inout->real*in->imag + 44 inout->imag*in->real; 45 *inout = c; 46 47in++; inout++; } 48

```
}
                                                                                       1
                                                                                       \mathbf{2}
                                                                                       3
/* and, to call it...
 */
                                                                                       4
                                                                                       5
. . .
                                                                                       6
                                                                                       7
    /* each process has an array of 100 Complexes
                                                                                       8
     */
    Complex a[100], answer[100];
                                                                                       9
                                                                                       10
    MPI_Op myOp;
                                                                                       11
    MPI_Datatype ctype;
                                                                                       12
    /* explain to MPI how type Complex is defined
                                                                                       13
                                                                                       14
     */
                                                                                       15
    MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
                                                                                       16
    MPI_Type_commit(&ctype);
                                                                                       17
    /* create the complex-product user-op
                                                                                       18
     */
    MPI_Op_create(myProd, 1, &myOp);
                                                                                       19
                                                                                       20
                                                                                       21
    MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
                                                                                       22
    /* At this point, the answer, which consists of 100 Complexes,
                                                                                       23
                                                                                       ^{24}
     * resides on process root
                                                                                       25
     */
                                                                                       26
                                                                                       27
Example 5.21 How to use the mpi_f08 interface of the Fortran MPI_User_function.
                                                                                       28
                                                                                       29
  subroutine my_user_function(invec, inoutvec, len, type)
                                                                  bind(c)
                                                                                       30
    use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
                                                                                       31
    use mpi_f08
                                                                                       32
    type(c_ptr), value :: invec, inoutvec
                                                                                       33
    integer :: len
```

5.9.6 All-Reduce

end subroutine

end if

type(MPI_Datatype) :: type

real, pointer :: invec_r(:), inoutvec_r(:)

if (type%MPI_VAL == MPI_REAL%MPI_VAL) then

inoutvec_r = invec_r + inoutvec_r

call c_f_pointer(invec, invec_r, (/ len /))

call c_f_pointer(inoutvec, inoutvec_r, (/ len /))

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.

34

35

36

37

38

39

40

41

42 43

44

45

46

1 MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm) 2 IN sendbuf starting address of send buffer (choice) 3 OUT recvbuf starting address of receive buffer (choice) 4 5IN number of elements in send buffer (non-negative intecount 6 ger) 7 IN datatype data type of elements of send buffer (handle) 8 IN ор operation (handle) 9 10 IN comm communicator (handle) 11 12C binding 13 int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, 14MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) 15F08 binding 16MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) 17TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 18 TYPE(*), DIMENSION(..) :: recvbuf 19 INTEGER, INTENT(IN) :: count 20TYPE(MPI_Datatype), INTENT(IN) :: datatype 21TYPE(MPI_Op), INTENT(IN) :: op 22 TYPE(MPI_Comm), INTENT(IN) :: comm 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24 25F binding 26MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 27<type> SENDBUF(*), RECVBUF(*) 28INTEGER COUNT, DATATYPE, OP, COMM, IERROR 29 If comm is an intracommunicator, MPI_ALLREDUCE behaves the same as 30 MPI_REDUCE except that the result appears in the receive buffer of all the group members. 31 32 Advice to implementors. The all-reduce operations can be implemented as a re-33 duce, followed by a broadcast. However, a direct implementation can lead to better 34 performance. (End of advice to implementors.) 3536 The "in place" option for intracommunicators is specified by passing the value 37 MPI_IN_PLACE to the argument sendbuf at all processes. In this case, the input data is 38 taken at each process from the receive buffer, where it will be replaced by the output data. 39 If comm is an intercommunicator, then the result of the reduction of the data provided 40 by processes in group A is stored at each process in group B, and vice versa. Both groups 41 should provide **count** and **datatype** arguments that specify the same type signature. 42The following example uses an intracommunicator. 43 44Example 5.22 45A routine that computes the product of a vector and an array that are distributed 46across a group of processes and returns the answer at all nodes (see also Example 5.16). 4748

```
1
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
                                                                                            \mathbf{2}
                        ! local slice of array
REAL a(m), b(m,n)
                                                                                             3
REAL c(n)
                        ! result
                                                                                            4
REAL sum(n)
INTEGER n, comm, i, j, ierr
                                                                                            5
                                                                                            6
                                                                                            7
! local sum
                                                                                             8
DO j= 1, n
  sum(j) = 0.0
                                                                                            9
                                                                                            10
  DO i = 1, m
                                                                                            11
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
                                                                                            12
END DO
                                                                                            13
                                                                                            14
                                                                                            15
! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
                                                                                            16
                                                                                            17
                                                                                            18
! return result at all nodes
                                                                                            19
RETURN
END
                                                                                            20
                                                                                            21
                                                                                            22
5.9.7
       Process-Local Reduction
                                                                                            23
The functions in this section are of importance to library implementors who may want to
                                                                                            ^{24}
implement special reduction patterns that are otherwise not easily covered by the standard
                                                                                            25
MPI operations.
                                                                                            26
    The following function applies a reduction operator to local arguments.
                                                                                            27
                                                                                            28
                                                                                            29
MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op)
                                                                                            30
  IN
           inbuf
                                        input buffer (choice)
                                                                                            31
                                                                                            32
  INOUT
           inoutbuf
                                        combined input and output buffer (choice)
                                                                                            33
  IN
                                        number of elements in inbuf and inoutbuf buffers (non-
           count
                                                                                            34
                                        negative integer)
                                                                                            35
  IN
           datatype
                                        data type of elements of inbuf and inoutbuf buffers
                                                                                            36
                                        (handle)
                                                                                            37
                                                                                            38
  IN
                                        operation (handle)
            ор
                                                                                            39
                                                                                            40
C binding
                                                                                            41
int MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,
                                                                                            42
               MPI_Datatype datatype, MPI_Op op)
                                                                                            43
F08 binding
                                                                                            44
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
                                                                                            45
    TYPE(*), DIMENSION(..), INTENT(IN) ::
                                                  inbuf
                                                                                            46
    TYPE(*), DIMENSION(..) :: inoutbuf
                                                                                            47
    INTEGER, INTENT(IN) :: count
                                                                                            48
```

```
1
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                 datatype
\mathbf{2}
          TYPE(MPI_Op), INTENT(IN) :: op
3
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
4
     F binding
5
     MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
6
          <type> INBUF(*), INOUTBUF(*)
7
          INTEGER COUNT, DATATYPE, OP, IERROR
8
9
         The function applies the operation given by op element-wise to the elements of inbuf
10
     and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined
11
     operations in Section 5.9.5. Both inbuf and inoutbuf (input as well as result) have the
12
     same number of elements given by count and the same datatype given by datatype. The
13
     MPI_IN_PLACE option is not allowed.
14
         Reduction operations can be queried for their commutativity.
15
16
     MPI_OP_COMMUTATIVE(op, commute)
17
18
       IN
                                             operation (handle)
                 ор
19
       OUT
                                             true if op is commutative, false otherwise (logical)
                 commute
20
21
     C binding
22
     int MPI_Op_commutative(MPI_Op op, int *commute)
23
24
     F08 binding
25
     MPI_Op_commutative(op, commute, ierror)
26
          TYPE(MPI_Op), INTENT(IN) :: op
27
          LOGICAL, INTENT(OUT) :: commute
28
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     F binding
30
     MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
^{31}
          LOGICAL COMMUTE
32
          INTEGER OP, IERROR
33
34
35
             Reduce-Scatter
     5.10
36
37
```

MPI includes variants of the reduce operations where the result is scattered to all processes in a group on return. One variant scatters equal-sized blocks to all processes, while another variant scatters blocks that may vary in size for each process.

```
40
41
```

38

39

```
42
43
```

3

- 44 45
- 46

47

5.10.1 MPI_REDUCE_SCATTER_BLOCK

MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm) IN sendbuf starting address of send buffer (choice) OUT recvbuf starting address of receive buffer (choice) IN element count per block (non-negative integer) recvcount IN datatype data type of elements of send and receive buffers (han-10 dle) 11 IN operation (handle) op 1213 IN communicator (handle) comm 1415C binding 16int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf, 17int recvcount, MPI_Datatype datatype, MPI_Op op, 18 MPI_Comm comm) 19 F08 binding 20MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 21ierror) 22 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 23TYPE(*), DIMENSION(..) :: recvbuf 24INTEGER, INTENT(IN) :: recvcount 2526TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op 27TYPE(MPI_Comm), INTENT(IN) :: comm 28 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 30 F binding 31MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 32 IERROR) 33 <type> SENDBUF(*), RECVBUF(*) 34 INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR 3536

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

Advice to implementors. The MPI_REDUCE_SCATTER_BLOCK routine is func-45tionally equivalent to: an MPI_REDUCE collective operation with count equal to 46recvcount*n, followed by an MPI_SCATTER with sendcount equal to recvcount. How-47ever, a direct implementation may run faster. (End of advice to implementors.) 48

Unofficial Draft for Comment Only

5

6

7 8

9

37

38

39

4041

42

¹ The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in ² the **sendbuf** argument on *all* processes. In this case, the input data is taken from the receive ³ buffer.

⁴ If comm is an intercommunicator, then the result of the reduction of the data provided ⁵ by processes in one group (group A) is scattered among processes in the other group (group ⁶ B) and vice versa. Within each group, all processes provide the same value for the recvcount ⁷ argument, and provide input vectors of count = n*recvcount elements stored in the send ⁸ buffers, where n is the size of the group. The number of elements count must be the same ⁹ for the two groups. The resulting vector from the other group is scattered in blocks of ¹⁰ 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.)

 $15 \\ 16$

11

12

13

14

```
17
18
```

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.

21 22 23

24

36

37

38

39

40

19

20

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

24			
25	IN	sendbuf	starting address of send buffer (choice)
26	OUT	recvbuf	starting address of receive buffer (choice)
27 28 29 30	IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements of the result distributed to each process.
30 31 32	IN	datatype	data type of elements of send and receive buffers (han-dle)
33	IN	ор	operation (handle)
34 35	IN	comm	communicator (handle)

C binding

int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf,

```
const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm)
```

⁴¹ F08 binding

```
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: recvcounts(*)
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
```

TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
F binding
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
IERROR)
<type> SENDBUF(*), RECVBUF(*)</type>
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

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

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

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

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

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

 24

1	5.11	Scan		
2 3	5 11 1	Inclusive Scan		
4	0.11.1			
5				
6	MPI_S	CAN(sendbuf, recvbuf, count, dat	atype, op, comm)	
7 8	IN	sendbuf	starting address of send buffer (choice)	
9	OUT	recvbuf	starting address of receive buffer (choice)	
10 11	IN	count	number of elements in input buffer (non-negative in-teger)	
12	IN	datatype	data type of elements of input buffer (handle)	
13 14	IN	ор	operation (handle)	
15	IN	comm	communicator (handle)	
16			, , , , , , , , , , , , , , , , , , ,	
17	C bind	ling		
18 19	int MP		, void* recvbuf, int count,	
20		MPI_Datatype dataty	pe, MPI_Op op, MPI_Comm comm)	
21	F08 bi	0		
22			, datatype, op, comm, ierror)	
23		<pre>PE(*), DIMENSION(), INTEN PE(*) DIMENSION() ··· </pre>		
24 25	TYPE(*), DIMENSION() :: recvbuf INTEGER, INTENT(IN) :: count			
26	TYPE(MPI_Datatype), INTENT(IN) :: datatype			
27	TYPE(MPI_Op), INTENT(IN) :: op			
28	TYPE(MPI_Comm), INTENT(IN) :: comm			
29	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
30 31	F bind	0		
32			, DATATYPE, OP, COMM, IERROR)	
33	<type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, IERROR</type>			
34				
35			MPI_SCAN is used to perform a prefix reduction on	
36 37	data distributed across the group. The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks $0, \ldots, i$			
38	(inclusive). The routine is called by all group members using the same arguments for count,			
39	datatype, op and comm, except that for user-defined operations, the same rules apply as			
40	for MPI_REDUCE. The type of operations supported, their semantics, and the constraints			
41	on send and receive buffers are as for MPI_REDUCE.			
42 43			imunicators is specified by passing MPI_IN_PLACE in he input data is taken from the receive buffer, and	
44		d by the output data.	is input data is taken nom the receive buildt, and	
45	-	is operation is invalid for interc	ommunicators.	
46				
47				
48				

5

43

44

4546

4748

5.11.2	Exclusive Scan		1
			2
			3
MPI_E	XSCAN(sendbuf, recvbuf, c	ount, datatype, op, comm)	4
IN	sendbuf	starting address of send buffer (choice)	5 6
OUT	recvbuf	starting address of receive buffer (choice)	7
IN	count	number of elements in input buffer (non-negative in-	8
		teger)	9 10
IN	datatype	data type of elements of input buffer (handle)	10
IN	ор	operation (handle)	12
IN	comm	intracommunicator (handle)	13
			14
C bind	ling		15
	0	sendbuf, void* recvbuf, int count,	16
	MPI_Datatype d	atatype, MPI_Op op, MPI_Comm comm)	17 18
F08 bi	Inding		19
		, count, datatype, op, comm, ierror)	20
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf			21
TYPE(*), DIMENSION() :: recvbuf			22
INTEGER, INTENT(IN) :: count			23
TY	PE(MPI_Datatype), INT	ENT(IN) :: datatype	24
TY	PE(MPI_Op), INTENT(IN)) :: op	25
TY	PE(MPI_Comm), INTENT(IN) :: comm	26
IN	TEGER, OPTIONAL, INTER	NT(OUT) :: ierror	27
F bind	ling		28
	0	, COUNT, DATATYPE, OP, COMM, IERROR)	29
	<pre>ype> SENDBUF(*), RECVI</pre>		30
	TEGER COUNT, DATATYPE		31
	-		32 33
		cator, MPI_EXSCAN is used to perform a prefix reduction	33 34
	0	roup. The value in recvbuf on the process with rank 0 is	04

0 undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process 35with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes 36 37 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 3839 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. 40 41 The type of operations supported, their semantics, and the constraints on send and receive 42buffers, are as for MPI_REDUCE.

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

This operation is invalid for intercommunicators.

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

5.11.3 Example using MPI_SCAN

The example in this section uses an intracommunicator.

Example 5.23

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

values v_1 v_2 v_8 15logicals 0 1 16 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 result17 18 The operator that produces this effect is 19 $\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\j\end{array}\right),$ 202122 where 23 $w = \begin{cases} u+v & \text{if } i=j \\ v & \text{if } i\neq j \end{cases}.$ 242526Note that this is a non-commutative operator. C code that implements it is given 27below. 2829 typedef struct { 30 double val; 31 int log; 32 } SegScanPair; 33 34 /* the user-defined function 35 */ 36 void segScan(SegScanPair *in, SegScanPair *inout, int *len, 37 MPI_Datatype *dptr) 38 { 39 int i; 40 SegScanPair c; 41 42for (i=0; i< *len; ++i) {</pre> 43 if (in->log == inout->log) 44 c.val = in->val + inout->val; 45 else 46 c.val = inout->val; 47 c.log = inout->log; 48

1

 $\mathbf{2}$

3

4

5 6

7

8 9

10

11

12

```
*inout = c;
        in++; inout++;
    }
}
```

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

```
int i,base;
SegScanPair
             a, answer;
MPI_Op
             myOp;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
MPI_Aint
             disp[2];
             blocklen[2] = \{ 1, 1\};
int
MPI_Datatype sspair;
/* explain to MPI how type SegScanPair is defined
 */
MPI_Get_address(&a, disp);
MPI_Get_address(&a.log, disp+1);
base = disp[0];
for (i=0; i<2; ++i) disp[i] -= base;</pre>
MPI_Type_create_struct(2, blocklen, disp, type, &sspair);
MPI_Type_commit(&sspair);
/* create the segmented-scan user-op
 */
MPI_Op_create(segScan, 0, &myOp);
. . .
MPI_Scan(&a, &answer, 1, sspair, myOp, comm);
```

5.12Nonblocking Collective Operations

As described in Section 3.7, performance of many applications can be improved by over-34 lapping communication and computation, and many systems enable this. Nonblocking 35collective operations combine the potential benefits of nonblocking point-to-point opera-36 tions, to exploit overlap and to avoid synchronization, with the optimized implementation 37 and message scheduling provided by collective operations [30, 34]. One way of doing this 38 would be to perform a blocking collective operation in a separate thread. An alternative 39 mechanism that often leads to better performance (e.g., avoids context switching, scheduler overheads, and thread management) is to use nonblocking collective communication [32]. 41

42The nonblocking collective communication model is similar to the model used for nonblocking point-to-point communication. A nonblocking call initiates a collective operation, 43 which must be completed in a separate completion call. Once initiated, the operation 44may progress independently of any computation or other communication at participating 45processes. In this manner, nonblocking collective operations can mitigate possible synchro-46nizing effects of collective operations by running them in the "background." In addition to 47

1

2

3

4

5

6

 $\overline{7}$

8 9

10

11

12

13

14

15

1617

18

19

20

21

22

23

 24

25

26

27

28

29

30 31

32 33

enabling communication-computation overlap, nonblocking collective operations can per form collective operations on overlapping communicators, which would lead to deadlocks
 with blocking operations. Their semantic advantages can also be useful in combination with
 point-to-point communication.

 $\mathbf{5}$ As in the nonblocking point-to-point case, all calls are local and return immediately, 6 irrespective of the status of other processes. The call initiates the operation, which indicates 7that the system may start to copy data out of the send buffer and into the receive buffer. 8 Once initiated, all associated send buffers and buffers associated with input arguments (such 9 as arrays of counts, displacements, or datatypes in the vector versions of the collectives) 10 should not be modified, and all associated receive buffers should not be accessed, until the 11collective operation completes. The call returns a request handle, which must be passed to 12a completion call.

13All completion calls (e.g., MPI_WAIT) described in Section 3.7.3 are supported for 14nonblocking collective operations. Similarly to the blocking case, nonblocking collective 15operations are considered to be complete when the local part of the operation is finished, 16i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be 17safely accessed and modified. Completion does not indicate that other processes have 18 completed or even started the operation (unless otherwise implied by the description of 19the operation). Completion of a particular nonblocking collective operation also does not 20indicate completion of any other posted nonblocking collective (or send-receive) operations, 21whether they are posted before or after the completed operation.

- 22 23
- $\frac{24}{25}$

26

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 27completes, the MPI_ERROR field in the associated status object is set appropriately, see 28Section 3.2.5 on page 32. The values of the MPI_SOURCE and MPI_TAG fields are unde-29fined. It is valid to mix different request types (i.e., any combination of collective requests, 30 I/O requests, generalized requests, or point-to-point requests) in functions that enable mul- 31 tiple completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or 32 MPI_CANCEL for a request associated with a nonblocking collective operation. Nonblock-33 ing collective requests created using the APIs described in this section are not persistent. 34However, persistent collective requests can be created using persistent collective operations 35 described in Sections 5.13 and 7.8. 36

- $37 \\ 38$
- 39 40

41

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.*)

⁴² Multiple nonblocking collective operations can be outstanding on a single communi-⁴³ cator. If the nonblocking call causes some system resource to be exhausted, then it will ⁴⁴ fail and generate an MPI exception. Quality implementations of MPI should ensure that ⁴⁵ this happens only in pathological cases. That is, an MPI implementation should be able to ⁴⁶ support a large number of pending nonblocking operations.

⁴⁷ Unlike point-to-point operations, nonblocking collective operations do not match with
 ⁴⁸ 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.

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

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

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

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

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

Progression rules for nonblocking collective operations are similar to progression of nonblocking point-to-point operations, refer to Section 3.7.4.

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

5.12.1 Nonblocking Barrier Synchronization

MPI_IBARRIER(comm , request)

IN	comm	communicator (handle)
OUT	request	communication request (handle)

44 C binding 45int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request) 46 F08 binding 4748

MPI_Ibarrier(comm, request, ierror)

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 $\overline{7}$

8

9

10

11 12

13

1415

16

17

18

19 20

21

22

23

2425

26

2728

2930

 31

32

33

34 35

1 2 3	TYPE	MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT ER, OPTIONAL, INTENT(OUT	C) :: request
4 5 6 7	F binding MPI_IBARF		JR)
8 9 10 11 12 13 14 15	a process a dent of wh are enforce communica called MP	notifies that it has reached t nether other processes have of ed at the corresponding comp ator case will complete only	ersion of MPI_BARRIER. By calling MPI_IBARRIER, he barrier. The call returns immediately, indepen- called MPI_IBARRIER. The usual barrier semantics pletion operation (test or wait), which in the intra- after all other processes in the communicator have municator case, it will complete when all processes BARRIER.
16 17 18 19 20 21	dent can o mant	computations between the \mathbb{N} overlap the barrier latency an	barrier can be used to hide latency. Moving indepen- MPI_IBARRIER and the subsequent completion call and therefore shorten possible waiting times. The se- when mixing collective operations and point-to-point s.)
22 23 24	5.12.2 N	onblocking Broadcast	
25	MPI_IBCA	ST(buffer, count, datatype, ro	oot comm request)
26			
27	INOUT	buffer	starting address of buffer (choice)
		`	. ,
27 28 29	INOUT	buffer	starting address of buffer (choice)
27 28	INOUT IN	buffer count	starting address of buffer (choice) number of entries in buffer (non-negative integer)
27 28 29 30	INOUT IN IN	buffer count datatype	starting address of buffer (choice) number of entries in buffer (non-negative integer) data type of buffer (handle)
27 28 29 30 31 32 33	INOUT IN IN IN	buffer count datatype root	starting address of buffer (choice) number of entries in buffer (non-negative integer) data type of buffer (handle) rank of broadcast root (integer)
27 28 29 30 31 32 33 34 35 36 37 38	INOUT IN IN IN OUT C binding int MPI_1	<pre>buffer count datatype root comm request g bcast(void* buffer, int MPI_Comm comm, MPI_1</pre>	<pre>starting address of buffer (choice) number of entries in buffer (non-negative integer) data type of buffer (handle) rank of broadcast root (integer) communicator (handle) communication request (handle) count, MPI_Datatype datatype, int root,</pre>
27 28 29 30 31 32 33 34 35 36 37	INOUT IN IN IN OUT C binding int MPI_J F08 bind	<pre>buffer count datatype root comm request g bcast(void* buffer, int MPI_Comm comm, MPI_I ing</pre>	<pre>starting address of buffer (choice) number of entries in buffer (non-negative integer) data type of buffer (handle) rank of broadcast root (integer) communicator (handle) communication request (handle) count, MPI_Datatype datatype, int root, Request *request)</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39	INOUT IN IN IN OUT C binding int MPI_I F08 bind MPI_Ibcas	<pre>buffer count datatype root comm request g bcast(void* buffer, int MPI_Comm comm, MPI_I ing</pre>	<pre>starting address of buffer (choice) number of entries in buffer (non-negative integer) data type of buffer (handle) rank of broadcast root (integer) communicator (handle) communication request (handle) count, MPI_Datatype datatype, int root, Request *request) be, root, comm, request, ierror)</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	INOUT IN IN IN OUT C binding int MPI_I F08 bind MPI_Ibcas TYPE(INTEC	<pre>buffer count datatype root comm request g bcast(void* buffer, int MPI_Comm comm, MPI_D ing st(buffer, count, datatyp (*), DIMENSION(), ASYNC EER, INTENT(IN) :: count</pre>	<pre>starting address of buffer (choice) number of entries in buffer (non-negative integer) data type of buffer (handle) rank of broadcast root (integer) communicator (handle) communication request (handle) communication request (handle) count, MPI_Datatype datatype, int root, Request *request) pe, root, comm, request, ierror) CHRONOUS :: buffer c, root</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	INOUT IN IN IN IN OUT C binding int MPI_I F08 bind MPI_Ibcas TYPE(INTEC	<pre>buffer count datatype root comm request g bbcast(void* buffer, int</pre>	<pre>starting address of buffer (choice) number of entries in buffer (non-negative integer) data type of buffer (handle) rank of broadcast root (integer) communicator (handle) communication request (handle) count, MPI_Datatype datatype, int root, Request *request) pe, root, comm, request, ierror) CHRONOUS :: buffer t, root U) :: datatype</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	INOUT IN IN IN IN OUT C binding int MPI_1 F08 bind MPI_Ibcas TYPE(INTEC TYPE(<pre>buffer count datatype root comm request g bcast(void* buffer, int MPI_Comm comm, MPI_D ing st(buffer, count, datatyp (*), DIMENSION(), ASYNC EER, INTENT(IN) :: count</pre>	<pre>starting address of buffer (choice) number of entries in buffer (non-negative integer) data type of buffer (handle) rank of broadcast root (integer) communicator (handle) communication request (handle) communication request (handle) count, MPI_Datatype datatype, int root, Request *request) e, root, comm, request, ierror) CHRONOUS :: buffer c, root l) :: datatype comm</pre>
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	INOUT IN IN IN OUT C binding int MPI_1 F08 bind MPI_Ibcas TYPE(TYPE(TYPE)	<pre>buffer count datatype root comm request B bcast(void* buffer, int MPI_Comm comm, MPI_J ing st(buffer, count, datatyp *), DIMENSION(), ASYNC ER, INTENT(IN) :: count MPI_Datatype), INTENT(IN) ::</pre>	<pre>starting address of buffer (choice) number of entries in buffer (non-negative integer) data type of buffer (handle) rank of broadcast root (integer) communicator (handle) communication request (handle) communication request (handle) count, MPI_Datatype datatype, int root, Request *request) e, root, comm, request, ierror) CHRONOUS :: buffer c, root N) :: datatype comm T) :: request</pre>

		TATYPE, ROOT, COMM, REQUEST, IERROR)	$\frac{1}{2}$
	<pre>:ype> BUFFER(*) ITEGER COUNT, DATATYPE,</pre>	ROOT, COMM, REQUEST, IERROR	3
		g variant of MPI_BCAST (see Section 5.4).	4
1.	ins can starts a nonblocking	, variant of MFI_BCAST (see Section 5.4).	5
Examp	le using MPI_IBCAST		6 7
	-	., .,	8
I ne ex	ample in this section uses a	an intracommunicator.	9
Exam	ple 5.24		10
	-	from process 0 to every process in the group, perform some	11
compu	tation on independent data	, and then complete the outstanding broadcast operation.	12
мт	PI_Comm comm;		13
	nt array1[100], array2[100]:	14 15
	nt root=0;		16
MI	PI_Request req;		17
•			18
	÷	MPI_INT, root, comm, &req);	19
	ompute(array2, 100); PI_Wait(&req, MPI_STATU	S TCNODE).	20
1.11	-1_wait(&req, mr1_SIATO	S_IGNORE,	21 22
5.12.3	Nonblocking Gather		22
0.22.0			24
			25
MPI_I	GATHER(sendbuf, sendcount request)	, sendtype, recvbuf, recvcount, recvtype, root, comm,	26 27
IN	sendbuf	starting address of send buffer (choice)	28
			29
IN	sendcount	number of elements in send buffer (non-negative integer)	30 31
IN	sendtype	data type of send buffer elements (handle)	32
OUT	recvbuf	address of receive buffer (choice, significant only at	33
		root)	34 35
IN	recvcount	number of elements for any single receive (non-negative	36
		integer, significant only at root)	37
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)	38 39
IN	root	rank of receiving process (integer)	40
IN	comm	communicator (handle)	41
OUT			42
001	request	communication request (handle)	43 44
C bin	ding		45
	_	sendbuf, int sendcount, MPI_Datatype sendtype,	46
	•	int recvcount, MPI_Datatype recvtype, int root,	47
	MPI_Comm comm,	MPI_Request *request)	48

1	F08 bindi	ng				
2	MPI_Igath	MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,				
3		root, comm, request, ierror)				
4	TYPE(*), DIMENSION(), INTEN	I(IN), ASYNCHRONOUS :: sendbuf			
5		*), DIMENSION(), ASYNCI				
6		ER, INTENT(IN) :: sendco				
7		MPI_Datatype), INTENT(IN)	VI VI			
8		TYPE(MPI_Comm), INTENT(IN) :: comm				
9		MPI_Request), INTENT(OUT)	-			
10	INTEG	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
11	F binding					
12 13		MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,				
13		ROOT, COMM, REQUEST,				
15	<type< td=""><td>> SENDBUF(*), RECVBUF(*)</td><td></td></type<>	> SENDBUF(*), RECVBUF(*)				
16	INTEG	ER SENDCOUNT, SENDTYPE, I	RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,			
17		IERROR				
18	This c	all starts a nonblocking varia	nt of MPI_GATHER (see Section 5.5).			
19		an starts a nonoroexing varia				
20						
21	MPI_IGATH	ι.	ndtype, recvbuf, recvcounts, displs, recvtype, root,			
22		comm, request)				
23 24	IN	sendbuf	starting address of send buffer (choice)			
24 25 26	IN	sendcount	number of elements in send buffer (non-negative integer)			
27	IN	sendtype	data type of send buffer elements (handle)			
28 29	OUT	recvbuf	address of receive buffer (choice, significant only at root)			
30 31 32 33	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)			
34 35 36 37	IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)			
38 39	IN	recvtype	data type of recv buffer elements (significant only at root) (handle)			
40 41	IN	root	rank of receiving process (integer)			
41	IN	comm	communicator (handle)			
43	OUT	request	communication request (handle)			
44	-	ı				
45	C binding	C binding				
46	int MPI_Igatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,					
47	<pre>void* recvbuf, const int recvcounts[], const int displs[],</pre>					
48						

1 MPI_Datatype recvtype, int root, MPI_Comm comm, 2 MPI_Request *request) 3 F08 binding 4 MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, 5 recvtype, root, comm, request, ierror) 6 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 7 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 8 INTEGER, INTENT(IN) :: sendcount, root 9 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*) 10 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 11 TYPE(MPI_Comm), INTENT(IN) :: comm 12TYPE(MPI_Request), INTENT(OUT) :: request 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415F binding 16MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 17RECVTYPE, ROOT, COMM, REQUEST, IERROR) 18 <type> SENDBUF(*), RECVBUF(*) 19 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, COMM, REQUEST, IERROR 2021This call starts a nonblocking variant of MPI_GATHERV (see Section 5.5). 22 235.12.4 Nonblocking Scatter 242526MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, 27request) 28 29IN sendbuf address of send buffer (choice, significant only at root) 30 number of elements sent to each process (non-negative IN sendcount 31integer, significant only at root) 32 IN sendtype data type of send buffer elements (significant only at 33 34 root) (handle) 35 OUT recvbuf address of receive buffer (choice) 36 number of elements in receive buffer (non-negative in-IN recvcount 37 teger) 38 IN recvtype data type of receive buffer elements (handle) 39 40 IN root rank of sending process (integer) 41 IN communicator (handle) comm 42OUT communication request (handle) request 43 44 C binding 45int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, 4647void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, 48

MPI_Comm comm, MPI_Request *request)

```
1
     F08 binding
\mathbf{2}
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
3
                     root, comm, request, ierror)
4
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
6
          INTEGER, INTENT(IN) :: sendcount, recvcount, root
7
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
8
          TYPE(MPI_Comm), INTENT(IN) :: comm
9
          TYPE(MPI_Request), INTENT(OUT) ::
                                                   request
10
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
11
     F binding
12
     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
13
                     ROOT, COMM, REQUEST, IERROR)
14
          <type> SENDBUF(*), RECVBUF(*)
15
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
16
                      IERROR
17
18
          This call starts a nonblocking variant of MPI_SCATTER (see Section 5.6).
19
20
     MPI_ISCATTERV(sendbuf, sendcounts, displs, sendtype, recybuf, recycount, recytype, root,
21
                     comm, request)
22
23
       IN
                  sendbuf
                                              address of send buffer (choice, significant only at root)
^{24}
       IN
                  sendcounts
                                              non-negative integer array (of length group size) spec-
25
                                              ifying the number of elements to send to each rank
26
       IN
                  displs
                                              integer array (of length group size). Entry i specifies
27
                                              the displacement (relative to sendbuf) from which to
28
                                              take the outgoing data to process i
29
30
       IN
                                              data type of send buffer elements (handle)
                  sendtype
^{31}
       OUT
                  recvbuf
                                              address of receive buffer (choice)
32
       IN
                  recvcount
                                              number of elements in receive buffer (non-negative in-
33
34
                                              teger)
35
       IN
                  recvtype
                                              data type of receive buffer elements (handle)
36
       IN
                                              rank of sending process (integer)
                  root
37
       IN
                  comm
                                              communicator (handle)
38
39
       OUT
                 request
                                              communication request (handle)
40
41
     C binding
42
     int MPI_Iscatterv(const void* sendbuf, const int sendcounts[],
43
                     const int displs[], MPI_Datatype sendtype, void* recvbuf,
44
                     int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
45
                     MPI_Request *request)
46
     F08 binding
47
48
```

```
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                         1
                                                                                         \mathbf{2}
               recvtype, root, comm, request, ierror)
                                                                                         3
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                         4
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
                                                                                         5
                                                                                         6
    INTEGER, INTENT(IN) :: recvcount, root
                                                                                         7
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                         8
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) ::
                                                                                         9
                                           request
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                                                                         10
                                           ierror
                                                                                         11
F binding
                                                                                         12
MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                         13
               RECVTYPE, ROOT, COMM, REQUEST, IERROR)
                                                                                         14
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         15
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
                                                                                         16
                COMM, REQUEST, IERROR
                                                                                         17
                                                                                         18
    This call starts a nonblocking variant of MPI_SCATTERV (see Section 5.6).
                                                                                         19
                                                                                        20
5.12.5 Nonblocking Gather-to-all
                                                                                        21
                                                                                        22
                                                                                        23
MPI_IALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,
                                                                                        24
               request)
                                                                                        25
  IN
            sendbuf
                                       starting address of send buffer (choice)
                                                                                         26
                                                                                        27
  IN
            sendcount
                                       number of elements in send buffer (non-negative inte-
                                                                                        28
                                       ger)
                                                                                        29
  IN
            sendtype
                                       data type of send buffer elements (handle)
                                                                                        30
  OUT
            recvbuf
                                       address of receive buffer (choice)
                                                                                        31
                                                                                        32
  IN
            recvcount
                                       number of elements received from any process (non-
                                                                                        33
                                       negative integer)
                                                                                        34
  IN
            recvtype
                                       data type of receive buffer elements (handle)
                                                                                        35
  IN
            comm
                                       communicator (handle)
                                                                                        36
                                                                                        37
  OUT
           request
                                       communication request (handle)
                                                                                        38
                                                                                        39
C binding
                                                                                         40
int MPI_Iallgather(const void* sendbuf, int sendcount,
                                                                                        41
               MPI_Datatype sendtype, void* recvbuf, int recvcount,
                                                                                        42
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
                                                                                        43
F08 binding
                                                                                         44
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                        45
               comm, request, ierror)
                                                                                         46
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                         47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                         48
```

1 2 3 4 5 6 7 8 9 10 11 12 13	<pre>INTEGER, INTENT(IN) :: sendcount, recvcount TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre> F binding MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR This call starts a nonblocking variant of MPI_ALLGATHER (see Section 5.7).</type>				
1415	MPI_IALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request)				
17	IN	sendbuf	starting address of send buffer (choice)		
18 19 20	IN	sendcount	number of elements in send buffer (non-negative inte- ger)		
21	IN	sendtype	data type of send buffer elements (handle)		
22	OUT	recvbuf	address of receive buffer (choice)		
23 24 25 26	IN	recvcounts	non-negative integer array (of length group size) con- taining the number of elements that are received from each process		
27 28 29	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i		
30	IN	recvtype	data type of receive buffer elements (handle)		
31 32	IN	comm	communicator (handle)		
33	OUT	request	communication request (handle)		
34 35 36 37 38 39	C binding int MPI_Iallgatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request* request)				
40	F08 bindi	ing			
41 42		atherv(sendbuf, sendcount	t, sendtype, recvbuf, recvcounts, displs,		
43		recvtype, comm, requ			
44	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf				
45 46	INTEGER, INTENT(IN) :: sendcount				
40 47	INTEG	ER, INTENT(IN), ASYNCHRON	NOUS :: recvcounts(*), displs(*)		
48	TYPE(<pre>MPI_Datatype), INTENT(IN)</pre>) :: sendtype, recvtype		

TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
<pre>F binding MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,</pre>							
This	This call starts a nonblocking variant of $MPI_ALLGATHERV$ (see Section 5.7).						
5.12.6 Nonblocking All-to-All Scatter/Gather							
¹⁵ MPI_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request) ¹⁶ ₁₇							
IN	sendbuf	starting address of send buffer (choice)	18 19				
IN	sendcount	number of elements sent to each process (non-negative integer)	20 21				
IN	sendtype	data type of send buffer elements (handle)	22 23				
OUT	recvbuf	address of receive buffer (choice)	24				
IN	recvcount	number of elements received from any process (non-negative integer)	25 26				
IN	recvtype	data type of receive buffer elements (handle)	27 28				
IN	comm	communicator (handle)	29				
OUT	request	communication request (handle)	30 31				
			32				
C bindin int MPI	g [alltoall(const void* sen	dbuf, int sendcount.	33				
_		pe, void* recvbuf, int recvcount,	34				
	MPI_Datatype recvtyp	pe, MPI_Comm comm, MPI_Request *request)	35 36				
F08 binding							
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 34							
TVDE	comm, request, ierro		39 40				
	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf4TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf4						
	GER, INTENT(IN) :: sendc		42				
TYPE	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 4						
TYPE(MPI_Comm), INTENT(IN) :: comm							
TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
F binding							

	216	CE	HAPTER 5. COLLECTIVE COMMUNICATION			
1 2 3 4	<pre>MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR</type></pre>					
5						
6 7	1 ms ca	This call starts a nonblocking variant of MPI_ALLTOALL (see Section 5.8).				
8 9	MPI_IALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request)					
10 11	IN	sendbuf	starting address of send buffer (choice)			
12 13	IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank			
14 15 16	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			
17 18	IN	sendtype	data type of send buffer elements (handle)			
19	OUT	recvbuf	address of receive buffer (choice)			
20 21 22	IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each rank			
23 24 25 26	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			
27	IN	recvtype	data type of receive buffer elements (handle)			
28	IN	comm	communicator (handle)			
29 30	OUT	request	communication request (handle)			
31	C binding					
32 33			ndbuf, const int sendcounts[],			
34	_	<pre>int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[],</pre>				
35		<pre>const int recvcounts[], const int rdispls[],</pre>				
36		MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)				
37 38		F08 binding				
39	MPI_Iallto	MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,				
40	rdispls, recvtype, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf					
41	TYPE(*), DIMENSION(), INTENT(IN), ASINCHRONOUS :: sendbul TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf					
42		INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),				
43 44		<pre>recvcounts(*), rdispls(*)</pre>				
45		TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype				
46	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request					
47		ER, OPTIONAL, INTENT(OUT)	-			
48	101201	,,				

F binding 1 2 MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, 3 RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) 4 INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 5RECVTYPE, COMM, REQUEST, IERROR 6 7 This call starts a nonblocking variant of MPI_ALLTOALLV (see Section 5.8). 8 9 10 MPI_IALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, 11 recvtypes, comm, request) 12IN sendbuf starting address of send buffer (choice) 13 IN sendcounts integer array (of length group size) specifying the num-14ber of elements to send to each rank (array of non-15negative integers) 1617 IN sdispls integer array (of length group size). Entry j specifies 18 the displacement in bytes (relative to sendbuf) from 19which to take the outgoing data destined for process j 20(array of integers) 21IN sendtypes array of datatypes (of length group size). Entry j spec-22 ifies the type of data to send to process j (array of 23handles) 24 OUT recvbuf address of receive buffer (choice) 2526IN recvcounts integer array (of length group size) specifying the num-27ber of elements that can be received from each rank 28(array of non-negative integers) 29IN integer array (of length group size). Entry i specifies rdispls 30 the displacement in bytes (relative to recvbuf) at which 31 to place the incoming data from process i (array of 32 integers) 33 IN recvtypes array of datatypes (of length group size). Entry i spec-34 ifies the type of data received from process i (array of 35handles) 36 37 IN communicator (handle) comm 38 OUT request communication request (handle) 39 40 C binding 41 int MPI_Ialltoallw(const void* sendbuf, const int sendcounts[], 42const int sdispls[], const MPI_Datatype sendtypes[], 43 void* recvbuf, const int recvcounts[], const int rdispls[], 44

const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)

F08 binding

45

46 47

```
1
     MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
\mathbf{2}
                    recvcounts, rdispls, recvtypes, comm, request, ierror)
3
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
4
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
5
6
                     recvcounts(*), rdispls(*)
7
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
8
                     recvtypes(*)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Request), INTENT(OUT) ::
                                                request
11
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
12
     F binding
13
     MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
14
                    RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
15
          <type> SENDBUF(*), RECVBUF(*)
16
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
17
                     RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
18
19
         This call starts a nonblocking variant of MPI_ALLTOALLW (see Section 5.8).
20
21
     5.12.7 Nonblocking Reduce
22
23
24
     MPI_IREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, request)
25
       IN
                 sendbuf
                                            address of send buffer (choice)
26
27
       OUT
                 recvbuf
                                            address of receive buffer (choice, significant only at
28
                                            root)
29
       IN
                 count
                                            number of elements in send buffer (non-negative inte-
30
                                            ger)
^{31}
       IN
                 datatype
                                            data type of elements of send buffer (handle)
32
33
       IN
                                            reduce operation (handle)
                 ор
34
       IN
                 root
                                            rank of root process (integer)
35
       IN
                 comm
                                            communicator (handle)
36
37
       OUT
                                            communication request (handle)
                 request
38
39
     C binding
40
     int MPI_Ireduce(const void* sendbuf, void* recvbuf, int count,
41
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
42
                    MPI_Request *request)
43
     F08 binding
44
     MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
45
                    ierror)
46
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::
47
                                                                     sendbuf
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
48
```

	ER, INTENT(IN) :: count,		1		
	MPI_Datatype), INTENT(IN)	v.	2 3		
	MPI_Op), INTENT(IN) :: c MPI_Comm), INTENT(IN) ::	-	4		
	MPI_COMMD, INTENT(IN) MPI_Request), INTENT(OUT)		5		
	ER, OPTIONAL, INTENT(OUT)	-	6		
			7		
F binding		NT, DATATYPE, OP, ROOT, COMM, REQUEST,	8		
MFI_IREDU	IERROR)	VI, DATATIFE, UF, RUUT, CUMM, REQUEST,	9		
<type< td=""><td>> SENDBUF(*), RECVBUF(*)</td><td></td><td>10</td></type<>	> SENDBUF(*), RECVBUF(*)		10		
• 1		ROOT, COMM, REQUEST, IERROR	11 12		
This c	all starts a nonblocking varia	nt of MPI_REDUCE (see Section $5.9.1$).	13		
1 1115 C	an starts a nonbiotking varia	it of wir I_REDUCE (see Section 5.3.1).	14		
Advic	ce to implementors. The im	plementation is explicitly allowed to use different	15		
0	0	ocking reduction operations that might change the	16		
		ons. However, as for MPI_REDUCE, it is strongly	17		
		be implemented so that the same result be obtained	18		
		the same arguments, appearing in the same order. A training that take advantage of the physical location	19 20		
	ccesses. (End of advice to imp		20		
-			22		
		which are not truly associative, the result delivered	23		
-	-	g reduction may not exactly equal the result deliv-	24		
	ered by the blocking reduction, even when specifying the same arguments in the same				
order. (End of advice to users.)					
5.12.8 No	5.12.8 Nonblocking All-Reduce				
J.12.0 N	5.12.0 Nonbiocking All-Neduce				
			29 30		
MPI_IALLF	REDUCE(sendbuf, recvbuf, cou	nt, datatype, op, comm, request)	31		
IN	sendbuf	starting address of send buffer (choice)	32		
OUT	recvbuf	starting address of receive buffer (choice)	33 34		
IN	count	number of elements in send buffer (non-negative inte-	35		
	count	ger)	36		
IN	datatype	data type of elements of send buffer (handle)	37		
IN	ор	operation (handle)	38 39		
IN	comm	communicator (handle)	40		
			41		
OUT	request	communication request (handle)	42		
C hinding	•		43		
C binding	-	udbuf, void* recybuf, int count	44		
	<pre>int MPI_Iallreduce(const void* sendbuf, void* recvbuf, int count,</pre>				
	MPI Request *request)				
FOS bind:	• • • 4				
F08 binding 48					

Unofficial Draft for Comment Only

1MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request, $\mathbf{2}$ ierror) 3 TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 4 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 5INTEGER, INTENT(IN) :: count 6 TYPE(MPI_Datatype), INTENT(IN) :: datatype 7 TYPE(MPI_Op), INTENT(IN) :: op 8 TYPE(MPI_Comm), INTENT(IN) :: comm 9 TYPE(MPI_Request), INTENT(OUT) :: request 10 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 11 F binding 12MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, 13 IERROR) 14 <type> SENDBUF(*), RECVBUF(*) 15INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 16 17 This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 5.9.6). 18 19 5.12.9 Nonblocking Reduce-Scatter with Equal Blocks 202122MPI_IREDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm, request) 2324IN sendbuf starting address of send buffer (choice) 2526OUT recvbuf starting address of receive buffer (choice) 27element count per block (non-negative integer) IN recvcount 28IN datatype data type of elements of send and receive buffers (han-29 dle) 30 31IN ор operation (handle) 32 IN comm communicator (handle) 33 OUT communication request (handle) 34 request 35 36 C binding 37 int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf, 38 int recvcount, MPI_Datatype datatype, MPI_Op op, 39MPI_Comm comm, MPI_Request *request) 40F08 binding 41 MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm, 42request, ierror) 43 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 44 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 45 INTEGER, INTENT(IN) :: recvcount 46 TYPE(MPI_Datatype), INTENT(IN) :: datatype 47 TYPE(MPI_Op), INTENT(IN) :: op 48

TYPE(1	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
<pre>F binding MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,</pre>					
This calculate tion $5.10.1$)	0	III OI MIFI_REDUCE_SCATTER_BEOCK (see Sec-	10 11 12		
5.12.10 N	lonblocking Reduce-Scatter	1	$13 \\ 14 \\ 15$		
MPI_IREDU	JCE_SCATTER(sendbuf, recvb	uf. recycounts, datatype, op. comm. request)	16 17		
IN	sendbuf	starting address of send buffer (choice)	18		
OUT	recvbuf	starting address of receive buffer (choice)	19		
IN	recvcounts	non-negative integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.	20 21 22 23		
IN	datatype		24		
IN	ор	operation (handle)	25		
IN	comm	communicator (handle)	26		
OUT			27 28		
C binding			29 30		
0			31		
_			32		
	MPI_Comm comm, MPI_Re	equest *request)	33		
F08 bindi	ng		34		
	0	uf recycounts datatype op comm	35 36		
	request, ierror)	3	37		
			38		
), DIMENSION(), ASYNCH ER, INTENT(IN), ASYNCHRON	OUS :: recycounts()	39		
	<pre>MPI_Datatype), INTENT(IN)</pre>	·· datatype	$\frac{40}{41}$		
TYPE(1	MPI_Op), INTENT(IN) :: o	n	+1 42		
	<pre>MPI_Comm), INTENT(IN) ::</pre>	4	43		
	<pre>MPI_Request), INTENT(OUT) ER, OPTIONAL, INTENT(OUT)</pre>	- 4	44		
			45		
F binding			46 17		
MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, REQUEST, IERROR) 42					

222 1 <type> SENDBUF(*), RECVBUF(*) $\mathbf{2}$ INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR 3 This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 5.10.2). 4 55.12.11 Nonblocking Inclusive Scan 6 7 8 9 MPI_ISCAN(sendbuf, recvbuf, count, datatype, op, comm, request) 10 IN sendbuf starting address of send buffer (choice) 11 OUT recvbuf starting address of receive buffer (choice) 12IN 13 count number of elements in input buffer (non-negative in-14teger) 15IN datatype data type of elements of input buffer (handle) 16IN operation (handle) op 1718 IN communicator (handle) comm 19OUT request communication request (handle) 2021C binding 22 int MPI_Iscan(const void* sendbuf, void* recvbuf, int count, 23 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 24MPI_Request *request) 2526F08 binding 27MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: 28sendbuf 29 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 30 INTEGER, INTENT(IN) :: count 31TYPE(MPI_Datatype), INTENT(IN) :: datatype 32 TYPE(MPI_Op), INTENT(IN) :: op 33 TYPE(MPI_Comm), INTENT(IN) :: comm 34 TYPE(MPI_Request), INTENT(OUT) :: request 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 F binding 37 MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 38 <type> SENDBUF(*), RECVBUF(*) 39 INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 40 41 This call starts a nonblocking variant of MPI_SCAN (see Section 5.11).

5.12.12 Nonblocking Exclusive Scan

			2		
			3		
MPI.	_IEXSCAN(sendbuf, recvbut	f, count, datatype, op, comm, request)	4		
IN	sendbuf	starting address of send buffer (choice)	5		
			6 7		
Οι	recvbut	starting address of receive buffer (choice)	8		
IN	count	number of elements in input buffer (non-negative in- teger)	9		
IN	datatype	data type of elements of input buffer (handle)	10 11		
IN	ор	operation (handle)	12		
IN	comm	intracommunicator (handle)	13		
OL			14		
00	IT request	communication request (handle)	15		
C h	inding		16		
	0	l* sendbuf, void* recvbuf, int count,	17		
1110		datatype, MPI_Op op, MPI_Comm comm,	18 19		
	MPI_Request >		20		
_	-		20		
	binding		22		
MP1_		<pre>puf, count, datatype, op, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf</pre>	23		
		, ASYNCHRONOUS :: recvbuf	24		
	INTEGER, INTENT(IN) ::		25		
	TYPE(MPI_Datatype), IN		26		
	TYPE(MPI_Op), INTENT(I	• •	27		
	TYPE(MPI_Comm), INTENT	-	28		
	TYPE(MPI_Request), INT		29		
	INTEGER, OPTIONAL, INT	-	30		
	1.		31		
	nding		32 33		
MP1_	4PI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)				
	<type> SENDBUF(*), REC</type>	PE, OP, COMM, REQUEST, IERROR	34		
	INTEGER COONT, DATAIT	L, OI, COURT, REQUENT, TERROR	35 36		
	This call starts a nonblock	ting variant of MPI_EXSCAN (see Section 5.11.2).	30		
			38		

5.13 Persistent Collective Operations

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

$47 \\ 48$

 41

In terms of data movement, each persistent collective operation has the same effect as its blocking and nonblocking counterparts for intracommunicators and intercommunicators after completion. Likewise, upon completion, persistent collective reduction operations perform the same operation as their blocking and nonblocking counterparts, and the same restrictions and recommendations on reduction orders apply (see also Section 5.9.1).

⁶ Initialization calls for MPI persistent collective operations are non-local and follow all ⁷ the existing rules for collective operations, in particular ordering; programs that do not ⁸ conform to these restrictions are erroneous. After initialization, all arrays associated with ⁹ input arguments (such as arrays of counts, displacements, and datatypes in the vector ¹⁰ versions of the collectives) must not be modified until the corresponding persistent request ¹¹ is freed with MPI_REQUEST_FREE.

¹² The request argument is an output argument that can be used zero or more times with ¹³ MPI_START or MPI_STARTALL in order to start the collective operation. The request is ¹⁴ initially inactive after the initialization call. Once initialized, persistent collective operations ¹⁵ can be started in any order and the order can differ among processes in the communicator.

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

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

29 30

17

18

19

20

21 22

23

 24

25

26

27

28

A request must be inactive when it is started. Starting the operation makes the request 31 active. Once any process starts a persistent collective operation, it must complete that 32 operation and all other processes in the communicator must eventually start (and complete) 33 the same persistent collective operation. Persistent collective operations cannot be matched 34with blocking or nonblocking collective operations. Completion of a persistent collective 35 operation makes the corresponding request inactive. After starting a persistent collective 36 operation, all associated send buffers must not be modified and all associated receive buffers 37 must not be accessed until the corresponding persistent request is completed. 38

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

For every nonblocking collective communication operation in MPI, there is a corresponding persistent collective operation with the analogous API signature.

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

5.13.1 F	Persistent Barrier Syı	nchronization	6		
	5		7		
			8		
MPI_BAR	RIER_INIT(comm, in	fo, request)	9 10		
IN	comm	communicator (handle)	11		
IN	info	info argument (handle)	12		
OUT	request	communication request (handle)	13		
001	request	communication request (nancie)	14		
C bindin	١ <u>٥</u> .		15		
	0	Comm comm, MPI_Info info, MPI_Request *request)	16 17		
FOS hind	ling		18		
F08 bind		fo, request, ierror)	19		
	(MPI_Comm), INTEN	-	20		
	(MPI_Info), INTEN		21		
TYPE	(MPI_Request), IN	TENT(OUT) :: request	22		
INTE	GER, OPTIONAL, IN	TENT(OUT) :: ierror	23		
F bindin	g		24 25		
	0	FO, REQUEST, IERROR)	26		
INTE	GER COMM, INFO, R	EQUEST, IERROR	27		
Creat	tes a persistent collec	tive communication request for the barrier operation.	28		
0	· · · · · · · · · · · · · · · · · · ·	The second	29		
5.13.2 F	Persistent Broadcast		30		
			31 32		
			33		
MPI_BCA	ST_INIT(buffer, cour	nt, datatype, root, comm, info, request)	34		
INOUT	buffer	starting address of buffer (choice)	35		
IN	count	number of entries in buffer (non-negative integer)	36		
IN	datatype	data type of buffer (handle)	37		
			38		
IN	root	rank of broadcast root (integer)	39 40		
IN	comm	communicator (handle)	41		
IN	info	info argument (handle)	42		
OUT	request	communication request (handle)	43		
			44		
C bindin	•		45 46		
int MPI_	Int MPI_bcast_init(void* builer, int count, MPI_batatype datatype,				
	int root, MP	YI_Comm comm, MPI_Info info, MPI_Request *request)	47 48		

Unofficial Draft for Comment Only

 $\mathbf{5}$

```
1
     F08 binding
\mathbf{2}
     MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
3
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
4
          INTEGER, INTENT(IN) :: count, root
5
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
          TYPE(MPI_Comm), INTENT(IN) :: comm
7
          TYPE(MPI_Info), INTENT(IN) ::
                                              info
8
          TYPE(MPI_Request), INTENT(OUT) :: request
9
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
10
     F binding
11
     MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)
12
          <type> BUFFER(*)
13
          INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR
14
15
          Creates a persistent collective communication request for the broadcast operation.
16
17
     5.13.3 Persistent Gather
18
19
20
      MPI_GATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
21
                     info, request)
22
       IN
                 sendbuf
                                              starting address of send buffer (choice)
23
24
       IN
                 sendcount
                                              number of elements in send buffer (non-negative inte-
25
                                              ger)
26
       IN
                 sendtype
                                              data type of send buffer elements (handle)
27
       OUT
                 recvbuf
                                              address of receive buffer (choice, significant only at
28
                                              root)
29
30
       IN
                                              number of elements for any single receive (non-negative
                  recvcount
^{31}
                                              integer, significant only at root)
32
                                              data type of recv buffer elements (significant only at
       IN
                  recvtype
33
                                              root) (handle)
34
                                              rank of receiving process (integer)
       IN
                  root
35
36
       IN
                                              communicator (handle)
                 comm
37
       IN
                 info
                                              info argument (handle)
38
       OUT
                 request
                                              communication request (handle)
39
40
41
     C binding
42
      int MPI_Gather_init(const void* sendbuf, int sendcount,
                     MPI_Datatype sendtype, void* recvbuf, int recvcount,
43
                     MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
44
45
                     MPI_Request *request)
46
     F08 binding
47
48
```

MPI_Gat	her_init(sendbuf, send	count, sendtype, recvbuf, recvcount, recvtype,	1			
	root, comm, info	, request, ierror)	2			
		NTENT(IN), ASYNCHRONOUS :: sendbuf	3 4			
	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf					
	INTEGER, INTENT(IN) :: sendcount, recvcount, root					
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype					
	E(MPI_Comm), INTENT(IN		7 8			
	TYPE(MPI_Info), INTENT(IN) :: info					
	E(MPI_Request), INTENT EGER, OPTIONAL, INTENT	-	9 10			
			11			
F bindi	0		12			
MPI_GAT		COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	13			
), REQUEST, IERROR)	14			
• •	pe> SENDBUF(*), RECVBU		15			
1 N I .	REQUEST, IERROR	PE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,	16 17			
Crea	ates a persistent collective	communication request for the gather operation.	18			
			19			
	/		20			
MPI_GA	I HERV_INI I (sendbut, send comm, info, request	count, sendtype, recvbuf, recvcounts, displs, recvtype, root,)	21 22			
IN	sendbuf	starting address of send buffer (choice)	23			
IN	sendcount	number of elements in send buffer (non-negative inte-	24			
	Sendebune	ger)	25 26			
IN	sendtype	data type of send buffer elements (handle)	27			
OUT	recvbuf	address of receive buffer (choice, significant only at root)	28 29			
IN	recvcounts	non-negative integer array (of length group size) con-	30			
		taining the number of elements that are received from	31			
		each process (significant only at root)	32			
IN	displs	integer array (of length group size). Entry i specifies	33 34			
	dispis	the displacement relative to recvbuf at which to place	35			
		the incoming data from process i (significant only at	36			
		root)	37			
IN	recvtype	data type of recv buffer elements (significant only at	38			
		root) (handle)	39			
IN	root	rank of receiving process (integer)	40			
IN	comm	communicator (handle)	41 42			
IN	info		42 43			
		info argument (handle)	44			
OUT	request	communication request (handle)	45			
			46			
C bindi	ng		47			

```
1
     int MPI_Gatherv_init(const void* sendbuf, int sendcount,
\mathbf{2}
                   MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
3
                   const int displs[], MPI_Datatype recvtype, int root,
4
                   MPI_Comm comm, MPI_Info info, MPI_Request *request)
5
     F08 binding
6
     MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
7
                   recvtype, root, comm, info, request, ierror)
8
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
         INTEGER, INTENT(IN) :: sendcount, root
11
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
12
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Info), INTENT(IN) :: info
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     F binding
19
     MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
20
                   RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
21
         <type> SENDBUF(*), RECVBUF(*)
22
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
23
                    COMM, INFO, REQUEST, IERROR
24
         Creates a persistent collective communication request for the gathery operation.
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

5.13.4 Persistent Scatter

MPI_SCATTER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, com		
	info, request)	
INI	م م م م الم ي ب	

IN	sendbuf	sendbuf address of send buffer (choice, significant only at root)			
IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)	8 9		
IN	sendtype	data type of send buffer elements (significant only at root) (handle)	10 11		
OUT	recvbuf	address of receive buffer (choice)	12 13		
IN	recvcount	number of elements in receive buffer (non-negative in- teger)	14 15		
IN	recvtype	data type of receive buffer elements (handle)	16		
IN	root	rank of sending process (integer)	17 18		
IN	comm	communicator (handle)	19		
IN	info	info argument (handle)	20		
OUT	request	communication request (handle)	21 22		
			23		
C binding					
int MPI_S	int MPI_Scatter_init(const void* sendbuf, int sendcount, 24				
MPI_Datatype sendtype, void* recvbuf, int recvcount, 26					

_		-	-
	MPI_Datatype sendtype,	<pre>void* recvbuf, int</pre>	recvcount,
	MPI_Datatype recvtype,	<pre>int root, MPI_Comm</pre>	comm, MPI_Info info,
	MPI_Request *request)		

F08 binding

ros binding	
MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,	30
recvtype, root, comm, info, request, ierror)	31
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	32
	33
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	34
INTEGER, INTENT(IN) :: sendcount, recvcount, root	35
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	
TYPE(MPI_Comm), INTENT(IN) :: comm	36
TYPE(MPI_Info), INTENT(IN) :: info	37
TYPE(MPI_Request), INTENT(OUT) :: request	38
	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
F binding	41
MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	42
RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)	43
<type> SENDBUF(*), RECVBUF(*)</type>	44
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,	45

REQUEST, IERROR

Creates a persistent collective communication request for the scatter operation.

Unofficial Draft for Comment Only

 $46 \\ 47$

1 2	MPI_SCAT	TERV_INIT(sendbuf, sendcou root, comm, info, request	nts, displs, sendtype, recvbuf, recvcount, recvtype,	
$\frac{3}{4}$	IN	sendbuf	address of send buffer (choice, significant only at root)	
5 6	IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	
7 8 9	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i	
10 11	IN	sendtype	data type of send buffer elements (handle)	
12	OUT	recvbuf	address of receive buffer (choice)	
13 14	IN	recvcount	number of elements in receive buffer (non-negative in-teger)	
15 16	IN	recvtype	data type of receive buffer elements (handle)	
17	IN	root	rank of sending process (integer)	
18	IN	comm	communicator (handle)	
19 20	IN	info	info argument (handle)	
20 21	OUT	request	communication request (handle)	
25 26 27 28 29	²⁶ int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, ²⁷ MPI_Info info, MPI_Request *request)			
30 31 32 33 34 35 36 37 38 39 40	MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*) INTEGER, INTENT(IN) :: recvcount, root TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: info TYPE(MPI_Info), INTENT(IN) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
41 42 43 44 45 46 47 48	<pre>MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,</pre>			

5.13.5 Persistent Gather-to-all

0.10			2
MPI	ALLGATHER INIT(sendbuf	f, sendcount, sendtype, recvbuf, recvcount, recvtype, comm,	3 4
	info, request)		5
IN	sendbuf	starting address of send buffer (choice)	6 7
IN	sendcount	number of elements in send buffer (non-negative integer)	8 9
IN	sendtype	data type of send buffer elements (handle)	10
οι	IT recvbuf	address of receive buffer (choice)	11 12
IN	recvcount	number of elements received from any process (non-negative integer)	$\frac{13}{14}$
IN	recvtype	data type of receive buffer elements (handle)	15
IN	comm	communicator (handle)	16 17
IN	info	info argument (handle)	18
οι	IT request	communication request (handle)	19
			20 21
	inding		21
int	e e	st void* sendbuf, int sendcount,	23
	<pre>MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,</pre>		
	MPI_Request *request)		
	-		26
F08 binding			27
MP1_	-	, sendcount, sendtype, recvbuf, recvcount,	28 29
	• =	m, info, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf	30
	TYPE(*), DIMENSION(),		31
	INTEGER, INTENT(IN) ::		32
		<pre>IENT(IN) :: sendtype, recvtype</pre>	33
	TYPE(MPI_Comm), INTENT((IN) :: comm	34
	TYPE(MPI_Info), INTENT		35
	TYPE(MPI_Request), INTE	-	36
	INTEGER, OPTIONAL, INTE	ENT(OUT) :: ierror	37
\mathbf{F} bi	nding		38 39
MPI_	ALLGATHER_INIT(SENDBUF,	, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	40
		M, INFO, REQUEST, IERROR)	41
	<type> SENDBUF(*), RECV</type>		42
		DTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	43
	IERROR		44
	Creates a persistent collection	ive communication request for the allgather operation.	45
			46
			47 48
			40

1 2	MPI_ALLG	ATHERV_INIT(sendbuf, sendc comm, info, request)	ount, sendtype, recvbuf, recvcounts, displs, recvtype,	
3	IN	sendbuf	starting address of send buffer (choice)	
4 5 6	IN	sendcount	number of elements in send buffer (non-negative integer)	
7	IN	sendtype	data type of send buffer elements (handle)	
8	OUT	recvbuf	address of receive buffer (choice)	
9 10	IN	recvcounts	non-negative integer array (of length group size) con-	
10 11 12			taining the number of elements that are received from each process	
13 14 15	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	
16	IN	recvtype	data type of receive buffer elements (handle)	
17 18	IN	comm	communicator (handle)	
19	IN	info	info argument (handle)	
20	OUT	request	communication request (handle)	
24 25 26 27 28	MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info_info_MPI_Bequest* request)			
29	F08 bindi	0	count, sendtype, recvbuf, recvcounts,	
30	m 1_A11ga		nm, info, request, ierror)	
31	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf			
32	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf			
33 34		ER, INTENT(IN) :: sendco		
35		MPI_Datatype), INTENT(IN)	<pre>IOUS :: recvcounts(*), displs(*)</pre>	
36	TYPE(MPI_Comm), INTENT(IN) :: comm			
37	TYPE(MPI_Info), INTENT(IN) :: info			
38 39		MPI_Request), INTENT(OUT)	-	
40	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
41	F binding	-		
42	MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,			
43	DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*)</type>			
44 45	01	-	RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	
45		INFO, REQUEST, IERRO		
47 48	Creates a persistent collective communication request for the allgathery operation.			

5.13.6 Persistent All-to-All Scatter/Gather

	request)	
IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each process (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)
	MDT Demos at the	
F08 bind	MPI_Request *r ding	request)
TYPE TYPE INTE TYPE TYPE TYPE	<pre>ding coall_init(sendbuf, s</pre>	<pre>sendcount, sendtype, recvbuf, recvcount, , info, request, ierror) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf sendcount, recvcount ENT(IN) :: sendtype, recvtype IN) :: comm IN) :: info NT(OUT) :: request</pre>
MPI_Allt TYPE TYPE TYPE TYPE TYPE TYPE INTE F bindir MPI_ALLT	ding coall_init(sendbuf, s recvtype, comm E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Comm), INTENT(S E(MPI_Info), INTENT(S E(MPI_Request), INTE EGER, OPTIONAL, INTE EGER, OPTIONAL, INTE OALL_INIT(SENDBUF, S	<pre>sendcount, sendtype, recvbuf, recvcount, , info, request, ierror) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf sendcount, recvcount ENT(IN) :: sendtype, recvtype IN) :: comm IN) :: info NT(OUT) :: request NT(OUT) :: ierror SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, , INFO, REQUEST, IERROR)</pre>
PI_Allt TYPE TYPE INTE TYPE TYPE TYPE INTE F bindir PI_ALLT <typ< td=""><td>ding coall_init(sendbuf, a recvtype, comm E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Comm), INTENT(E(MPI_Comm), INTENT(E(MPI_Info), INTENT(E(MPI_Request), INTE EGER, OPTIONAL, INTE EGER, OPTIONAL, INTE Mg COALL_INIT(SENDBUF, S RECVTYPE, COMM be> SENDBUF(*), RECV</td><td><pre>sendcount, sendtype, recvbuf, recvcount, , info, request, ierror) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf sendcount, recvcount ENT(IN) :: sendtype, recvtype IN) :: comm IN) :: info NT(OUT) :: request NT(OUT) :: ierror SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, , INFO, REQUEST, IERROR)</pre></td></typ<>	ding coall_init(sendbuf, a recvtype, comm E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Comm), INTENT(E(MPI_Comm), INTENT(E(MPI_Info), INTENT(E(MPI_Request), INTE EGER, OPTIONAL, INTE EGER, OPTIONAL, INTE Mg COALL_INIT(SENDBUF, S RECVTYPE, COMM be> SENDBUF(*), RECV	<pre>sendcount, sendtype, recvbuf, recvcount, , info, request, ierror) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf sendcount, recvcount ENT(IN) :: sendtype, recvtype IN) :: comm IN) :: info NT(OUT) :: request NT(OUT) :: ierror SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, , INFO, REQUEST, IERROR)</pre>
(PI_Allt TYPE TYPE INTE TYPE TYPE TYPE INTE F bindir (PI_ALLT <typ INTE</typ 	ding coall_init(sendbuf, a recvtype, comm E(*), DIMENSION(), E(*), DIMENSION(), EGER, INTENT(IN) :: E(MPI_Datatype), INTE E(MPI_Comm), INTENT() E(MPI_Comm), INTENT() E(MPI_Comm), INTENT() E(MPI_Request), INTENT() E(MPI_Request), INTENT() EGER, OPTIONAL, INTEN EGER, OPTIONAL, INTEN EGER, OPTIONAL, INTEN EGER, SENDBUF(*), RECVI EGER SENDCOUNT, SEND IERROR	<pre>sendcount, sendtype, recvbuf, recvcount, , info, request, ierror) INTENT(IN), ASYNCHRONOUS :: sendbuf ASYNCHRONOUS :: recvbuf sendcount, recvcount ENT(IN) :: sendtype, recvtype IN) :: comm IN) :: info NT(OUT) :: request NT(OUT) :: request NT(OUT) :: ierror SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, I, INFO, REQUEST, IERROR) BUF(*)</pre>

12	MPI_ALLT	OALLV_INIT(sendbuf, sendcou recvtype, comm, info, rec	ınts, sdispls, sendtype, recvbuf, recvcounts, rdispls, juest)	
3	IN	sendbuf	starting address of send buffer (choice)	
4 5 6	IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each rank	
7 8 9	IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	
10 11	IN	sendtype	data type of send buffer elements (handle)	
12	OUT	recvbuf	address of receive buffer (choice)	
13 14 15	IN	recvcounts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each rank	
16 17 18 19	IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	
20	IN	recvtype	data type of receive buffer elements (handle)	
21	IN	comm	communicator (handle)	
22 23	IN	info	info argument (handle)	
24	OUT	request	communication request (handle)	
26 27 28 29 30 31	<pre>int MPI_Alltoallv_init(const void* sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_info info, MPI_Datatype recvtype, MPI_Comm comm, MPI_info info,</pre>			
32	F08 bind	ing		
33 34 35 36 37 38 39 40 41 42 43 44	MPI_Allto TYPE TYPE INTE TYPE TYPE TYPE	<pre>pallv_init(sendbuf, sendco recvcounts, rdispls, (*), DIMENSION(), INTENT (*), DIMENSION(), ASYNCH</pre>	<pre>HRONOUS :: recvbuf NOUS :: sendcounts(*), sdispls(*), pls(*)) :: sendtype, recvtype comm info) :: request</pre>	
45 46 47 48		DALLV_INIT(SENDBUF, SENDCO	DUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVTYPE, COMM, INFO, REQUEST, IERROR)	

INTE	· · · · ·	LS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	1 2	
Crea	Creates a persistent collective communication request for the alltoally operation			
	5			
MPI_ALL	TOALLW_INIT(sendbuf, sendco recvtypes, comm, info, r	ounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, request)	6 7	
IN	sendbuf	starting address of send buffer (choice)	8 9	
IN	sendcounts	integer array (of length group size) specifying the num- ber of elements to send to each rank (array of non- negative integers)	10 11 12	
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)	13 14 15 16 17	
IN	sendtypes	array of datatypes (of length group size). Entry j spec- ifies the type of data to send to process j (array of handles)	18 19 20	
OUT	recvbuf	address of receive buffer (choice)	21	
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)	22 23 24 25	
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 28 29	
IN	recvtypes	array of datatypes (of length group size). Entry i spec- ifies the type of data received from process i (array of handles)	30 31 32	
IN	comm	communicator (handle)	33 34	
IN	info	info argument (handle)	35	
OUT	request	communication request (handle)	36	
C bindir	ıg		37 38	
int MPI_	<pre>const int sdispls[] void* recvbuf, cons</pre>	<pre>d* sendbuf, const int sendcounts[], , const MPI_Datatype sendtypes[], t int recvcounts[], const int rdispls[], recvtypes[], MPI_Comm comm, MPI_Info info, t)</pre>	39 40 41 42 43	
F08 bind	0		44 45	
MPI_Allt		counts, sdispls, sendtypes, recvbuf,	46	
TYDE(*) DIMENSION() INTENT(IN) ASYNCHDONOUS ·· condbuf			47 48	

```
1
          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
\mathbf{2}
          INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
3
                     recvcounts(*), rdispls(*)
4
          TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
5
                     recvtypes(*)
6
          TYPE(MPI_Comm), INTENT(IN) :: comm
7
          TYPE(MPI_Info), INTENT(IN) ::
                                             info
8
          TYPE(MPI_Request), INTENT(OUT) :: request
9
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     F binding
11
     MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
12
                    RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)
13
          <type> SENDBUF(*), RECVBUF(*)
14
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
15
                     RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR
16
17
          Creates a persistent collective communication request for the alltoally operation.
18
19
     5.13.7 Persistent Reduce
20
21
22
     MPI_REDUCE_INIT(sendbuf, recvbuf, count, datatype, op, root, comm, info, request)
23
       IN
                 sendbuf
                                            address of send buffer (choice)
^{24}
25
       OUT
                 recvbuf
                                             address of receive buffer (choice, significant only at
26
                                            root)
27
                                            number of elements in send buffer (non-negative inte-
       IN
                 count
28
                                            ger)
29
       IN
                 datatype
                                            data type of elements of send buffer (handle)
30
^{31}
       IN
                 ор
                                            reduce operation (handle)
32
       IN
                                            rank of root process (integer)
                 root
33
       IN
                                            communicator (handle)
                 comm
34
35
       IN
                 info
                                            info argument (handle)
36
       OUT
                 request
                                             communication request (handle)
37
38
     C binding
39
     int MPI_Reduce_init(const void* sendbuf, void* recvbuf, int count,
40
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
41
                    MPI_Info info, MPI_Request *request)
42
43
     F08 binding
^{44}
     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
45
                    request, ierror)
46
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
47
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
48
          INTEGER, INTENT(IN) :: count, root
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        2
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        3
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                        4
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                        5
                                                                                        6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        7
F binding
MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO,
                                                                                        9
               REQUEST, IERROR)
                                                                                        10
    <type> SENDBUF(*), RECVBUF(*)
                                                                                        11
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR
                                                                                        12
                                                                                        13
    Creates a persistent collective communication request for the reduce operation.
                                                                                        14
                                                                                        15
5.13.8 Persistent All-Reduce
                                                                                        16
                                                                                        17
                                                                                        18
MPI_ALLREDUCE_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)
                                                                                        19
  IN
           sendbuf
                                       starting address of send buffer (choice)
                                                                                        20
                                                                                        21
  OUT
           recvbuf
                                       starting address of receive buffer (choice)
                                                                                        22
  IN
           count
                                       number of elements in send buffer (non-negative inte-
                                                                                        23
                                       ger)
                                                                                        ^{24}
  IN
                                       data type of elements of send buffer (handle)
           datatype
                                                                                        25
                                                                                        26
  IN
                                       operation (handle)
           ор
                                                                                        27
  IN
           comm
                                       communicator (handle)
                                                                                        28
  IN
           info
                                       info argument (handle)
                                                                                        29
                                                                                        30
  OUT
           request
                                       communication request (handle)
                                                                                        31
                                                                                        32
C binding
                                                                                        33
int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count,
                                                                                        34
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                                                                                        35
               MPI_Info info, MPI_Request *request)
                                                                                        36
F08 binding
                                                                                        37
MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
                                                                                        38
               request, ierror)
                                                                                        39
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                        40
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                        41
    INTEGER, INTENT(IN) :: count
                                                                                        42
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        43
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                        44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                        45
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                        46
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                        47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        48
```

Unofficial Draft for Comment Only

1	T 1 · 1 ·		
2	F binding		
3	MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR)		
4	<t.vr< td=""><td>e> SENDBUF(*), REC</td><td></td></t.vr<>	e> SENDBUF(*), REC	
5	01		PE, OP, COMM, INFO, REQUEST, IERROR
6			
7	Crea	tes a persistent collect	tive communication request for the all reduce operation.
8	5.13.9 Persistent Reduce-Scatter with Equal Blocks		
9 10	J.1J.9 I	ersistent Neutre-Sca	
10			
12 13	MPI_RED	DUCE_SCATTER_BLO info, request)	CK_INIT(sendbuf, recvbuf, recvcount, datatype, op, comm,
14	IN	sendbuf	starting address of send buffer (choice)
15 16	OUT	recvbuf	starting address of receive buffer (choice)
17	IN	recvcount	element count per block (non-negative integer)
18	IN	datatype	data type of elements of send and receive buffers (han-
19 20		ddddype	dle)
21	IN	ор	operation (handle)
22	IN	comm	communicator (handle)
23	IN	info	info argument (handle)
24 25	OUT	request	communication request (handle)
26			
27	C bindin	0	ob init (const wordt sondbuf wordt noorbuf
28 29	IIIC MFI_		<pre>ock_init(const void* sendbuf, void* recvbuf, t, MPI_Datatype datatype, MPI_Op op,</pre>
30			n, MPI_Info info, MPI_Request *request)
31	Dec 1		_,,
32	F08 bine	0	nit(sendbuf, recvbuf, recvcount, datatype, op,
33	hr I_heau		request, ierror)
34	TYPE		, INTENT(IN), ASYNCHRONOUS :: sendbuf
35	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf		
36 37	INTEGER, INTENT(IN) :: recvcount		
38	TYPE(MPI_Datatype), INTENT(IN) :: datatype		
39	TYPE(MPI_Op), INTENT(IN) :: op		
40	TYPE(MPI_Comm), INTENT(IN) :: comm		
41	TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request		
42		-	ENT(OUT) :: ierror
43			
44	F bindir	•	
45 46	NFI_KEDU		NIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, REQUEST, IERROR)
47	<tvr< td=""><td><pre>> SENDBUF(*), REC</pre></td><td></td></tvr<>	<pre>> SENDBUF(*), REC</pre>	
48	01	-	ATYPE, OP, COMM, INFO, REQUEST, IERROR

	ates a persistent collect peration.	ive communication request for the reduce-scatter with equal	1 2
5.13.10	Persistent Reduce-Sc	atter	3 4 5 6
MPI_RE	DUCE_SCATTER_INIT quest)	(sendbuf, recvbuf, recvcounts, datatype, op, comm, info, re-	7 8 9
IN	sendbuf	starting address of send buffer (choice)	10
OUT	recvbuf	starting address of receive buffer (choice)	11
IN	recvcounts	non-negative integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.	12 13 14 15
IN	datatype	data type of elements of input buffer (handle)	16
IN	ор	operation (handle)	17
IN	comm	communicator (handle)	18 19
IN	info	info argument (handle)	20
OUT	request	communication request (handle)	21
IIIC III I	const int rec	t(const void* sendbuf, void* recvbuf, vcounts[], MPI_Datatype datatype, MPI_Op op, n, MPI_Info info, MPI_Request *request)	25 26 27
TYP TYP INT TYP TYP TYP TYP	uce_scatter_init(se info, request E(*), DIMENSION() E(*), DIMENSION() EGER, INTENT(IN), A	<pre>, INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf SYNCHRONOUS :: recvcounts(*) TENT(IN) :: datatype N) :: op (IN) :: comm (IN) :: comm (IN) :: info ENT(OUT) :: request</pre>	28 29 30 31 32 33 34 35 36 37 38 39
F bindi MPI_RED <ty INT</ty 	ng UCE_SCATTER_INIT(SE INFO, REQUEST pe> SENDBUF(*), REC EGER RECVCOUNTS(*),	NDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, ', IERROR)	40 41 42 43 44 45 46 47
			41

```
240
                                         CHAPTER 5. COLLECTIVE COMMUNICATION
1
     5.13.11 Persistent Inclusive Scan
\mathbf{2}
3
4
     MPI_SCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)
5
       IN
                 sendbuf
                                             starting address of send buffer (choice)
6
       OUT
7
                 recvbuf
                                             starting address of receive buffer (choice)
8
       IN
                                             number of elements in input buffer (non-negative in-
                 count
9
                                             teger)
10
       IN
                                             data type of elements of input buffer (handle)
                 datatype
11
       IN
                                             operation (handle)
12
                 ор
13
       IN
                 comm
                                             communicator (handle)
14
       IN
                 info
                                             info argument (handle)
15
16
       OUT
                 request
                                             communication request (handle)
17
18
     C binding
19
     int MPI_Scan_init(const void* sendbuf, void* recvbuf, int count,
20
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
21
                    MPI_Info info, MPI_Request *request)
22
     F08 binding
23
     MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
24
                    ierror)
25
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
          INTEGER, INTENT(IN) :: count
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          TYPE(MPI_Op), INTENT(IN) :: op
30
          TYPE(MPI_Comm), INTENT(IN) :: comm
^{31}
          TYPE(MPI_Info), INTENT(IN) ::
                                             info
32
          TYPE(MPI_Request), INTENT(OUT) :: request
33
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
34
35
     F binding
36
     MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
37
                    IERROR)
38
          <type> SENDBUF(*), RECVBUF(*)
39
          INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
40
          Creates a persistent collective communication request for the inclusive scan operation.
41
42
43
44
45
46
47
48
```

5.13.12 Persistent Exclusive Scan

5.13.12	Persistent Exclusive :	Scan
MPI_EX	SCAN_INIT(sendbuf, re	cvbuf, count, datatype, op, comm, info, request)
IN sendbuf starting address of send buffer (choice)		
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative in-teger)
IN	datatype	data type of elements of input buffer (handle)
IN	ор	operation (handle)
IN	comm	intracommunicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)
-	,	
C bindi	ng	
int MPI		void* sendbuf, void* recvbuf, int count,
	• -	datatype, MPI_Op op, MPI_Comm comm,
	MPI_Into info	o, MPI_Request *request)
F08 bin	0	
MPI_Exs		recvbuf, count, datatype, op, comm, info, request,
מעיד	ierror)	, INTENT(IN), ASYNCHRONOUS :: sendbuf
		, INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf
	EGER, INTENT(IN) ::	
TYP	E(MPI_Datatype), IN	TENT(IN) :: datatype
	E(MPI_Op), INTENT(I	÷
	E(MPI_Comm), INTENT	
	E(MPI_Info), INTENT	
	E(MPI_Request), INT EGER, OPTIONAL, INT	`ENT(OUT) :: request `ENT(OUT) :: ierror
		ENI(UUI) TEIIUI
F bindi	-	
MPI_EXS		ECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
/+	IERROR)	
	pe> SENDBUF(*), REC EGER COUNT, DATATYP	PE, OP, COMM, INFO, REQUEST, IERROR
	-	
Crea	ates a persistent collect	tive communication request for the exclusive scan operation.
	_	
5.14	Correctness	
A correct	t portable program my	st invoke collective communications so that deadlock will not
		unications are synchronizing or not. The following examples

A correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not. The following examples illustrate dangerous use of collective routines on intracommunicators. 47
48

Unofficial Draft for Comment Only

```
1
     Example 5.25
\mathbf{2}
          The following is erroneous.
3
4
     switch(rank) {
          case 0:
5
6
              MPI_Bcast(buf1, count, type, 0, comm);
              MPI_Bcast(buf2, count, type, 1, comm);
7
8
              break;
9
          case 1:
              MPI_Bcast(buf2, count, type, 1, comm);
10
              MPI_Bcast(buf1, count, type, 0, comm);
11
              break;
12
     }
13
14
          We assume that the group of comm is \{0,1\}. Two processes execute two broadcast
15
     operations in reverse order. If the operation is synchronizing then a deadlock will occur.
16
          Collective operations must be executed in the same order at all members of the com-
17
     munication group.
18
19
     Example 5.26
20
          The following is erroneous.
21
22
     switch(rank) {
23
          case 0:
24
              MPI_Bcast(buf1, count, type, 0, comm0);
25
              MPI_Bcast(buf2, count, type, 2, comm2);
26
              break:
27
          case 1:
28
              MPI_Bcast(buf1, count, type, 1, comm1);
29
              MPI_Bcast(buf2, count, type, 0, comm0);
30
              break;
31
          case 2:
32
              MPI_Bcast(buf1, count, type, 2, comm2);
33
              MPI_Bcast(buf2, count, type, 1, comm1);
34
              break;
35
     }
36
37
          Assume that the group of comm0 is \{0,1\}, of comm1 is \{1, 2\} and of comm2 is \{2,0\}. If
38
     the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast
39
     in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes
40
     only after the broadcast in comm1; and the broadcast in comm1 completes only after the
41
     broadcast in comm2. Thus, the code will deadlock.
42
```

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

```
<sup>45</sup> Example 5.27
```

The following is erroneous.

```
47
48
```

46

43

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

Process zero executes a broadcast, followed by a blocking send operation. Process one first executes a blocking receive that matches the send, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process zero *may* block until process one executes the matching broadcast call, so that the send is not executed. Process one will definitely block on the receive and so, in this case, never executes the broadcast.

The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

Example 5.28

An unsafe, non-deterministic program.

```
^{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
}
```

All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

Two possible executions of this program, with different matchings of sends and receives, ⁴³ are illustrated in Figure 5.12. Note that the second execution has the peculiar effect that ⁴⁴ a send executed after the broadcast is received at another node before the broadcast. This ⁴⁵ example illustrates the fact that one should not rely on collective communication functions ⁴⁶ to have particular synchronization effects. A program that works correctly only when the ⁴⁷ first execution occurs (only when broadcast is synchronizing) is erroneous. ⁴⁸

Unofficial Draft for Comment Only

1

2

3

45

6

7

9

10 11

12

13

14

15

16

17

18

19

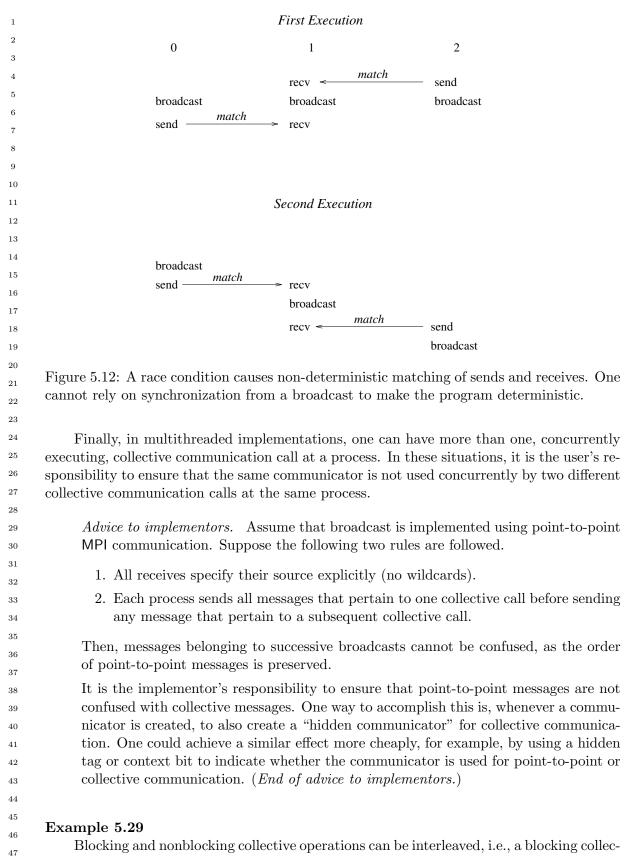
20 21

22

23

39 40

41



MPI_	Request	req;
------	---------	------

MPI_Ibarrier(comm, &req); MPI_Bcast(buf1, count, type, 0, comm); MPI_Wait(&req, MPI_STATUS_IGNORE);

Each process starts a nonblocking barrier operation, participates in a blocking broadcast and then waits until every other process started the barrier operation. This effectively turns the broadcast into a synchronizing broadcast with possible communication/communication overlap (MPI_Bcast is allowed, but not required to synchronize).

Example 5.30

The starting order of collective operations on a particular communicator defines their matching. The following example shows an erroneous matching of different collective operations on the same communicator.

```
MPI_Request req;
switch(rank) {
    case 0:
        /* erroneous matching */
        MPI_Ibarrier(comm, &req);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
    case 1:
        /* erroneous matching */
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Ibarrier(comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
```

```
}
```

This ordering would match MPI_lbarrier on rank 0 with MPI_Bcast on rank 1 which is erroneous and the program behavior is undefined. However, if such an order is required, the user must create different duplicate communicators and perform the operations on them. If started with two processes, the following program would be correct:

```
37
MPI_Request req;
                                                                                       38
MPI_Comm dupcomm;
                                                                                       39
MPI_Comm_dup(comm, &dupcomm);
switch(rank) {
                                                                                       40
                                                                                       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);
```

1 $\mathbf{2}$ 3

4

56

7

8

9

10 11

12

13

14

1516

17

18

19

20

21

22

23

24

25

26

27

28

29

30

3132

33

34

35

1	<pre>MPI_Wait(&req, MPI_STATUS_IGNORE);</pre>
2	break;
3	}
4	Advice to users. The use of different communicators offers some flexibility regarding
5	the matching of nonblocking collective operations. In this sense, communicators could
6	be used as an equivalent to tags. However, communicator construction might induce
7	overheads so that this should be used carefully. (End of advice to users.)
8	
9	Example 5.31
10	Nonblocking collective operations can rely on the same progression rules as nonblocking
11	point-to-point messages. Thus, if started with two processes, the following program is a
12	valid MPI program and is guaranteed to terminate:
13	vand wit't program and is guaranteed to terminate.
14	MPI_Request req;
15 16	
17	<pre>switch(rank) {</pre>
18	case 0:
19	<pre>MPI_Ibarrier(comm, &req);</pre>
20	<pre>MPI_Wait(&req, MPI_STATUS_IGNORE);</pre>
21	<pre>MPI_Send(buf, count, dtype, 1, tag, comm);</pre>
22	break;
23	case 1:
24	MPI_Ibarrier(comm, &req);
25	MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
26	<pre>MPI_Wait(&req, MPI_STATUS_IGNORE); break;</pre>
27	}
28	
29	The MPI library must progress the barrier in the MPI_Recv call. Thus, the MPI_Wait
30	call in rank 0 will eventually complete, which enables the matching MPI_Send so all calls
31	eventually return.
32	Example 5.32
33	Blocking and nonblocking collective operations do not match. The following example
34	is erroneous.
35	
36	MPI_Request req;
37	
38 39	<pre>switch(rank) {</pre>
40	case 0:
41	/* erroneous false matching of Alltoall and Ialltoall */
42	<pre>MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req); MPI_Wait(&req, MPI_STATUS_IGNORE);</pre>
43	break;
44	case 1:
45	/* erroneous false matching of Alltoall and Ialltoall */
46	MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
47	break;
48	}

Example 5.33

Collective and point-to-point requests can be mixed in functions that enable multiple completions. If started with two processes, the following program is valid.

```
MPI_Request reqs[2];
switch(rank) {
    case 0:
      MPI_Ibarrier(comm, &reqs[0]);
      MPI_Send(buf, count, dtype, 1, tag, comm);
      MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
      break;
    case 1:
      MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
      MPI_Ibarrier(comm, &reqs[1]);
      MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
      break;
```

}

The MPI_Waitall call returns only after the barrier and the receive completed.

Example 5.34

Multiple nonblocking collective operations can be outstanding on a single communicator and match in order.

```
MPI_Request reqs[3];
```

```
compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
compute(buf3);
MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
```

Advice to users. Pipelining and double-buffering techniques can efficiently be used to overlap computation and communication. However, having too many outstanding requests might have a negative impact on performance. (End of advice to users.)

Advice to implementors. The use of pipelining may generate many outstanding requests. A high-quality hardware-supported implementation with limited resources should be able to fall back to a software implementation if its resources are exhausted. In this way, the implementation could limit the number of outstanding requests only by the available memory. (End of advice to implementors.)

Example 5.35

1 2

3

4

56

7

8

9

10

11

12

13

14

15

16

17

18 19

2021

22

23

2425

2627

28

29

30

31

32

33

34 35

36

37

38 39

40

41

42

43

44 4546

CHAPTER 5. COLLECTIVE COMMUNICATION

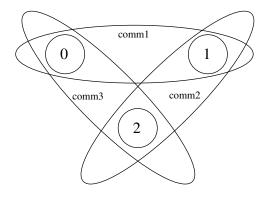


Figure 5.13: Example with overlapping communicators.

¹⁴ Nonblocking collective operations can also be used to enable simultaneous collective ¹⁵ operations on multiple overlapping communicators (see Figure 5.13). The following example ¹⁶ is started with three processes and three communicators. The first communicator comm1 ¹⁷ includes ranks 0 and 1, comm2 includes ranks 1 and 2, and comm3 spans ranks 0 and 2. It is ¹⁸ not possible to perform a blocking collective operation on all communicators because there ¹⁹ exists no deadlock-free order to invoke them. However, nonblocking collective operations ²⁰ can easily be used to achieve this task.

```
<sup>22</sup> MPI_Request reqs[2];
```

```
^{24}
     switch(rank) {
25
         case 0:
26
           MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
27
           MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
28
           break;
29
         case 1:
30
           MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
31
           MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[1]);
32
           break;
33
         case 2:
34
           MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[0]);
35
           MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
36
           break;
37
     }
38
     MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
39
          Advice to users. This method can be useful if overlapping neighboring regions (halo
40
          or ghost zones) are used in collective operations. The sequence of the two calls in
41
          each process is irrelevant because the two nonblocking operations are performed on
42
          different communicators. (End of advice to users.)
43
44
45
```

Example 5.36

46

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

1 2

3

4 5

> 6 7

12 13

5.14. CORRECTNESS

```
1
MPI_Request reqs[2];
                                                                                              \mathbf{2}
                                                                                               3
compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
                                                                                              4
compute(buf2);
                                                                                              5
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
                                                                                               6
MPI_Wait(&reqs[1], MPI_STATUS_IGNORE);
                                                                                               7
/* nothing is known about the status of the first bcast here */
                                                                                               8
                                                                                              9
MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
                                                                                              10
    Finishing the second MPI_IBCAST is completely independent of the first one. This
                                                                                              11
means that it is not guaranteed that the first broadcast operation is finished or even started
                                                                                              12
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
```

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

6.1.1 Features Needed to Support Libraries

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

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

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

 24

6.1.2 MPI's Support for Libraries

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

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

¹³ ¹⁴ **Communicators** (see [21, 52, 56]) encapsulate all of these ideas in order to provide the ¹⁵ appropriate scope for all communication operations in MPI. Communicators are divided ¹⁶ into two kinds: intra-communicators for operations within a single group of processes and ¹⁷ inter-communicators for operations between two groups of processes.

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

24

1

2

3

4 5

6

7 8

9 10

11

12

18

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

31

36

37

38

39

40

41

42

43

44

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

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

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

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

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

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

6.2.2 Contexts

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

 $41 \\ 42$

43

44

45

46

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

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

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

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

6.2.3 Intra-Communicators

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

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

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

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

6.2.4 Predefined Intra-Communicators

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

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

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

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

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11 12

13 14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28 29

30

31

32 33

34

35

MPI_COMM_GROUP (see below). MPI does not specify the correspondence between the
 process rank in MPI_COMM_WORLD and its (machine-dependent) absolute address. Neither
 does MPI specify the function of the host process, if any. Other implementation-dependent,
 predefined communicators may also be provided.

5 6 7

8

9

10 11

6.3 Group Management

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

```
6.3.1 Group Accessors
12
13
14
     MPI_GROUP_SIZE(group, size)
15
16
       IN
                                             group (handle)
                 group
17
       OUT
                 size
                                             number of processes in the group (non-negative inte-
18
                                             ger)
19
20
     C binding
21
     int MPI_Group_size(MPI_Group group, int *size)
22
23
     F08 binding
^{24}
     MPI_Group_size(group, size, ierror)
25
          TYPE(MPI_Group), INTENT(IN) :: group
26
          INTEGER, INTENT(OUT) :: size
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     F binding
29
     MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
30
          INTEGER GROUP, SIZE, IERROR
^{31}
32
33
34
     MPI_GROUP_RANK(group, rank)
35
       IN
                                             group (handle)
                 group
36
       OUT
                 rank
                                             rank of the calling process in group, or
37
                                             MPI_UNDEFINED if the process is not a member (non-
38
                                             negative integer)
39
40
     C binding
41
     int MPI_Group_rank(MPI_Group group, int *rank)
42
43
     F08 binding
44
     MPI_Group_rank(group, rank, ierror)
45
          TYPE(MPI_Group), INTENT(IN) :: group
46
          INTEGER, INTENT(OUT) :: rank
47
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

F binding

F DINAIR MPI_GROU	'S IP_RANK(GROUP, RAN	K, IERROR)
	GER GROUP, RANK,	
MPI_GRO	DUP_TRANSLATE_R	ANKS(group1, n, ranks1, group2, ranks2)
IN	group1	group1 (handle)
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)
IN	ranks1	array of zero or more valid ranks in group1 (array of
		non-negative integers)
IN	group2	group2 (handle)
OUT	ranks2	array of corresponding ranks in group2,
		MPI_UNDEFINED when no correspondence exists. (ar-
		ray of non-negative integers)
~		
C bindi	0	anka (MDI Group group1 int a const int park-15]
IIIC MPI_	-	<pre>manks(MPI_Group group1, int n, const int ranks1[], coup2, int ranks2[])</pre>
		<u>r</u> -,
F08 bine	0	(group1 n ronkg1 group) ronkg2 iorron)
	-	(group1, n, ranks1, group2, ranks2, ierror) NT(IN) :: group1, group2
	GER, INTENT(IN) :	
INTE	GER, INTENT(OUT)	:: ranks2(n)
INTE	GER, OPTIONAL, IN	TENT(OUT) :: ierror
F bindir	ıg	
		(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
INTE	CGER GROUP1, N, RA	NKS1(*), GROUP2, RANKS2(*), IERROR
This	function is important	t for determining the relative numbering of the same processes
	° *	stance, if one knows the ranks of certain processes in the group
	,	ight want to know their ranks in a subset of that group.
	PROC_NULL is a valid IPI_PROC_NULL as th	l rank for input to MPI_GROUP_TRANSLATE_RANKS, which be translated rank
	II I NOC_NOLL as th	
	OUP_COMPARE(grou	p1, group2, result)
IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	result	result (integer)
C bindi	•	
int MPI_	Group_compare(MPI	_Group group1, MPI_Group group2, int *result)
F08 bin	ding	
MDT Crow	n compare(group1	group2, result, ierror)

Unofficial Draft for Comment Only

```
1
          TYPE(MPI_Group), INTENT(IN) :: group1, group2
2
          INTEGER, INTENT(OUT) :: result
3
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
4
     F binding
5
     MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
6
          INTEGER GROUP1, GROUP2, RESULT, IERROR
7
8
     MPI_IDENT results if the group members and group order is exactly the same in both groups.
9
     This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if
10
     the group members are the same but the order is different. MPI_UNEQUAL results otherwise.
11
12
     6.3.2 Group Constructors
13
     Group constructors are used to subset and superset existing groups. These constructors
14
     construct new groups from existing groups. These are local operations, and distinct groups
15
     may be defined on different processes; a process may also define a group that does not
16
     include itself. Consistent definitions are required when groups are used as arguments in
17
     communicator-building functions. MPI does not provide a mechanism to build a group
18
     from scratch, but only from other, previously defined groups. The base group, upon which
19
     all other groups are defined, is the group associated with the initial communicator
20
     MPI_COMM_WORLD (accessible through the function MPI_COMM_GROUP).
21
22
                        In what follows, there is no group duplication function analogous to
           Rationale.
23
           MPI_COMM_DUP, defined later in this chapter. There is no need for a group dupli-
24
           cator. A group, once created, can have several references to it by making copies of
25
           the handle. The following constructors address the need for subsets and supersets of
26
           existing groups. (End of rationale.)
27
28
           Advice to implementors.
                                     Each group constructor behaves as if it returned a new
29
           group object. When this new group is a copy of an existing group, then one can
30
           avoid creating such new objects, using a reference-count mechanism. (End of advice
31
           to implementors.)
32
33
34
35
     MPI_COMM_GROUP(comm, group)
36
       IN
                 comm
                                             communicator (handle)
37
       OUT
                                             group corresponding to comm (handle)
                 group
38
39
40
     C binding
     int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
41
42
     F08 binding
43
     MPI_Comm_group(comm, group, ierror)
44
          TYPE(MPI_Comm), INTENT(IN) :: comm
45
          TYPE(MPI_Group), INTENT(OUT) ::
                                               group
46
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
     F binding
```

MPI_COMM_GROUP(COMM, GROUP, IERROR)			1
INTEGER COMM, GROUP, IERROR			2 3
MPI_COMM_GROUP returns in group a handle to the group of comm.			4
			5
MPL CR	OUP_UNION(group1, g		6
		,	7
IN	group1	first group (handle)	8
IN	group2	second group (handle)	9
OUT	newgroup	union group (handle)	10
			11
C bindi	ng		12 13
int MPI	_Group_union(MPI_Gr	oup group1, MPI_Group group2,	13
	MPI_Group *ne	wgroup)	15
F08 bin	ding		16
		oup2, newgroup, ierror)	17
TYPI	E(MPI_Group), INTEN	T(IN) :: group1, group2	18
	•	T(OUT) :: newgroup	19
INT	EGER, OPTIONAL, INT	ENT(OUT) :: ierror	20
F bindi	ng		21 22
MPI_GRO	UP_UNION(GROUP1, GR	OUP2, NEWGROUP, IERROR)	22
INT	EGER GROUP1, GROUP2	, NEWGROUP, IERROR	24
			25
			26
MPI_GR	OUP_INTERSECTION(group1, group2, newgroup)	27
IN	group1	first group (handle)	28
IN	group2	second group (handle)	29
OUT	newgroup	intersection group (handle)	30 31
001	newgroup	intersection group (nandie)	31
C bindi	ng		33
		(MPI_Group group1, MPI_Group group2,	34
	 MPI_Group *ne		35
FOR him	ding		36
F08 bin		up1, group2, newgroup, ierror)	37
		T(IN) :: group1, group2	38
	-	T(OUT) :: newgroup	39
	-	ENT(OUT) :: ierror	40 41
F bindi	nœ		42
	0	UP1, GROUP2, NEWGROUP, IERROR)	43
	EGER GROUP1, GROUP2		44
	· · · · · · · · · · · · · · · · · · ·		45
			46
			47

```
1
     MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
\mathbf{2}
       IN
                  group1
                                               first group (handle)
3
       IN
                 group2
                                               second group (handle)
4
5
       OUT
                  newgroup
                                               difference group (handle)
6
7
      C binding
8
      int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
9
                     MPI_Group *newgroup)
10
     F08 binding
11
     MPI_Group_difference(group1, group2, newgroup, ierror)
12
          TYPE(MPI_Group), INTENT(IN) :: group1, group2
13
          TYPE(MPI_Group), INTENT(OUT) :: newgroup
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     F binding
17
     MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
18
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
19
      The set-like operations are defined as follows:
20
21
      union All elements of the first group (group1), followed by all elements of second group
22
           (group2) not in the first group.
23
^{24}
     intersect all elements of the first group that are also in the second group, ordered as in
25
           the first group.
26
      difference all elements of the first group that are not in the second group, ordered as in
27
           the first group.
28
29
      Note that for these operations the order of processes in the output group is determined
30
      primarily by order in the first group (if possible) and then, if necessary, by order in the
31
      second group. Neither union nor intersection are commutative, but both are associative.
32
          The new group can be empty, that is, equal to MPI_GROUP_EMPTY.
33
34
35
      MPI_GROUP_INCL(group, n, ranks, newgroup)
36
       IN
                                               group (handle)
                 group
37
       IN
                                               number of elements in array ranks (and size of
38
                  n
                                               newgroup) (integer)
39
40
       IN
                  ranks
                                               ranks of processes in group to appear in
41
                                               newgroup (array of non-negative integers)
42
       OUT
                                               new group derived from above, in the order defined by
                  newgroup
43
                                               ranks (handle)
44
45
     C binding
46
      int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
47
                     MPI_Group *newgroup)
48
```

F08 bind	ing		1
<pre>MPI_Group_incl(group, n, ranks, newgroup, ierror)</pre>			
TYPE(MPI_Group), INTENT(IN) :: group			
	GER, INTENT(IN) :: n, ra		4
	(MPI_Group), INTENT(OUT)		5 6
INTE(GER, OPTIONAL, INTENT(OUT) :: ierror	6 7
F bindin	8		8
MPI_GROU	P_INCL(GROUP, N, RANKS, N	IEWGROUP, IERROR)	9
INTE	GER GROUP, N, RANKS(*), N	IEWGROUP, IERROR	10
The f	unction MPL GROUP INCL	reates a group newgroup that consists of the	11
		,, ranks[n-1]; the process with rank i in newgroup	12
-		p. Each of the n elements of ranks must be a valid	13
-		distinct, or else the program is erroneous. If $n = 0$,	14
-	•	This function can, for instance, be used to reorder	15
the element	nts of a group. See also MPI_	GROUP_COMPARE.	16
			17
			18
MPI_GRU	UP_EXCL(group, n, ranks, nev	,	19
IN	group	group (handle)	20
IN	n	number of elements in array ranks (integer)	21 22
IN	ranks	ranks of processes in group not to appear in newgroup	23
		(array of non-negative integers)	24
OUT	newgroup	new group derived from above, preserving the order	25
		defined by group (handle)	26
			27
C bindin	g		28
int MPI_(Group_excl(MPI_Group grou	p, int n, const int ranks[],	29
	MPI_Group *newgroup)		30 31
F08 bind	ina		32
	p_excl(group, n, ranks, n	lewgroup, jerror)	33
-	(MPI_Group), INTENT(IN) :		34
	GER, INTENT(IN) :: n, ra	5 I	35
	(MPI_Group), INTENT(OUT)		36
	GER, OPTIONAL, INTENT(OUT		37
F hindin			38
F bindin	g P_EXCL(GROUP, N, RANKS, N		39
	GER GROUP, N, RANKS, N		40
			41
		reates a group of processes newgroup that is obtained	42
-		with ranks ranks[0] , ranks[n-1]. The ordering of	43
processes	in newgroup is identical to the	e ordering in group. Each of the n elements of ranks	44

must be a valid rank in group and all elements must be distinct; otherwise, the program is 45erroneous. If n = 0, then newgroup is identical to group. 46

Unofficial Draft for Comment Only

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

```
1
      MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
2
        IN
                                                  group (handle)
                   group
3
        IN
                                                  number of triplets in array ranges (integer)
                   n
4
5
        IN
                                                  a one-dimensional array of integer triplets, of the form
                   ranges
6
                                                  (first rank, last rank, stride) indicating ranks in group
7
                                                  of processes to be included in newgroup (array of non-
8
                                                  negative integers)
9
        OUT
                                                  new group derived from above, in the order defined by
                   newgroup
10
                                                  ranges (handle)
11
12
      C binding
13
      int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
14
                       MPI_Group *newgroup)
15
16
      F08 binding
17
      MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
18
           TYPE(MPI_Group), INTENT(IN) :: group
19
           INTEGER, INTENT(IN) :: n, ranges(3,n)
20
           TYPE(MPI_Group), INTENT(OUT) :: newgroup
21
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
      F binding
23
      MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
24
           INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
25
26
      If ranges consists of the triplets
27
            (first_1, last_1, stride_1), \ldots, (first_n, last_n, stride_n)
28
29
      then newgroup consists of the sequence of processes in group with ranks
30
            first_1, first_1 + stride_1, \dots, first_1 + \left| \frac{last_1 - first_1}{stride_1} \right| stride_1, \dots,
^{31}
32
33
            first_n, first_n + stride_n, \dots, first_n + \left| \frac{last_n - first_n}{stride_n} \right| stride_n.
34
35
36
           Each computed rank must be a valid rank in group and all computed ranks must be
37
      distinct, or else the program is erroneous. Note that we may have first_i > last_i, and stride_i
38
      may be negative, but cannot be zero.
39
           The functionality of this routine is specified to be equivalent to expanding the array
40
      of ranges to an array of the included ranks and passing the resulting array of ranks and
41
      other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call
42
      to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the
43
      argument ranges.
44
45
46
47
48
```

MPI_GRO	UP_RANGE_EXCL(group, n, ra	anges, newgroup)	1
IN	group	group (handle)	2
IN	n	number of triplets in array ranges (integer)	3 4
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group	5
		of processes to be excluded from the output group newgroup (array of non-negative integers)	7 8
OUT	newgroup	new group derived from above, preserving the order in $group$ (handle)	9 10 11
C hindin	~		12

Ν

C binding

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
             MPI_Group *newgroup)
```

F08 binding

F binding

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

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

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

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

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

13

14

1516

17

23

 24

2526

27

28

29

30

 31

32

33 34

35

36

37

38

39

40

```
1
     6.3.3
            Group Destructors
\mathbf{2}
3
4
     MPI_GROUP_FREE(group)
5
       INOUT
                 group
                                              group (handle)
6
7
     C binding
8
     int MPI_Group_free(MPI_Group *group)
9
10
     F08 binding
11
     MPI_Group_free(group, ierror)
12
          TYPE(MPI_Group), INTENT(INOUT) ::
                                                  group
13
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
14
     F binding
15
     MPI_GROUP_FREE(GROUP, IERROR)
16
          INTEGER GROUP, IERROR
17
18
          This operation marks a group object for deallocation. The handle group is set to
19
     MPI_GROUP_NULL by the call. Any on-going operation using this group will complete
20
     normally.
21
22
           Advice to implementors. One can keep a reference count that is incremented for each
23
           call to MPI_COMM_GROUP, MPI_COMM_CREATE, MPI_COMM_DUP, and
^{24}
           MPI_COMM_IDUP, and decremented for each call to MPI_GROUP_FREE or
25
           MPI_COMM_FREE; the group object is ultimately deallocated when the reference
26
           count drops to zero. (End of advice to implementors.)
27
28
     6.4
            Communicator Management
29
30
     This section describes the manipulation of communicators in MPI. Operations that access
^{31}
     communicators are local and their execution does not require interprocess communication.
32
     Operations that create communicators are collective and may require interprocess commu-
33
     nication.
34
35
           Advice to implementors.
                                      High-quality implementations should amortize the over-
36
           heads associated with the creation of communicators (for the same group, or subsets
37
           thereof) over several calls, by allocating multiple contexts with one collective commu-
38
           nication. (End of advice to implementors.)
39
40
     6.4.1 Communicator Accessors
41
42
     The following are all local operations.
43
44
45
46
47
48
```

		:)		1
	COMM_SIZE(comm, s	,		2
IN	comm	co	ommunicator (handle)	3
OUT	size		umber of processes in the group of	4
		CC	omm (non-negative integer)	5
				6
C bir				7
int M	PI_Comm_size(MPI_(Comm comm, int	*size)	8
F08 h	oinding			9 10
MPI_C	omm_size(comm, siz	ze, ierror)		10
	YPE(MPI_Comm), INT		omm	12
	NTEGER, INTENT(OUT			13
I	NTEGER, OPTIONAL,	INTENT(OUT) :	: ierror	14
F bin	ding			15
MPI_C	OMM_SIZE(COMM, SIZ	ZE, IERROR)		16
I	NTEGER COMM, SIZE	IERROR		17
				18
	Rationale. This fun	ction is equivale	nt to accessing the communicator's group with	19
	MPI_COMM_GROUP	(see above), con	mputing the size using MPI_GROUP_SIZE, and	20
			MPI_GROUP_FREE. However, this function is	21
1	so commonly used th	at this shortcut	was introduced. (End of rationale.)	22 23
				23
			dicates the number of processes involved in a	25
	communicator. For MPI_COMM_WORLD, it indicates the total number of processes			26
		-	esses has been changed by using the functions	27
	-	,	he number of processes in MPI_COMM_WORLD	28
	does not change duri	0		29
			call to determine the amount of concurrency	30
	-	° *	gram. The following call, MPI_COMM_RANK	31
			calls it in the range from $0 \dots$ size -1 , where size ZE.(<i>End of advice to users.</i>)	32
	is the feturit value of		ZE :(<i>Ena of auvice to users.</i>)	33
				34
				35
MPI_0	COMM_RANK(comm,	rank)		36 37
IN	comm	СС	ommunicator (handle)	38
OUT	- rank	ra	ank of the calling process in group	39
			omm (non-negative integer)	40
				41
C bir	ding			42
	PI_Comm_rank(MPI_(Comm comm, int	*rank)	43
		-		44
	pinding			45
	omm_rank(comm, ran YPE(MPI_Comm), INT		omm	46
	NTEGER, INTENT(OUT		omm	47
1		·/ ·· I allA		48

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
     F binding
3
     MPI_COMM_RANK(COMM, RANK, IERROR)
4
          INTEGER COMM, RANK, IERROR
5
6
           Rationale. This function is equivalent to accessing the communicator's group with
7
           MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK,
8
           and then freeing the temporary group via MPI_GROUP_FREE. However, this function
9
           is so commonly used that this shortcut was introduced. (End of rationale.)
10
11
           Advice to users. This function gives the rank of the process in the particular commu-
12
           nicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.
13
           Many programs will be written with the master-slave model, where one process (such
14
           as the rank-zero process) will play a supervisory role, and the other processes will
15
           serve as compute nodes. In this framework, the two preceding calls are useful for
16
           determining the roles of the various processes of a communicator. (End of advice to
17
           users.)
18
19
20
     MPI_COMM_COMPARE(comm1, comm2, result)
21
22
       IN
                 comm1
                                             first communicator (handle)
23
       IN
                 comm2
                                             second communicator (handle)
24
       OUT
                 result
                                             result (integer)
25
26
     C binding
27
     int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
28
29
     F08 binding
30
     MPI_Comm_compare(comm1, comm2, result, ierror)
^{31}
          TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
32
          INTEGER, INTENT(OUT) :: result
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     F binding
35
     MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
36
          INTEGER COMM1, COMM2, RESULT, IERROR
37
38
     MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical
39
     groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical
40
     in constituents and rank order; these communicators differ only by context. MPI_SIMILAR
41
     results if the group members of both communicators are the same but the rank order differs.
42
     MPI_UNEQUAL results otherwise.
43
44
```

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.

Unofficial Draft for Comment Only

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

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

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

MPI_COMM_DUP(comm, newcomm)

IN	comm	communicator (handle)
OUT	newcomm	copy of comm (handle)

```
C binding
```

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

F08 binding

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

F binding

MPI_COMM_DUP(COMM, NEWCOMM, IERROR) INTEGER COMM, NEWCOMM, IERROR

MPI_COMM_DUP duplicates the existing communicator comm with associated key values and topology information. 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 communicator. MPI_COMM_DUP returns in newcomm a new communicator with the same group or groups, same topology, and any copied cached information, but a new context (see Section 6.7.1).

Advice to users. This operation is used to provide a parallel library with a duplicate communication space that has the same properties as the original communicator. This

Unofficial Draft for Comment Only

```
1
           includes any attributes (see below) and topologies (see Chapter 7). This call is valid
\mathbf{2}
           even if there are pending point-to-point communications involving the communicator
3
           comm. A typical call might involve a MPI_COMM_DUP at the beginning of the
4
           parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end
5
           of the call. Other models of communicator management are also possible.
6
           This call applies to both intra- and inter-communicators. (End of advice to users.)
7
8
           Advice to implementors. One need not actually copy the group information, but only
9
           add a new reference and increment the reference count. Copy on write can be used
10
           for the cached information. (End of advice to implementors.)
11
12
13
     MPI_COMM_DUP_WITH_INFO(comm, info, newcomm)
14
15
       IN
                                             communicator (handle)
                 comm
16
       IN
                 info
                                             info object (handle)
17
       OUT
                 newcomm
                                             copy of comm (handle)
18
19
     C binding
20
21
     int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
22
     F08 binding
23
     MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
24
          TYPE(MPI_Comm), INTENT(IN) ::
                                             comm
25
          TYPE(MPI_Info), INTENT(IN) ::
                                              info
26
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
27
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
28
     F binding
29
30
     MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
^{31}
          INTEGER COMM, INFO, NEWCOMM, IERROR
32
          MPI_COMM_DUP_WITH_INFO behaves exactly as MPI_COMM_DUP except that the
33
     hints provided by the argument info are associated with the output communicator newcomm.
34
35
           Rationale. It is expected that some hints will only be valid at communicator creation
36
           time. However, for legacy reasons, most communicator creation calls do not provide
37
           an info argument. One may associate info hints with a duplicate of any communicator
38
           at creation time through a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.)
39
40
41
     MPI_COMM_IDUP(comm, newcomm, request)
42
43
       IN
                                             communicator (handle)
                 comm
44
       OUT
                                             copy of comm (handle)
                 newcomm
45
       OUT
                 request
                                             communication request (handle)
46
47
48
     C binding
```

· ,)/DT			1
<pre>int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)</pre>			2
F08 binding			3
	<pre>MPI_Comm_idup(comm, newcomm, request, ierror)</pre>		
	(MPI_Comm), INTENT(IN) ::		5
	(MPI_Comm), INTENT(OUT),		6
	(MPI_Request), INTENT(OU) GER, OPTIONAL, INTENT(OU)	-	7
	JER, OFIIONAL, INIENI(00)		8
F bindin	-		9
	_IDUP(COMM, NEWCOMM, REQU	-	10
INTE	GER COMM, NEWCOMM, REQUES	ST, IERROR	11 12
MPI_	COMM_IDUP is a nonblockin	g variant of MPI_COMM_DUP. With the exception	12
of its nonb	locking behavior, the semanti	ics of MPI_COMM_IDUP are as if MPI_COMM_DUP	14
was execu	ted at the time that MPI_CON	MM_IDUP is called. For example, attributes changed	15
		pied to the new communicator. All restrictions and	16
-	_	operations (see Section 5.12) apply to	17
	IM_IDUP and the returned re	-	18
		cator newcomm as an input argument to other MPI	19
functions	before the MPI_COMM_IDUF	operation completes.	20
			21
MPI_COM	1M_IDUP_WITH_INFO(comm	, info, newcomm, request)	22
IN	comm	communicator (handle)	23 24
IN	info	info object (handle)	24 25
		• • • •	26
OUT	newcomm	copy of comm (handle)	27
OUT	request	communication request (handle)	28
			29
C bindin	•		30
int MPI_	Comm_idup_with_info(MPI_(31
	MP1_Comm *newcomm, 1	MPI_Request *request)	32
F08 bind	ling		33
MPI_Comm	_idup_with_info(comm, inf	io, newcomm, request, ierror)	$\frac{34}{35}$
	(MPI_Comm), INTENT(IN) ::		36
	(MPI_Info), INTENT(IN) ::		37
	(MPI_Comm), INTENT(OUT),		38
	(MPI_Request), INTENT(OUT	-	39
INTE	GER, OPTIONAL, INTENT(OUT	C) :: ierror	40
F bindin	g		41
		FO, NEWCOMM, REQUEST, IERROR)	42
INTE	GER COMM, INFO, NEWCOMM,	REQUEST, IERROR	43
MPI	COMM_IDUP_WITH_INFO is	s a nonblocking variant of	44
		the exception of its nonblocking behavior, the se-	45
		NFO are as if MPI_COMM_DUP_WITH_INFO was	46 47
executed a	at the time that MPI_COMM_I	DUP_WITH_INFO is called. For example, attributes	47 48
			-0

1 or info hints changed after MPI_COMM_IDUP_WITH_INFO will not be copied to the new $\mathbf{2}$ communicator. All restrictions and assumptions for nonblocking collective operations (see 3 Section 5.12) apply to MPI_COMM_IDUP_WITH_INFO and the returned request. 4 It is erroneous to use the communicator newcomm as an input argument to other MPI 5functions before the MPI_COMM_IDUP_WITH_INFO operation completes. 6 The MPI COMM IDUP and MPI COMM IDUP WITH INFO functions Rationale. 7 are crucial for the development of purely nonblocking libraries (see [36]). (End of 8 rationale.) 9 10 11 MPI_COMM_CREATE(comm, group, newcomm) 1213IN communicator (handle) comm 14IN group, which is a subset of the group of comm (handle) group 15OUT new communicator (handle) newcomm 1617C binding 18 19int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm) 20F08 binding 21MPI_Comm_create(comm, group, newcomm, ierror) 22TYPE(MPI_Comm), INTENT(IN) :: comm 23TYPE(MPI_Group), INTENT(IN) :: group 24 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2627F binding MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR) 28INTEGER COMM, GROUP, NEWCOMM, IERROR 2930 If comm is an intracommunicator, this function returns a new communicator 31 newcomm with communication group defined by the group argument. No cached information 32 propagates from comm to newcomm. Each process must call MPI_COMM_CREATE with 33 a group argument that is a subgroup of the group associated with comm; this could be 34 MPI_GROUP_EMPTY. The processes may specify different values for the group argument. 35 If a process calls with a non-empty group then all processes in that group must call the 36 function with the same group as argument, that is the same processes in the same order. 37 Otherwise, the call is erroneous. This implies that the set of groups specified across the 38 processes must be disjoint. If the calling process is a member of the group given as group 39 argument, then **newcomm** is a communicator with group as its associated group. In the case 40 that a process calls with a group to which it does not belong, e.g., MPI_GROUP_EMPTY, 41 then MPI_COMM_NULL is returned as newcomm. The function is collective and must be 42called by all processes in the group of comm. 43 44The interface supports the original mechanism from MPI-1.1, which re-Rationale. 45quired the same group in all processes of comm. It was extended in MPI-2.2 to allow 46the use of disjoint subgroups in order to allow implementations to eliminate unnec-

Unofficial Draft for Comment Only

knows the membership of the disjoint subgroups. (*End of rationale.*)

essary communication that MPI_COMM_SPLIT would incur when the user already

47

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

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

(End of rationale.)

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

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.

Unofficial Draft for Comment Only

 $\mathbf{2}$

```
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
                                                                      8
                                                                 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 inter_comm, new_inter_comm;
26
              MPI_Group local_group, group;
27
                          rank = 0; /* rank on left side to include in
              int
28
                                        new inter-comm */
29
30
              /* Construct the original intercommunicator: "inter_comm" */
^{31}
               . . .
32
33
              /* Construct the group of processes to be in new
34
                  intercommunicator */
35
              if (/* I'm on the left side of the intercommunicator */) {
36
                MPI_Comm_group(inter_comm, &local_group);
37
                MPI_Group_incl(local_group, 1, &rank, &group);
38
                MPI_Group_free(&local_group);
39
              }
40
              else
41
                MPI_Comm_group(inter_comm, &group);
42
43
              MPI_Comm_create(inter_comm, group, &new_inter_comm);
44
              MPI_Group_free(&group);
45
46
47
48
```

MPL COM	M_CREATE_GROUP(comm, g	roup tag newcomm)	1
IN	comm	intracommunicator (handle)	2
			3
IN	group	group, which is a subset of the group of $comm$ (handle)	4
IN	tag	tag (integer)	5
OUT	newcomm	new communicator (handle)	6 7
			8
C binding			9
int MPI_C	• •	n comm, MPI_Group group, int tag,	10
	MPI_Comm *newcomm)		11
F08 bindi	ing		12
	create_group(comm, group,	-	13
	MPI_Comm), INTENT(IN) ::		14
	MPI_Group), INTENT(IN) ::	group	15 16
	ER, INTENT(IN) :: tag MPI_Comm), INTENT(OUT) ::	newcomm	17
	ER, OPTIONAL, INTENT(OUT)		18
			19
F binding MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)			20
	ER COMM, GROUP, TAG, NEWC		21
			22
		milar to MPI_COMM_CREATE; however,	23 24
		by all processes in the group of	25
,		GROUP must be called by all processes in group, mm. In addition, MPI_COMM_CREATE_GROUP	26
		ator. MPI_COMM_CREATE_GROUP returns a new	27
-		h the group argument defines the communication	28
		tes from comm to newcomm. Each process must	29
-		bgroup of the group associated with comm; this	30
		empty group is specified, then all processes in that	31 32
		these processes must provide the same arguments,	33
		me members with the same ordering. Otherwise cess is a member of the group given as the group	34
		icator with group as its associated group. If the	35
		p, e.g., group is MPI_GROUP_EMPTY, then the call	36
• •	peration and MPI_COMM_NUL		37
			38
	onale. Functionality similar ed through repeated MPI_IN	to MPI_COMM_CREATE_GROUP can be imple- ITERCOMM_CREATE and	$\frac{39}{40}$

mented through repeated MPI_INTERCOMM_CREATE and MPI_INTERCOMM_MERGE calls that start with the MPI_COMM_SELF communicators at each process in group and build up an intracommunicator with group group [16]. Such an algorithm requires the creation of many intermediate communicators; MPI_COMM_CREATE_GROUP can provide a more efficient implementation that avoids this overhead. (End of rationale.)

Advice to users. An intercommunicator can be created collectively over processes in 47the union of the local and remote groups by creating the local communicator using

Unofficial Draft for Comment Only

41

42

43

44

4546

```
1
          MPI_COMM_CREATE_GROUP and using that communicator as the local communi-
\mathbf{2}
          cator argument to MPI_INTERCOMM_CREATE. (End of advice to users.)
3
4
         The tag argument does not conflict with tags used in point-to-point communication and
     is not permitted to be a wildcard. If multiple threads at a given process perform concurrent
5
     MPI_COMM_CREATE_GROUP operations, the user must distinguish these operations by
6
     providing different tag or comm arguments.
\overline{7}
8
           Advice to users.
                             MPI_COMM_CREATE may provide lower overhead than
9
           MPI_COMM_CREATE_GROUP because it can take advantage of collective communi-
10
           cation on comm when constructing newcomm. (End of advice to users.)
11
12
13
14
     MPI_COMM_SPLIT(comm, color, key, newcomm)
15
       IN
                                             communicator (handle)
                 comm
16
17
       IN
                                             control of subset assignment (integer)
                 color
18
       IN
                 kev
                                             control of rank assignment (integer)
19
       OUT
                 newcomm
                                             new communicator (handle)
20
21
     C binding
22
     int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
23
^{24}
     F08 binding
25
     MPI_Comm_split(comm, color, key, newcomm, ierror)
26
          TYPE(MPI_Comm), INTENT(IN) :: comm
27
          INTEGER, INTENT(IN) :: color, key
28
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
29
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
30
     F binding
^{31}
     MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
32
33
          INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
34
     This function partitions the group associated with comm into disjoint subgroups, one for
35
     each value of color. Each subgroup contains all processes of the same color. Within each
36
     subgroup, the processes are ranked in the order defined by the value of the argument
37
     key, with ties broken according to their rank in the old group. A new communicator is
38
     created for each subgroup and returned in newcomm. A process may supply the color value
39
     MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective
40
     call, but each process is permitted to provide different values for color and key.
41
          With an intracommunicator comm, a call to MPI_COMM_CREATE(comm, group, new-
42
     comm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where
43
     processes that are members of their group argument provide color = number of the group
44
     (based on a unique numbering of all disjoint groups) and key = rank in group, and all
45
     processes that are not members of their group argument provide color = MPI_UNDEFINED.
46
         The value of color must be non-negative or MPI_UNDEFINED.
47
48
```

1 Advice to users. This is an extremely powerful mechanism for dividing a single $\mathbf{2}$ communicating group of processes into k subgroups, with k chosen implicitly by the 3 user (by the number of colors asserted over all the processes). Each resulting com-4 municator will be non-overlapping. Such a division could be useful for defining a $\mathbf{5}$ hierarchy of computations, such as for multigrid, or linear algebra. For intracommunicators, MPI_COMM_SPLIT provides similar capability as MPI_COMM_CREATE to 6 7 split a communicating group into disjoint subgroups. MPI_COMM_SPLIT is useful when some processes do not have complete information of the other members in their 8 9 group, but all processes know (the color of) the group to which they belong. In this case, the MPI implementation discovers the other group members via communication. 10 11MPI_COMM_CREATE is useful when all processes have complete information of the 12members of their group. In this case, MPI can avoid the extra communication required 13 to discover group membership. MPI_COMM_CREATE_GROUP is useful when all pro-14cesses in a given group have complete information of the members of their group and 15synchronization with processes outside the group can be avoided.

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

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

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

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

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

Unofficial Draft for Comment Only

16

17

18

19

20

21

22

23

 24

25 26

27

28 29

30

31 32

33

34

35

36

37

38 39

40

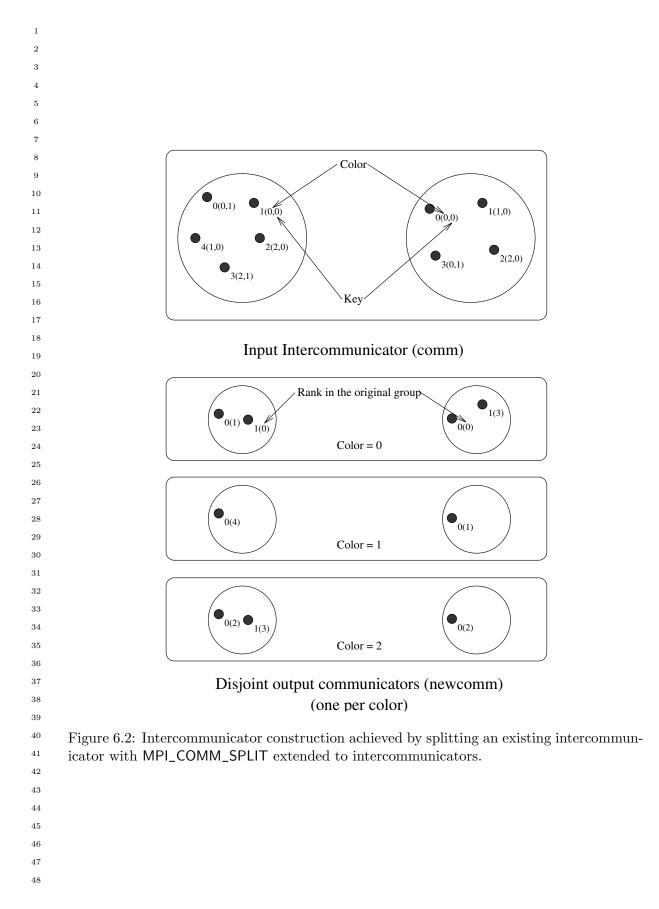
41

42

43 44

45

46



```
1
        /* Client code */
                                                                                         \mathbf{2}
        MPI_Comm multiple_server_comm;
                                                                                         3
        MPI_Comm single_server_comm;
        int
                    color, rank, num_servers;
                                                                                         4
                                                                                         5
                                                                                         6
         /* Create intercommunicator with clients and servers:
                                                                                         7
            multiple_server_comm */
                                                                                         8
         . . .
                                                                                         9
                                                                                         10
        /* Find out the number of servers available */
                                                                                         11
        MPI_Comm_remote_size(multiple_server_comm, &num_servers);
                                                                                        12
        /* Determine my color */
                                                                                         13
                                                                                        14
        MPI_Comm_rank(multiple_server_comm, &rank);
                                                                                         15
        color = rank % num_servers;
                                                                                         16
                                                                                         17
        /* Split the intercommunicator */
                                                                                         18
        MPI_Comm_split(multiple_server_comm, color, rank,
                                                                                         19
                         &single_server_comm);
                                                                                        20
The following is the corresponding server code:
                                                                                        21
                                                                                        22
        /* Server code */
                                                                                        23
        MPI_Comm multiple_client_comm;
                                                                                        24
        MPI_Comm single_server_comm;
                                                                                        25
        int
                    rank;
                                                                                         26
                                                                                        27
        /* Create intercommunicator with clients and servers:
                                                                                        28
            multiple_client_comm */
                                                                                        29
         . . .
                                                                                        30
                                                                                        31
        /* Split the intercommunicator for a single server per group
                                                                                        32
            of clients */
                                                                                        33
        MPI_Comm_rank(multiple_client_comm, &rank);
                                                                                        34
        MPI_Comm_split(multiple_client_comm, rank, 0,
                                                                                        35
                         &single_server_comm);
                                                                                        36
                                                                                        37
                                                                                        38
MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)
                                                                                        39
                                                                                         40
 IN
           comm
                                       communicator (handle)
                                                                                        41
 IN
           split_type
                                       type of processes to be grouped together (integer)
                                                                                        42
 IN
           key
                                      control of rank assignment (integer)
                                                                                        43
                                                                                        44
 IN
           info
                                      info argument (handle)
                                                                                         45
 OUT
           newcomm
                                      new communicator (handle)
                                                                                         46
                                                                                         47
C binding
                                                                                         48
```

```
1
     int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
\mathbf{2}
                    MPI_Info info, MPI_Comm *newcomm)
3
     F08 binding
4
     MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
5
          TYPE(MPI_Comm), INTENT(IN) :: comm
6
          INTEGER, INTENT(IN) :: split_type, key
7
          TYPE(MPI_Info), INTENT(IN) :: info
8
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
9
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     F binding
12
     MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
13
          INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
14
     This function partitions the group associated with comm into disjoint subgroups, based on
15
     the type specified by split_type. Each subgroup contains all processes of the same type.
16
     Within each subgroup, the processes are ranked in the order defined by the value of the
17
     argument key, with ties broken according to their rank in the old group. A new commu-
18
     nicator is created for each subgroup and returned in newcomm. This is a collective call;
19
     all processes must provide the same split_type, but each process is permitted to provide
20
     different values for key. An exception to this rule is that a process may supply the type
21
     value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL.
22
         The following type is predefined by MPI:
23
24
      MPI_COMM_TYPE_SHARED — this type splits the communicator into subcommunicators,
25
           each of which can create a shared memory region.
26
27
           Advice to implementors.
                                    Implementations can define their own types, or use the
28
          info argument, to assist in creating communicators that help expose platform-specific
29
          information to the application. (End of advice to implementors.)
30
^{31}
     6.4.3 Communicator Destructors
32
33
34
     MPI_COMM_FREE(comm)
35
36
       INOUT
                                            communicator to be destroyed (handle)
                comm
37
38
     C binding
39
     int MPI_Comm_free(MPI_Comm *comm)
40
     F08 binding
41
     MPI_Comm_free(comm, ierror)
42
          TYPE(MPI_Comm), INTENT(INOUT) :: comm
43
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     F binding
46
     MPI_COMM_FREE(COMM, IERROR)
47
          INTEGER COMM, IERROR
48
```

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 6.7) are called in arbitrary order.

Advice to implementors. Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

6.4.4 Communicator Info

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

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

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

- mpi_assert_no_any_tag (boolean, default: false): If set to true, then the implementation
 may assume that the process will not use the MPI_ANY_TAG wildcard on the given
 communicator.
- mpi_assert_no_any_source (boolean, default: false): If set to true, then the implementation
 may assume that the process will not use the MPI_ANY_SOURCE wildcard on the given
 communicator.
- mpi_assert_exact_length (boolean, default: false): If set to true, then the implementation may assume that the lengths of messages received by the process are equal to the lengths of the corresponding receive buffers, for point-to-point communication operations on the given communicator.
- mpi_assert_allow_overtaking (boolean, default: false): If set to true, then the implementation may assume that point-to-point communications on the given communicator do

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $45 \\ 46$

not rely on the non-overtaking rule specified in Section 3.5. In other words, the application asserts that send operations are not required to be matched at the receiver in the order in which the send operations were posted by the sender, and receive operations are not required to be matched in the order in which they were posted by the receiver.

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

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

```
13 \\ 14
```

1

2

3

4

5

6

7

8 9

10

11

12

```
<sup>15</sup> MPI_COMM_SET_INFO(comm, info)
```

```
16
       INOUT
                 comm
                                            communicator (handle)
17
       IN
                info
                                            info object (handle)
18
19
     C binding
20
21
     int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
22
     F08 binding
23
     MPI_Comm_set_info(comm, info, ierror)
24
         TYPE(MPI_Comm), INTENT(IN) ::
                                            comm
25
         TYPE(MPI_Info), INTENT(IN) ::
                                            info
26
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
27
```

```
<sup>28</sup> F binding
```

```
    MPI_COMM_SET_INFO(COMM, INFO, IERROR)
    INTEGER COMM, INFO, IERROR
```

³¹ MPI_COMM_SET_INFO updates the hints of the communicator associated with comm ³² using the hints provided in info. This operation has no effect on previously set or defaulted ³⁴ hints that are not specified by info. It also has no effect on previously set or defaulted ³⁵ hints that are specified by info, but are ignored by the MPI implementation in this call to ³⁶ MPI_COMM_SET_INFO. MPI_COMM_SET_INFO is a collective routine. The info object ³⁷ may be different on each process, but any info entries that an implementation requires to ³⁸ be the same on all processes must appear with the same value in each process's info object.

39 Some info items that an implementation can use when it creates Advice to users. 40 a communicator cannot easily be changed once the communicator has been created. 41 Thus, an implementation may ignore hints issued in this call that it would have 42accepted in a creation call. An implementation may also be unable to update certain 43 info hints in a call to MPI_COMM_SET_INFO. MPI_COMM_GET_INFO can be used to 44determine whether updates to existing info hints were ignored by the implementation. 45(End of advice to users.) 46

Advice to users. Setting info hints on the predefined communicators
 MPI_COMM_WORLD and MPI_COMM_SELF may have unintended effects, as changes to

Unofficial Draft for Comment Only

these global objects may affect all components of the application, including libraries and tools. Users must ensure that all components of the application that use a given communicator, including libraries and tools, can comply with any info hints associated with that communicator. (*End of advice to users.*)

MPI_COMM_GET_INFO(comm, info_used)			
IN	comm	communicator object (handle)	
OUT	info_used	new info object (handle)	

C binding

```
int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
```

F08 binding

<pre>MPI_Comm_get_info(comm, info_used,</pre>	ierror)
TYPE(MPI_Comm), INTENT(IN) ::	comm
TYPE(MPI_Info), INTENT(OUT) ::	info_used
INTEGER, OPTIONAL, INTENT(OUT)	:: ierror

F binding

```
MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
INTEGER COMM, INFO_USED, IERROR
```

MPI_COMM_GET_INFO returns a new info object containing the hints of the communicator associated with comm. The current setting of all hints related to this communicator is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

6.5 Motivating Examples

```
6.5.1 Current Practice #1
```

```
Example #1a:
```

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

 $45 \\ 46$

```
1
           MPI_Finalize();
\mathbf{2}
           return 0;
3
         }
4
     Example \#1a is a do-nothing program that initializes itself, and refers to the "all" commu-
5
     nicator, and prints a message. It terminates itself too. This example does not imply that
6
     MPI supports printf-like communication itself.
7
     Example #1b (supposing that size is even):
8
9
          int main(int argc, char *argv[])
10
          {
11
              int me, size;
12
              int SOME_TAG = 0;
13
              . . .
14
             MPI_Init(&argc, &argv);
15
16
             MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
17
             MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
18
19
              if((me % 2) == 0)
20
              {
21
                 /* send unless highest-numbered process */
22
                 if((me + 1) < size)
23
                    MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
24
              }
25
              else
26
                 MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
27
28
              . . .
29
             MPI_Finalize();
30
             return 0;
^{31}
          }
32
33
     Example #1b schematically illustrates message exchanges between "even" and "odd" pro-
34
     cesses in the "all" communicator.
35
36
     6.5.2 Current Practice #2
37
         int main(int argc, char *argv[])
38
         {
39
           int me, count;
40
           void *data;
41
           . . .
42
43
           MPI_Init(&argc, &argv);
44
           MPI_Comm_rank(MPI_COMM_WORLD, &me);
45
46
           if(me == 0)
47
           ſ
48
```

```
1
         /* get input, create buffer ''data'' */
                                                                                        \mathbf{2}
          . . .
                                                                                        3
     }
                                                                                        4
     MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
                                                                                        5
                                                                                        6
                                                                                        7
     . . .
                                                                                        8
     MPI_Finalize();
                                                                                        9
     return 0;
   }
                                                                                        10
                                                                                        11
This example illustrates the use of a collective communication.
                                                                                        12
                                                                                        13
      (Approximate) Current Practice #3
6.5.3
                                                                                        14
                                                                                        15
  int main(int argc, char *argv[])
                                                                                        16
  ſ
                                                                                        17
    int me, count, count2;
                                                                                        18
    void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
                                                                                        19
    MPI_Group group_world, grprem;
                                                                                        20
    MPI_Comm commslave;
                                                                                        21
    static int ranks[] = {0};
                                                                                        22
    . . .
                                                                                        23
    MPI_Init(&argc, &argv);
                                                                                        ^{24}
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);
                                                                                        25
    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
                                                                                        26
                                                                                        27
    MPI_Group_excl(group_world, 1, ranks, &grprem); /* local */
                                                                                        28
    MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);
                                                                                        29
                                                                                        30
    if(me != 0)
                                                                                        31
    ſ
                                                                                        32
      /* compute on slave */
                                                                                        33
      . . .
                                                                                        34
      MPI_Reduce(send_buf,recv_buf,count, MPI_INT, MPI_SUM, 1, commslave);
                                                                                        35
      . . .
                                                                                        36
      MPI_Comm_free(&commslave);
                                                                                        37
    }
                                                                                        38
    /* zero falls through immediately to this reduce, others do later... */
                                                                                        39
    MPI_Reduce(send_buf2, recv_buf2, count2,
                                                                                        40
                MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
                                                                                        41
                                                                                        42
    MPI_Group_free(&group_world);
                                                                                        43
    MPI_Group_free(&grprem);
                                                                                        44
    MPI_Finalize();
                                                                                        45
    return 0;
                                                                                        46
  }
                                                                                        47
```

This example illustrates how a group consisting of all but the zeroth process of the "all"

¹ group is created, and then how a communicator is formed (commslave) for that new group. ² The new communicator is used in a collective call, and all processes execute a collective call ³ in the MPI_COMM_WORLD context. This example illustrates how the two communicators ⁴ (that inherently possess distinct contexts) protect communication. That is, communication ⁵ in MPI_COMM_WORLD is insulated from communication in commslave, and vice versa. ⁶ In summary "group safety" is achieved via communicators because distingt contexts

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

8 9

7

10

14

6.5.4 Example #4

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

```
#define TAG_ARBITRARY 12345
15
        #define SOME_COUNT
                                    50
16
17
        int main(int argc, char *argv[])
18
        {
19
          int me;
20
          MPI_Request request[2];
21
          MPI_Status status[2];
22
          MPI_Group group_world, subgroup;
23
          int ranks[] = {2, 4, 6, 8};
24
          MPI_Comm the_comm;
25
          . . .
26
          MPI_Init(&argc, &argv);
27
          MPI_Comm_group(MPI_COMM_WORLD, &group_world);
28
29
          MPI_Group_incl(group_world, 4, ranks, &subgroup); /* local */
30
          MPI_Group_rank(subgroup, &me);
                                                 /* local */
31
32
          MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
33
34
          if(me != MPI_UNDEFINED)
35
          {
36
               MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
37
                                   the_comm, request);
38
               MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
39
                                   the_comm, request+1);
40
               for(i = 0; i < SOME_COUNT; i++)</pre>
41
                 MPI_Reduce(..., the_comm);
42
               MPI_Waitall(2, request, status);
43
44
               MPI_Comm_free(&the_comm);
45
          }
46
47
          MPI_Group_free(&group_world);
48
```

```
1
     MPI_Group_free(&subgroup);
                                                                                            \mathbf{2}
     MPI_Finalize();
                                                                                            3
     return 0;
                                                                                            4
   }
                                                                                            5
                                                                                            6
6.5.5
       Library Example \#1
                                                                                            7
The main program:
                                                                                            8
                                                                                            9
   int main(int argc, char *argv[])
                                                                                            10
   {
                                                                                            11
     int done = 0;
                                                                                            12
     user_lib_t *libh_a, *libh_b;
                                                                                            13
     void *dataset1, *dataset2;
                                                                                            14
      . . .
                                                                                            15
     MPI_Init(&argc, &argv);
                                                                                            16
     . . .
                                                                                            17
     init_user_lib(MPI_COMM_WORLD, &libh_a);
                                                                                            18
     init_user_lib(MPI_COMM_WORLD, &libh_b);
                                                                                            19
     . . .
                                                                                            20
     user_start_op(libh_a, dataset1);
                                                                                            21
     user_start_op(libh_b, dataset2);
                                                                                            22
     . . .
                                                                                            23
     while(!done)
                                                                                            ^{24}
     {
                                                                                            25
         /* work */
                                                                                            26
         . . .
                                                                                            27
         MPI_Reduce(..., MPI_COMM_WORLD);
                                                                                            28
         . . .
                                                                                            29
         /* see if done */
                                                                                            30
         . . .
                                                                                            ^{31}
     }
                                                                                            32
     user_end_op(libh_a);
                                                                                            33
     user_end_op(libh_b);
                                                                                            34
                                                                                            35
     uninit_user_lib(libh_a);
                                                                                            36
     uninit_user_lib(libh_b);
                                                                                            37
     MPI_Finalize();
                                                                                            38
     return 0;
                                                                                            39
   }
                                                                                            40
                                                                                            ^{41}
The user library initialization code:
                                                                                            42
   void init_user_lib(MPI_Comm comm, user_lib_t **handle)
                                                                                            43
   {
                                                                                            44
     user_lib_t *save;
                                                                                            45
                                                                                            46
     user_lib_initsave(&save); /* local */
                                                                                            47
     MPI_Comm_dup(comm, &(save->comm));
                                                                                            48
```

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

```
1
     . . .
                                                                                        \mathbf{2}
     MPI_Init(&argc, &argv);
                                                                                        3
     MPI_Comm_group(MPI_COMM_WORLD, &group_world);
                                                                                        4
     MPI_Group_incl(group_world, size_list_a, list_a, &group_a);
                                                                                        5
                                                                                        6
     MPI_Group_incl(group_world, size_list_b, list_b, &group_b);
                                                                                        7
                                                                                        8
     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
                                                                                        9
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
                                                                                        10
                                                                                        11
     if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
                                                                                        12
     if(comm_b != MPI_COMM_NULL)
                                                                                        13
        MPI_Comm_rank(comm_b, &mb);
                                                                                        14
                                                                                        15
                                                                                        16
     if(comm_a != MPI_COMM_NULL)
                                                                                        17
        lib_call(comm_a);
                                                                                        18
     if(comm_b != MPI_COMM_NULL)
                                                                                        19
                                                                                        20
     ſ
       lib_call(comm_b);
                                                                                        21
       lib_call(comm_b);
                                                                                        22
     }
                                                                                        23
                                                                                        ^{24}
                                                                                        25
     if(comm_a != MPI_COMM_NULL)
                                                                                        26
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
                                                                                        27
       MPI_Comm_free(&comm_b);
                                                                                        28
     MPI_Group_free(&group_a);
                                                                                        29
                                                                                        30
     MPI_Group_free(&group_b);
                                                                                        ^{31}
     MPI_Group_free(&group_world);
                                                                                        32
     MPI_Finalize();
                                                                                        33
     return 0;
                                                                                        34
   }
                                                                                        35
The library:
                                                                                        36
                                                                                        37
   void lib_call(MPI_Comm comm)
   {
                                                                                        38
                                                                                        39
     int me, done = 0;
     MPI_Status status;
                                                                                        40
                                                                                        41
     MPI_Comm_rank(comm, &me);
                                                                                        42
     if(me == 0)
        while(!done)
                                                                                        43
                                                                                        44
        {
            MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
                                                                                        45
                                                                                        46
            . . .
                                                                                        47
        }
                                                                                        48
     else
```

```
{
            /* work */
            MPI_Send(..., 0, ARBITRARY_TAG, comm);
          }
6
    #ifdef EXAMPLE_2C
          /* include (resp, exclude) for safety (resp, no safety): */
          MPI_Barrier(comm);
9
     #endif
10
        }
```

The above example is really three examples, depending on whether or not one includes rank 123 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates 13 that, despite contexts, subsequent calls to lib_call with the same context need not be safe 14from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is 15added. What this demonstrates is that libraries have to be written carefully, even with 16contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from 17back-masking. 18

Algorithms like "reduce" and "allreduce" have strong enough source selectivity prop-19erties so that they are inherently okay (no back-masking), provided that MPI provides basic 20guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root 21or different roots (see [56]). Here we rely on two guarantees of MPI: pairwise ordering of 22 messages between processes in the same context, and source selectivity — deleting either 23feature removes the guarantee that back-masking cannot be required. 24

Algorithms that try to do non-deterministic broadcasts or other calls that include wild-25card operations will not generally have the good properties of the deterministic implemen-26tations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize 27the monotonically increasing tags (within a communicator scope) to keep things straight. 28

All of the foregoing is a supposition of "collective calls" implemented with point-to-29 point operations. MPI implementations may or may not implement collective calls using 30 point-to-point operations. These algorithms are used to illustrate the issues of correctness 31 and safety, independent of how MPI implements its collective calls. See also Section 6.9. 32

33 34

35 36

37

38

1

2

3

4 5

 $\overline{7}$

8

11

Inter-Communication 6.6

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

39 All communication described thus far has involved communication between processes 40that are members of the same group. This type of communication is called "intra-com-41 munication" and the communicator used is called an "intra-communicator," as we have 42noted earlier in the chapter.

43 In modular and multi-disciplinary applications, different process groups execute distinct 44modules and processes within different modules communicate with one another in a pipeline 45or a more general module graph. In these applications, the most natural way for a process 46to specify a target process is by the rank of the target process within the target group. In 47applications that contain internal user-level servers, each server may be a process group that 48provides services to one or more clients, and each client may be a process group that uses the

Unofficial Draft for Comment Only

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 MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the processes in the created intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to intercommunicators makes the most sense when the groups are disjoint. (End of advice to users.)

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point and collective communication is the same for both interand intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- A communicator will provide either intra- or inter-communication, never both.

The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_CART_CREATE).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

group	43
	44
send_context	45
receive_context	46
source	47
	48

 $\mathbf{2}$

 $\mathbf{5}$

1 2 3 4 5		the process in the group (remote=lo	local group. For cal), source is the t are identical. A	scribes the remote group, and <i>source</i> is the rank of r intra-communicators, <i>group</i> is the communicator rank of the process in this group, and <i>send context</i> A group can be represented by a rank-to-absolute-	
6 7 8 9 10		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			
11		• C _P .group d	escribes the grou	p \mathcal{Q} and $\mathbf{C}_{\mathcal{Q}}$.group describes the group \mathcal{P} .	
12 13 14	• $C_{\mathcal{P}}$.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} .				
14		• $\mathbf{C}_{\mathcal{P}}$.source is rank of P in \mathcal{P} and $\mathbf{C}_{\mathcal{Q}}$.source is rank of Q in \mathcal{Q} .			
16 17 18 19	Assume that P sends a message to Q using the inter-communicator. Then P uses the group table to find the absolute address of Q ; source and send_context are appended to the message. Assume that Q posts a receive with an explicit source argument using the inter- communicator. Then Q matches receive_context to the message context and source argument to the message source.				
20 21 22					
23		The same algorithm is appropriate for intra-communicators as well.			
24 25 26 27 28	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. (<i>End of advice to implementors.</i>)				
29 30 31	6.6.1	Inter-communica	ator Accessors		
31 32 33	MPI_	COMM_TEST_IN1	ER(comm, flag)		
34	IN	comm		communicator (handle)	
35	OU.	T flag		(logical)	
36 37 38 39	C binding int MPI_Comm_test_inter(MPI_Comm comm, int *flag)				
40	F08 binding				
41 42	<pre>MPI_Comm_test_inter(comm, flag, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm</pre>				
43	LOGICAL, INTENT(OUT) :: flag				
44		INTEGER, OPTIONA	0	:: ierror	
45	F binding				
46 47	MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)				
48	INTEGER COMM, IERROR				

LOGICAL FLAG

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

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

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

Table 6.1: MPI_COMM_* Function Behavior (in Inter-Communication Mode)

Furthermore, the operation MPI_COMM_COMPARE is valid for inter-communicators. Both communicators must be either intra- or inter-communicators, or else MPI_UNEQUAL results. Both corresponding local and remote groups must compare correctly to get the results MPI_CONGRUENT or MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result because either the local or remote groups were similar but not identical.

The following accessors provide consistent access to the remote group of an intercommunicator. The following are all local operations.

MPI_COMM_REMOTE_SIZE(comm, size)

IN	comm	inter-communicator (handle)
OUT	size	number of processes in the remote group of comm (non-negative integer)

C binding

int MPI_Comm_remote_size(MPI_Comm comm, int *size)

F08 binding

MPI_Comm_remote_size(comm, size, ierror)
 TYPE(MPI_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(OUT) :: size
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR

MPI_COMM_REMOTE_GROUP(comm, group)

IN	comm	inter-communicator (handle)
OUT	group	remote group corresponding to $comm$ (handle)

C binding

 $\mathbf{2}$

int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)

```
<sup>2</sup><sub>3</sub> F08 binding
```

```
MPI_Comm_remote_group(comm, group, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Group), INTENT(OUT) :: group
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)

- INTEGER COMM, GROUP, IERROR
- 11 12

13

14

15 16

17

1

4

5

6

7 8

9

10

Rationale. Symmetric access to both the local and remote groups of an intercommunicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE have been provided. (*End of rationale.*)

6.6.2 Inter-communicator Operations

¹⁸ This section introduces four blocking inter-communicator operations.

MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-communicator; the function MPI_INTERCOMM_MERGE creates an intra-communicator by merg ing the local and remote groups of an inter-communicator. The functions MPI_COMM_DUP
 and MPI_COMM_FREE, introduced previously, duplicate and free an inter-communicator, respectively.

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

The function MPI_INTERCOMM_CREATE can be used to create an inter-communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator. Furthermore, members of each group know the rank of their leader.

³⁵ Construction of an inter-communicator from two intra-communicators requires separate ³⁶ collective operations in the local group and in the remote group, as well as a point-to-point ³⁷ 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

⁴⁴ MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or carte ⁴⁵ sian topology capabilities to that intra-communicator, creating an appropriate topology ⁴⁶ oriented intra-communicator. Alternatively, it may be reasonable to devise one's own ap ⁴⁷ plication topology mechanisms for this case, without loss of generality.

MPI_INTE	Ϋ́Υ,	n, local_leader, peer_comm, remote_leader, tag, newin-	1
	tercomm)		2
IN	local_comm	local intra-communicator (handle)	3 4
IN	local_leader	rank of local group leader in local_comm (non-negative	5
		integer)	6
IN	peer_comm	"peer" communicator; significant only at the local_leader (handle)	7 8
IN	remote_leader	rank of remote group leader in peer_comm; significant	9 10
		only at the local_leader (non-negative integer)	11
IN	tag	tag (integer)	12
OUT	newintercomm	new inter-communicator (handle)	13
			14 15
C binding			16
int MPI_I		local_comm, int local_leader,	17
	MPI_Comm peer_comm, MPI_Comm *newintercom	int remote_leader, int tag,	18
	_		19
F08 bind	0		20
MPI_Inter		<pre>local_leader, peer_comm, remote_leader,</pre>	21
TVDF (tag, newintercomm, i		22
TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm INTEGER, INTENT(IN) :: local_leader, remote_leader, tag			23 24
	MPI_Comm), INTENT(OUT) :	C C	25
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	26
F binding	r		27
	COMM_CREATE(LOCAL_COMM, I	LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,	28 29
тмтес	TAG, NEWINTERCOMM, I		30
	TERCOMM, IERROR	DER, PEER_COMM, REMOTE_LEADER, TAG,	31
	-		32
		r. It is collective over the union of the local and	33
-		e identical local_comm and local_leader arguments	34
within eac	n group. Wildcards are not pe	ermitted for remote_leader, local_leader, and tag.	$\frac{35}{36}$
			37
MPI_INTE	RCOMM_MERGE(intercomm,	high, newintracomm)	38
IN	intercomm	Inter-Communicator (handle)	39
IN	high	(logical)	40
OUT	newintracomm	new intra-communicator (handle)	41
001		new mira-communicator (nanule)	42
C binding	у.		43 44
	ntercomm_merge(MPI_Comm_i	intercomm, int high,	44 45
		, ₀ ,	

MPI_Comm *newintracomm)

F08 binding

 $46 \\ 47$

```
1
2
3
                       Group 0
                                            Group 1
                                                                  Group 2
4
5
6
7
                                Figure 6.3: Three-group pipeline
8
9
     MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
10
          TYPE(MPI_Comm), INTENT(IN) ::
                                             intercomm
11
          LOGICAL, INTENT(IN) :: high
12
          TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
13
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
14
15
     F binding
16
     MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
17
          INTEGER INTERCOMM, NEWINTRACOMM, IERROR
18
          LOGICAL HIGH
19
     This function creates an intra-communicator from the union of the two groups that are
20
     associated with intercomm. All processes should provide the same high value within each
21
     of the two groups. If processes in one group provided the value high = false and processes
22
     in the other group provided the value high = true then the union orders the "low" group
23
     before the "high" group. If all processes provided the same high argument then the order
24
     of the union is arbitrary. This call is blocking and collective within the union of the two
25
     groups.
26
         The error handler on the new intercommunicator in each process is inherited from
27
     the communicator that contributes the local group. Note that this can result in different
28
     processes in the same communicator having different error handlers.
29
30
           Advice to implementors. The implementation of MPI_INTERCOMM_MERGE,
31
           MPI_COMM_FREE, and MPI_COMM_DUP are similar to the implementation of
32
           MPI_INTERCOMM_CREATE, except that contexts private to the input inter-com-
33
           municator are used for communication between group leaders rather than contexts
34
           inside a bridge communicator. (End of advice to implementors.)
35
36
     6.6.3 Inter-Communication Examples
37
38
     Example 1: Three-Group "Pipeline"
39
     Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires
40
     one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1
41
     inter-communicator.
42
43
         int main(int argc, char *argv[])
44
         Ł
45
           MPI_Comm
                       myComm;
                                       /* intra-communicator of local sub-group */
46
           MPI_Comm
                       myFirstComm;
                                       /* inter-communicator */
47
           MPI_Comm
                       mySecondComm; /* second inter-communicator (group 1 only) */
48
```

```
1
  int membershipKey;
                                                                                   \mathbf{2}
  int rank;
                                                                                   3
  MPI_Init(&argc, &argv);
                                                                                   4
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                                                                                   5
                                                                                   6
                                                                                   7
  /* User code must generate membershipKey in the range [0, 1, 2] */
                                                                                   8
  membershipKey = rank % 3;
                                                                                   9
                                                                                   10
  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
                                                                                  11
                                                                                  12
  /* Build inter-communicators. Tags are hard-coded. */
                                                                                  13
                                                                                  14
  if (membershipKey == 0)
                                                                                   15
  {
                          /* Group 0 communicates with group 1. */
                                                                                   16
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
                                                                                   17
                           1, &myFirstComm);
                                                                                   18
  }
  else if (membershipKey == 1)
                                                                                   19
                                                                                  20
  ſ
                  /* Group 1 communicates with groups 0 and 2. */
                                                                                  21
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
                           1, &myFirstComm);
                                                                                  22
                                                                                  23
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
                                                                                   24
                           12, &mySecondComm);
                                                                                  25
  }
                                                                                   26
  else if (membershipKey == 2)
                          /* Group 2 communicates with group 1. */
                                                                                  27
  {
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
                                                                                  28
                                                                                  29
                           12, &myFirstComm);
                                                                                  30
  }
                                                                                   31
  /* Do work ... */
                                                                                   32
                                                                                   33
                                                                                  34
  switch(membershipKey) /* free communicators appropriately */
  ſ
                                                                                  35
                                                                                  36
  case 1:
                                                                                  37
     MPI_Comm_free(&mySecondComm);
                                                                                   38
  case 0:
                                                                                   39
  case 2:
     MPI_Comm_free(&myFirstComm);
                                                                                   40
                                                                                   41
     break;
                                                                                   42
  }
                                                                                   43
                                                                                   44
  MPI_Finalize();
                                                                                   45
  return 0;
                                                                                   46
}
                                                                                   47
```

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

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
- 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.)

 $\mathbf{2}$

 31

One difficulty is the potential for size differences between Fortran integers and C $\mathbf{2}$ pointers. For this reason, the Fortran versions of these routines use integers of kind 3 MPI_ADDRESS_KIND. 4

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (End of advice to implementors.)

6.7.1 Functionality

1

5

6

7

8

9

10 11

1213

14

15

16

17

18 19

20

21

22

2324

25

26

2728

29

30

31

32

33 34

3536

37

38

39

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.)

40 The choice of key values is under control of MPI. This allows MPI to optimize its 41 implementation of attribute sets. It also avoids conflict between independent modules 42caching information on the same communicators. 43

A much smaller interface, consisting of just a callback facility, would allow the entire 44 caching facility to be implemented by portable code. However, with the minimal call-45back interface, some form of table searching is implied by the need to handle arbitrary 4647 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 48

	efficiency "hit" inherent in the minimal interface, the more complete interface defined	1 2 3	
	here is seen to be superior. (<i>Linu of untile to implementors.</i>)		
MPI	MPI provides the following services related to caching. They are all process local.		
		6	
6.7.2	Communicators	7	
Func	ions for eaching on communicators are:	8 9	
	-	9 10	
MPI_	COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, ex- tra_state)	11 12	
IN	comm converter frame and conversion for comm knowed (function)	13 14	
IN		14 15	
OL		16	
		17	
IN	extra_state extra state for callback function	18	
C L		19	
		20 21	
1110		22	
		23	
EU6	hinding	24	
	Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,	25 26	
	PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn	27	
	PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn	28	
	NTEGER, INTENT(OUT) :: comm_keyval	29 30	
	NTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state	31	
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32	
F bi	nding	33	
MPI_	COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,	34	
	EXTRA_STATE, IERROR)	35	
		36	
		37	
		38	
	Generates a new attribute key. Keys are locally unique in a process, and opaque to	39 40	
,	though they are explicitly stored in integers. Once allocated, the key value can be	41	
	to associate attributes and access them on any locally defined communicator. $\frac{1}{2}$	42	
	C callback functions are:	43	
type	<pre>lef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,</pre>	44	
	<pre>void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag);</pre>	45	
	voia mattribute_var_out, int milag/,	46	
and		47	
	4	48	

```
1
     typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
\mathbf{2}
                    void *attribute_val, void *extra_state);
3
     which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.
4
     With the mpi_f08 module, the Fortran callback functions are:
5
     ABSTRACT INTERFACE
6
       SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
7
       attribute_val_in, attribute_val_out, flag, ierror)
8
            TYPE(MPI_Comm) :: oldcomm
9
            INTEGER :: comm_keyval, ierror
10
            INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
11
            attribute_val_out
12
            LOGICAL :: flag
13
14
     and
15
     ABSTRACT INTERFACE
16
       SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
17
       attribute_val, extra_state, ierror)
18
            TYPE(MPI_Comm) :: comm
19
            INTEGER :: comm_keyval, ierror
20
            INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
21
     With the mpi module and mpif.h, the Fortran callback functions are:
22
     SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
23
                    ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
24
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
25
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
26
                     ATTRIBUTE_VAL_OUT
27
         LOGICAL FLAG
28
29
     and
30
     SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
31
                    EXTRA_STATE, IERROR)
32
         INTEGER COMM, COMM_KEYVAL, IERROR
33
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
34
         The comm_copy_attr_fn function is invoked when a communicator is duplicated by
35
     MPI_COMM_DUP or MPI_COMM_IDUP. comm_copy_attr_fn should be of type
36
     MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in
37
     oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its
38
     corresponding attribute. If it returns flag = 0 or .FALSE., then the attribute is deleted in the
39
     duplicated communicator. Otherwise (flag = 1 or .TRUE.), the new attribute value is set to
40
     the value returned in attribute_val_out. The function returns MPI_SUCCESS on success and
41
     an error code on failure (in which case MPI_COMM_DUP or MPI_COMM_IDUP will fail).
42
         The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN
43
     or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a
44
     function that does nothing other than returning flag = 0 or .FALSE. (depending on whether
45
     the keyval was created with a C or Fortran binding to MPI_COMM_CREATE_KEYVAL) and
46
     MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1 or
47
     .TRUE., returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.
48
```

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

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

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

Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows. The comm_delete_attr_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_COMM_DELETE_ATTR. comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.

This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_FREE will fail).

The argument comm_delete_attr_fn may be specified as MPI_COMM_NULL_DELETE_FN from either C or Fortran. MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose use is deprecated.

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.

The special key value MPI_KEYVAL_INVALID is never returned by MPI_COMM_CREATE_KEYVAL. Therefore, it can be used for static initialization of key values.

Advice to implementors. The predefined Fortran functions 39 MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and 40 MPI_COMM_NULL_DELETE_FN are defined in the mpi module (and mpif.h) and 41 the mpi_f08 module with the same name, but with different interfaces. Each function 42can coexist twice with the same name in the same MPI library, one routine as an 43 implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other 44routine within mpi_f08 declared with CONTAINS. These routines have different link 45names, which are also different to the link names used for the routines used in C. 46(End of advice to implementors.) 47

1 2

3

4

5

6

 $\overline{7}$

8 9

10

11

12

13

14 15

16

17

18

19

20 21

22

23

24 25

26

27

28

29

30 31

32 33

34

35

36 37

38

```
1
                             Callbacks, including the predefined Fortran functions
           Advice to users.
\mathbf{2}
           MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and
3
           MPI_COMM_NULL_DELETE_FN should not be passed from one application routine
4
           that uses the mpi_f08 module to another application routine that uses the mpi module
5
           or mpif.h, and vice versa; see also the advice to users on page 676. (End of advice to
6
           users.)
7
8
9
     MPI_COMM_FREE_KEYVAL(comm_keyval)
10
11
       INOUT
                 comm_keyval
                                             key value (integer)
12
13
     C binding
14
     int MPI_Comm_free_keyval(int *comm_keyval)
15
     F08 binding
16
     MPI_Comm_free_keyval(comm_keyval, ierror)
17
          INTEGER, INTENT(INOUT) :: comm_keyval
18
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     F binding
21
     MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
22
          INTEGER COMM_KEYVAL, IERROR
23
          Frees an extant attribute key. This function sets the value of keyval to
24
     MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use,
25
     because the actual free does not transpire until after all references (in other communicators
26
     on the process) to the key have been freed. These references need to be explicitly freed by the
27
     program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance,
28
     or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed
29
     communicator.
30
^{31}
32
     MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)
33
       INOUT
                                             communicator to which attribute will be attached (han-
34
                 comm
                                             dle)
35
36
       IN
                 comm_keyval
                                             key value (integer)
37
       IN
                 attribute_val
                                             attribute value
38
39
     C binding
40
     int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void* attribute_val)
41
42
     F08 binding
43
     MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
44
          TYPE(MPI_Comm), INTENT(IN) :: comm
45
          INTEGER, INTENT(IN) :: comm_keyval
46
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
47
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

INTEG			1 2 3 4
by MPI_C MPI_COM function co is erroneous	OMM_GET_ATTR. If the va M_DELETE_ATTR was first o omm_delete_attr_fn was exect us if there is no key with value	attribute value attribute_val for subsequent retrieval lue is already present, then the outcome is as if called to delete the previous value (and the callback uted), and a new value was next stored. The call ne keyval; in particular MPI_KEYVAL_INVALID is an the comm_delete_attr_fn function returned an error	5 6 7 8 9 10 11 12 13
MPI_COM	M_GET_ATTR(comm, comm_	_keyval, attribute_val, flag)	14
IN	comm	communicator to which the attribute is attached (han-dle)	15 16 17
IN	comm_keyval	key value (integer)	18
OUT	attribute_val	attribute value, unless $flag = false$	19
OUT	flag	false if no attribute is associated with the key (logical)	20 21
C binding int MPI_C	0	mm, int comm_keyval, void* attribute_val,	22 23 24 25
TYPE(INTEG INTEG LOGIC	0	keyval), INTENT(OUT) :: attribute_val	26 27 28 29 30 31 32 33
INTEG INTEG LOGIC	GET_ATTR(COMM, COMM_KEYV EER COMM, COMM_KEYVAL, IE EER(KIND=MPI_ADDRESS_KIND CAL FLAG) ATTRIBUTE_VAL	34 35 36 37 38 39
Retric	eves attribute value by key '	The call is erroneous if there is no key with value	

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

Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the value of the attribute; the call to MPI_Comm_get_attr passes in attribute_val the address of the location where the attribute value is to be returned. Thus, if the attribute value itself is a pointer of type void*, then the actual attribute_val parameter to MPI_Comm_set_attr

		he actual attribute_val parameter to MPI_Comm_get_a
will	be of type void**. (End	of advice to users.)
void	**) avoids the messy ty	formal parameter attribute_val of type void* (rather the pe casting that would be needed if the attribute value than void*. (<i>End of rationale.</i>)
MPI_CON	/IM_DELETE_ATTR(cor	nm, comm_keyval)
INOUT	comm	communicator from which the attribute is deleted (I dle)
IN	comm_keyval	key value (integer)
C bindir	זס	
		[_Comm comm, int comm_keyval)
F08 bind	ding	
		comm_kevval, ierror)
	(MPI_Comm), INTENT(]	•
	GER, INTENT(IN) ::	
INTE	GER, OPTIONAL, INTEN	NT(OUT) :: ierror
F bindin	NG .	
	•	COMM KEYVAL, IERROR)
MPI_COMM	"g I_DELETE_ATTR(COMM, (IGER COMM, COMM_KEYVA	
MPI_COMM INTE Delet comm_de comm_de Whe MPI_COM invoked (L_DELETE_ATTR(COMM, C GGER COMM, COMM_KEYVA te attribute from cache l lete_attr_fn specified wh lete_attr_fn function ret never a communicator i MM_IDUP, all call-back in arbitrary order). Wh	AL, IERROR by key. This function invokes the attribute delete funct en the keyval was created. The call will fail if the urns an error code other than MPI_SUCCESS. s replicated using the function MPI_COMM_DUP or copy functions for attributes that are currently set enever a communicator is deleted using the function
MPI_COMM INTE Delet comm_de comm_de Whe MPI_COM invoked (MPI_COM invoked.	L_DELETE_ATTR(COMM, C GGER COMM, COMM_KEYVA te attribute from cache l lete_attr_fn specified wh lete_attr_fn function ret never a communicator i MM_IDUP, all call-back in arbitrary order). Wh	AL, IERROR by key. This function invokes the attribute delete funct en the keyval was created. The call will fail if the urns an error code other than MPI_SUCCESS. s replicated using the function MPI_COMM_DUP or copy functions for attributes that are currently set enever a communicator is deleted using the function
MPI_COMM INTE Delet comm_de comm_de Whe MPI_COM invoked (MPI_COM invoked. 6.7.3 W	L_DELETE_ATTR(COMM, C GER COMM, COMM_KEYVA te attribute from cache l lete_attr_fn specified wh lete_attr_fn function ret never a communicator i MM_IDUP , all call-back in arbitrary order). Wh MM_FREE all callback of	AL, IERROR by key. This function invokes the attribute delete funct nen the keyval was created. The call will fail if the urns an error code other than MPI_SUCCESS. s replicated using the function MPI_COMM_DUP or copy functions for attributes that are currently set nenever a communicator is deleted using the function delete functions for attributes that are currently set
MPI_COMM INTE Delet comm_de comm_de Whe MPI_COM invoked (MPI_COM invoked. 6.7.3 W	L_DELETE_ATTR(COMM, C GER COMM, COMM_KEYVA te attribute from cache l lete_attr_fn specified wh lete_attr_fn function ret never a communicator i MM_IDUP, all call-back in arbitrary order). Wh MM_FREE all callback of /indows	AL, IERROR by key. This function invokes the attribute delete funct nen the keyval was created. The call will fail if the urns an error code other than MPI_SUCCESS. s replicated using the function MPI_COMM_DUP or copy functions for attributes that are currently set nenever a communicator is deleted using the function delete functions for attributes that are currently set
MPI_COMM INTE Delet comm_de comm_de Whe MPI_COM invoked (MPI_COM invoked. 6.7.3 W The funct	L_DELETE_ATTR(COMM, C GER COMM, COMM_KEYVA te attribute from cache l lete_attr_fn specified wh lete_attr_fn function ret never a communicator i MM_IDUP, all call-back in arbitrary order). Wh MM_FREE all callback of /indows tions for caching on win	AL, IERROR by key. This function invokes the attribute delete funct en the keyval was created. The call will fail if the urns an error code other than MPI_SUCCESS. s replicated using the function MPI_COMM_DUP or copy functions for attributes that are currently set enever a communicator is deleted using the function delete functions for attributes that are currently set dows are:
MPI_COMM INTE Delet comm_de comm_de Whe MPI_COM invoked (MPI_COM invoked. 6.7.3 W The funct	L_DELETE_ATTR(COMM, C GER COMM, COMM_KEYVA te attribute from cache l lete_attr_fn specified wh lete_attr_fn function ret never a communicator i MM_IDUP, all call-back in arbitrary order). Wh MM_FREE all callback of /indows tions for caching on win	AL, IERROR by key. This function invokes the attribute delete funct en the keyval was created. The call will fail if the urns an error code other than MPI_SUCCESS. s replicated using the function MPI_COMM_DUP or copy functions for attributes that are currently set enever a communicator is deleted using the function delete functions for attributes that are currently set dows are:
MPI_COMM INTE Delet comm_de comm_de Whe MPI_COM invoked (MPI_COM invoked. 6.7.3 W The funct MPI_WIN	L_DELETE_ATTR(COMM, C GER COMM, COMM_KEYVA te attribute from cache l lete_attr_fn specified wh lete_attr_fn function ret never a communicator i MM_IDUP, all call-back in arbitrary order). Wh MM_FREE all callback of /indows tions for caching on win	AL, IERROR by key. This function invokes the attribute delete funct en the keyval was created. The call will fail if the urns an error code other than MPI_SUCCESS. s replicated using the function MPI_COMM_DUP or copy functions for attributes that are currently set enever a communicator is deleted using the function delete functions for attributes that are currently set dows are:
MPI_COMM INTE Delet comm_de comm_de Whe MPI_COM invoked (MPI_COM invoked. 6.7.3 W The funct MPI_WIN IN	I_DELETE_ATTR(COMM, C GER COMM, COMM_KEYVA te attribute from cache I lete_attr_fn specified wh lete_attr_fn function ret never a communicator i MM_IDUP, all call-back in arbitrary order). Wh MM_FREE all callback o /indows tions for caching on win I_CREATE_KEYVAL(wir win_copy_attr_fn	AL, IERROR by key. This function invokes the attribute delete funct en the keyval was created. The call will fail if the urns an error code other than MPI_SUCCESS. s replicated using the function MPI_COMM_DUP or copy functions for attributes that are currently set enever a communicator is deleted using the function delete functions for attributes that are currently set dows are: b_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_sta copy callback function for win_keyval (function)

⁴⁸ C binding

```
1
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
                                                                                     \mathbf{2}
              MPI_Win_delete_attr_function *win_delete_attr_fn,
                                                                                     3
              int *win_keyval, void *extra_state)
                                                                                     \mathbf{4}
F08 binding
                                                                                     5
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
                                                                                     6
              extra_state, ierror)
                                                                                     7
    PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
    PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
                                                                                     9
    INTEGER, INTENT(OUT) :: win_keyval
                                                                                     10
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                     11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     12
                                                                                     13
F binding
                                                                                     14
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
                                                                                     15
              EXTRA_STATE, IERROR)
                                                                                     16
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
                                                                                     17
    INTEGER WIN_KEYVAL, IERROR
                                                                                     18
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                     19
    The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or
                                                                                     20
MPI_WIN_DUP_FN from either C or Fortran. MPI_WIN_NULL_COPY_FN is a function
                                                                                     21
that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_WIN_DUP_FN is
                                                                                     22
a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in
                                                                                     23
attribute_val_out, and returns MPI_SUCCESS.
                                                                                     ^{24}
    The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN
                                                                                     25
from either C or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing,
                                                                                     26
other than returning MPI_SUCCESS.
                                                                                     27
The C callback functions are:
                                                                                     28
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
                                                                                     29
              void *extra_state, void *attribute_val_in,
                                                                                     30
              void *attribute_val_out, int *flag);
                                                                                     31
                                                                                     32
and
                                                                                     33
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
                                                                                     34
              void *attribute_val, void *extra_state);
                                                                                     35
With the mpi_f08 module, the Fortran callback functions are:
                                                                                     36
ABSTRACT INTERFACE
                                                                                     37
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
                                                                                     38
  attribute_val_in, attribute_val_out, flag, ierror)
                                                                                     39
      TYPE(MPI_Win) :: oldwin
                                                                                     40
      INTEGER :: win_keyval, ierror
                                                                                     41
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                     42
      attribute_val_out
                                                                                     43
      LOGICAL :: flag
                                                                                     44
                                                                                     45
and
                                                                                     46
ABSTRACT INTERFACE
                                                                                     47
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
                                                                                     48
```

```
1
       extra_state, ierror)
\mathbf{2}
            TYPE(MPI_Win) :: win
3
            INTEGER :: win_keyval, ierror
4
            INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
5
     With the mpi module and mpif.h, the Fortran callback functions are:
6
     SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
7
                    ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
8
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
9
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
10
                     ATTRIBUTE_VAL_OUT
11
         LOGICAL FLAG
12
13
     and
14
     SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
15
                    EXTRA_STATE, IERROR)
16
          INTEGER WIN, WIN_KEYVAL, IERROR
17
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
18
         If an attribute copy function or attribute delete function returns other than
19
     MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is
20
     erroneous.
21
22
23
     MPI_WIN_FREE_KEYVAL(win_keyval)
^{24}
       INOUT
                win_keyval
                                            key value (integer)
25
26
     C binding
27
     int MPI_Win_free_keyval(int *win_keyval)
28
29
     F08 binding
30
     MPI_Win_free_keyval(win_keyval, ierror)
^{31}
          INTEGER, INTENT(INOUT) :: win_keyval
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     F binding
     MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
35
          INTEGER WIN_KEYVAL, IERROR
36
37
38
39
     MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)
40
       INOUT
                win
                                            window to which attribute will be attached (handle)
41
                win_keyval
       IN
                                            key value (integer)
42
43
                attribute_val
       IN
                                            attribute value
44
45
     C binding
46
     int MPI_Win_set_attr(MPI_Win win, int win_keyval, void* attribute_val)
47
     F08 binding
48
```

MPT Win s	set_attr(win, win_keyval,	attribute val jerror)	1	
	TYPE(MPI_Win), INTENT(IN) :: win			
	INTEGER, INTENT(IN) :: win_keyval			
INTEC	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val			
INTEC	SER, OPTIONAL, INTENT(OUT) :: ierror	5	
F binding			6	
	SET_ATTR(WIN, WIN_KEYVAL,	ATTRIBUTE_VAL, IERROR)	7 8	
INTEC	GER WIN, WIN_KEYVAL, IERR	DR	9	
INTEC	ER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	10	
			11	
			12	
MPI_WIN	_GET_ATTR(win, win_keyval,	attribute_val, flag)	13	
IN	win	window to which the attribute is attached (handle)	14	
IN	win_keyval	key value (integer)	15 16	
OUT	attribute_val	attribute value, unless $flag = false$	10	
OUT	flag	false if no attribute is associated with the key (logical)	18	
001	iiag	laise if no attribute is associated with the key (logical)	19	
C bindin	σ		20	
		int win_keyval, void* attribute_val,	21	
_	int *flag)	_ , , _ ,	22	
F08 bind	ing		23 24	
	0	attribute_val, flag, ierror)	24 25	
	(MPI_Win), INTENT(IN) ::	-	26	
	GER, INTENT(IN) :: win_k		27	
INTEC	ER(KIND=MPI_ADDRESS_KIND)	, INTENT(OUT) :: attribute_val	28	
	CAL, INTENT(OUT) :: flag		29	
INTEC	ER, OPTIONAL, INTENT(OUT)) :: ierror	30	
F binding	g		31	
MPI_WIN_C	GET_ATTR(WIN, WIN_KEYVAL,	ATTRIBUTE_VAL, FLAG, IERROR)	32 33	
	GER WIN, WIN_KEYVAL, IERR		34	
	SER(KIND=MPI_ADDRESS_KIND)) ATTRIBUTE_VAL	35	
LOGI	CAL FLAG		36	
			37	
	DELETE ATTD(in in he	- 1)	38	
	_DELETE_ATTR(win, win_key	,	39	
INOUT	win	window from which the attribute is deleted (handle)	40	
IN	win_keyval	key value (integer)	41 42	
			42	
C bindin	g		44	
int MPI_V	Vin_delete_attr(MPI_Win w	in, int win_keyval)	45	
F08 bind	ing		46	
	lelete_attr(win, win_keyv	al, ierror)	47	
	(MPI_Win), INTENT(IN) ::	win	48	

```
1
         INTEGER, INTENT(IN) :: win_keyval
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     F binding
4
     MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
5
          INTEGER WIN, WIN_KEYVAL, IERROR
6
7
8
     6.7.4
            Datatypes
9
     The new functions for caching on datatypes are:
10
11
12
     MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval, extra_state)
13
14
       IN
                type_copy_attr_fn
                                            copy callback function for type_keyval (function)
15
16
       IN
                type_delete_attr_fn
                                            delete callback function for type_keyval (function)
17
       OUT
                type_keyval
                                            key value for future access (integer)
18
       IN
                extra_state
                                            extra state for callback function
19
20
21
     C binding
^{22}
     int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
23
                    MPI_Type_delete_attr_function *type_delete_attr_fn,
^{24}
                    int *type_keyval, void *extra_state)
25
     F08 binding
26
     MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
27
                    extra_state, ierror)
28
         PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
29
         PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
30
         INTEGER, INTENT(OUT) :: type_keyval
31
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     F binding
35
     MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
36
                    EXTRA_STATE, IERROR)
37
         EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
38
         INTEGER TYPE_KEYVAL, IERROR
39
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
40
         The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or
41
     MPI_TYPE_DUP_FN from either C or Fortran. MPI_TYPE_NULL_COPY_FN is a function
42
     that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN
43
     is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in
44
     attribute_val_out, and returns MPI_SUCCESS.
45
         The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN
46
     from either C or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing,
47
     other than returning MPI_SUCCESS.
48
```

```
The C callback functions are:
                                                                                     1
                                                                                     2
typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
                                                                                     3
              int type_keyval, void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
                                                                                     4
                                                                                     5
and
                                                                                     6
typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
                                                                                     7
              int type_keyval, void *attribute_val, void *extra_state);
                                                                                     8
                                                                                     9
With the mpi_f08 module, the Fortran callback functions are:
                                                                                     10
ABSTRACT INTERFACE
                                                                                     11
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
                                                                                     12
                                                                                     13
      TYPE(MPI_Datatype) :: oldtype
                                                                                     14
      INTEGER :: type_keyval, ierror
                                                                                     15
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                     16
      attribute_val_out
                                                                                     17
      LOGICAL :: flag
                                                                                     18
and
                                                                                     19
ABSTRACT INTERFACE
                                                                                     20
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
                                                                                     21
  attribute_val, extra_state, ierror)
                                                                                     22
      TYPE(MPI_Datatype) :: datatype
                                                                                     23
      INTEGER :: type_keyval, ierror
                                                                                     24
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                     25
                                                                                     26
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                     27
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
                                                                                     28
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                     29
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
                                                                                     30
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
                                                                                     31
               ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
                                                                                     32
    LOGICAL FLAG
                                                                                     33
and
                                                                                     34
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
                                                                                     35
              EXTRA_STATE, IERROR)
                                                                                     36
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
                                                                                     37
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                     38
                                                                                     39
    If an attribute copy function or attribute delete function returns other than
                                                                                     40
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),
                                                                                     41
is erroneous.
                                                                                     42
                                                                                     43
MPI_TYPE_FREE_KEYVAL(type_keyval)
                                                                                     44
                                                                                     45
 INOUT
          type_keyval
                                     key value (integer)
                                                                                     46
                                                                                     47
C binding
                                                                                     48
```

```
1
     int MPI_Type_free_keyval(int *type_keyval)
\mathbf{2}
     F08 binding
3
     MPI_Type_free_keyval(type_keyval, ierror)
4
          INTEGER, INTENT(INOUT) :: type_keyval
5
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
     F binding
8
     MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
9
          INTEGER TYPE_KEYVAL, IERROR
10
11
12
     MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)
13
       INOUT
                 datatype
                                             datatype to which attribute will be attached (handle)
14
15
       IN
                 type_keyval
                                            key value (integer)
16
                 attribute_val
       IN
                                            attribute value
17
18
     C binding
19
     int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
20
                    void* attribute_val)
21
22
     F08 binding
23
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
^{24}
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
          INTEGER, INTENT(IN) :: type_keyval
26
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     F binding
29
     MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
30
          INTEGER DATATYPE, TYPE_KEYVAL, IERROR
^{31}
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
32
33
34
     MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)
35
36
       IN
                 datatype
                                            datatype to which the attribute is attached (handle)
37
       IN
                 type_keyval
                                            key value (integer)
38
       OUT
                 attribute_val
                                            attribute value, unless flag = false
39
40
       OUT
                 flag
                                            false if no attribute is associated with the key (logical)
41
42
     C binding
43
     int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
44
                    void* attribute_val, int *flag)
45
     F08 binding
46
47
     MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

	<pre>INTEGER, INTENT(IN) :: type_keyval INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror inding</pre>	1 2 3 4 5
MPI_	_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL LOGICAL FLAG	6 7 8 9 10 11
MPI	_TYPE_DELETE_ATTR(datatype, type_keyval)	12 13
	OUT datatype datatype from which the attribute is deleted (handle)	14
IN		15 16
C bi	inding	17
	MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)	18 19
	binding	20
	_Type_delete_attr(datatype, type_keyval, ierror)	21
	TYPE(MPI_Datatype), INTENT(IN) :: datatype	22
	INTEGER, INTENT(IN) :: type_keyval	23
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	24
F bi	inding	25 26
	_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)	20
_	INTEGER DATATYPE, TYPE_KEYVAL, IERROR	28
		29
675	E Error Close for Invalid Kouval	30
0.7.3	5 Error Class for Invalid Keyval	31
e	values for attributes are system-allocated, by	32
	_{TYPE,COMM,WIN}_CREATE_KEYVAL. Only such values can be passed to the func-	33
	s that use key values as input arguments. In order to signal that an erroneous key value	34
	been passed to one of these functions, there is a new MPI error class: MPI_ERR_KEYVAL.	35
	an be returned by MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE, _KEYVAL_FREE, MPI_{TYPE,COMM,WIN}_DELETE_ATTR,	36 37
	_{TYPE,COMM,WIN}_SET_ATTR, MPI_{TYPE,COMM,WIN}_GET_ATTR,	38
	_{TYPE,COMM,WIN}_FREE_KEYVAL, MPI_COMM_DUP, MPI_COMM_IDUP,	39
	_COMM_DISCONNECT, and MPI_COMM_FREE. The last four are included because	40
	al is an argument to the copy and delete functions for attributes.	41
2		42
6.7.6	6 Attributes Example	43
		44
	Advice to users. This example shows how to write a collective communication operation that uses caching to be more efficient after the first call. (End of advice to	45
	users.)	46
	····· ,	47

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

```
/* Third, store gop_stuff as the attribute value */
    MPI_Comm_set_attr(comm, gop_key, gop_stuff);
  }
  /* Then, in any case, use contents of *gop_stuff
     to do the global op ... */
}
/* The following routine is called by MPI when a group is freed */
int gop_stuff_destructor(MPI_Comm comm, int keyval, void *gop_stuffP,
                         void *extra)
{
  gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
  if (keyval != gop_key) { /* abort -- programming error */ }
  /* The group's being freed removes one reference to gop_stuff */
  gop_stuff->ref_count -= 1;
  /* If no references remain, then free the storage */
  if (gop_stuff->ref_count == 0) {
    free((void *)gop_stuff);
  }
  return MPI_SUCCESS;
}
/* The following routine is called by MPI when a group is copied */
int gop_stuff_copier(MPI_Comm comm, int keyval, void *extra,
               void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
{
  gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
  gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
  if (keyval != gop_key) { /* abort -- programming error */ }
  /* The new group adds one reference to this gop_stuff */
  gop_stuff_in->ref_count += 1;
  *gop_stuff_out = gop_stuff_in;
  return MPI_SUCCESS;
}
```

6.8 Naming Objects

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

Unofficial Draft for Comment Only

1 2

3

4

5 6

7

9

10 11

12

13

 $14 \\ 15$

16 17

18

19

20 21

22

23 24

25

26

27

28 29

30

31

32

33

34

35 36

37

38

39 40 41

42 43

44

45

46

47

MPI_COMM_SET_NAME(comm, comm_name)

```
2
       INOUT
                                              communicator whose identifier is to be set (handle)
                 comm
3
       IN
                                              the character string which is remembered as the name
                 comm_name
4
                                              (string)
5
6
\overline{7}
     C binding
8
     int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
9
     F08 binding
10
     MPI_Comm_set_name(comm, comm_name, ierror)
11
          TYPE(MPI_Comm), INTENT(IN) :: comm
12
          CHARACTER(LEN=*), INTENT(IN) :: comm_name
13
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
14
15
     F binding
16
     MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
17
          INTEGER COMM, IERROR
18
          CHARACTER*(*) COMM_NAME
19
          MPI_COMM_SET_NAME allows a user to associate a name string with a communicator.
20
     The character string which is passed to MPI_COMM_SET_NAME will be saved inside the
21
     MPI library (so it can be freed by the caller immediately after the call, or allocated on the
22
     stack). Leading spaces in name are significant but trailing ones are not.
23
          MPI_COMM_SET_NAME is a local (non-collective) operation, which only affects the
^{24}
     name of the communicator as seen in the process which made the MPI_COMM_SET_NAME
25
     call. There is no requirement that the same (or any) name be assigned to a communicator
26
     in every process where it exists.
27
28
           Advice to users. Since MPI_COMM_SET_NAME is provided to help debug code, it
29
           is sensible to give the same name to a communicator in all of the processes where it
30
           exists, to avoid confusion. (End of advice to users.)
^{31}
32
          The length of the name which can be stored is limited to the value of
33
     MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C to allow for the
34
     null terminator. Attempts to put names longer than this will result in truncation of the
35
     name. MPI_MAX_OBJECT_NAME must have a value of at least 64.
36
37
           Advice to users. Under circumstances of store exhaustion an attempt to put a name
38
           of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be
39
           viewed only as a strict upper bound on the name length, not a guarantee that setting
40
           names of less than this length will always succeed. (End of advice to users.)
41
42
           Advice to implementors. Implementations which pre-allocate a fixed size space for a
43
           name should use the length of that allocation as the value of MPI_MAX_OBJECT_NAME.
44
           Implementations which allocate space for the name from the heap should still define
45
           MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate
46
           space for a string of up to this size when calling MPI_COMM_GET_NAME. (End of
47
           advice to implementors.)
48
```

MPI_COM	IM_GET_NAME(comm, comm	_name, resultlen)	1
IN	comm	communicator whose name is to be returned (handle)	2
OUT	comm_name	the name previously stored on the communicator, or	3 4
		an empty string if no such name exists (string)	4 5
OUT	resultlen	length of returned name (integer)	6
001	resulten	length of retained hame (moger)	7
C bindin	Ø		8
	0	nm, char *comm_name, int *resultlen)	9
	-	,	10
F08 bind	0	nogul+lon ionnon)	11
	_get_name(comm, comm_name) (MPI_Comm), INTENT(IN) ::		12
	ACTER(LEN=MPI_MAX_OBJECT_1	comm VAME), INTENT(OUT) :: comm_name	13
	GER, INTENT(OUT) :: resul		14
	GER, OPTIONAL, INTENT(OUT)		15 16
			17
F bindin	0		18
	_GET_NAME(COMM, COMM_NAME		19
	GER COMM, RESULTLEN, IERRO	JR	20
CHAR.	ACTER*(*) COMM_NAME		21
MPI_	COMM_GET_NAME returns t	he last name which has previously been associated	22
with the g	iven communicator. The name	e may be set and retrieved from any language. The	23
same nam	e will be returned independen	t of the language used. name should be allocated	24
	0 0	length MPI_MAX_OBJECT_NAME characters.	25
	IM_GET_NAME returns a copy	•	26
		ly stored at name[resultlen]. The value of resultlen	27
	0	CT_NAME-1. In Fortran, name is padded on the	28
		e of resultlen cannot be larger than	29
	_OBJECT_NAME.		30
		me with a communicator, or an error occurs, empty string (all spaces in Fortran, "" in C). The	31
		ve predefined names associated with them. Thus,	32 33
-		_COMM_SELF, and the communicator returned by	34
	,	_COMM_NULL) will have the default of	35
		and MPI_COMM_PARENT. The fact that the system	36
		to a communicator does not prevent the user from	37
-	_	tor; doing this removes the old name and assigns	38
the new o			39
			40
		unctions for setting and getting the name of a com-	41
mun	icator, rather than simply pro	viding a predefined attribute key for the following	42
reas	ons:		43

- 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.

44

45

46

1	•	To make the attribu	te key useful additional code to call strdup is necessary. If
2		this is not standardized	zed then users have to write it. This is extra unneeded work
3		which we can easily	
4		0	
5	•		g is not trivial to write (it will depend on details of the
6		-	system), and will not be portable. Therefore it should be in
7		the library rather th	an in user code.
8			
	(Enotemetric)	d of rationale.)	
9	1.1		
10			ove definition means that it is safe simply to print the string
11		-	_GET_NAME, as it is always a valid string even if there was
12	no n	name.	
13	Note	e that associating a n	ame with a communicator has no effect on the semantics of
14		0	(necessarily) increase the store requirement of the program,
15			saved. Therefore there is no requirement that users use these
16			mes with communicators. However debugging and profiling
17			made easier if names are associated with communicators,
18		• •	ofiler should then be able to present information in a less
19			
20	cryp	otic manner. (End of	aavice to users.)
21	The	following functions	no used for setting and setting names of determore. The
22		0	re used for setting and getting names of datatypes. The
22	constant M	MPI_MAX_OBJECT_NA	AME also applies to these names.
24			
24 25	MPI_TYP	E_SET_NAME(dataty	pe, type_name)
26		· · ·	,
27	INOUT	datatype	datatype whose identifier is to be set (handle)
28	IN	type_name	the character string which is remembered as the name
29			(string)
30	C bindin		
31		19°	
32	THC HIT	•	Datatuna datatuna const char stuna nama)
		•	Datatype datatype, const char *type_name)
33	F08 bind	Type_set_name(MPI_	Datatype datatype, const char *type_name)
33 34		Type_set_name(MPI_ ling	
	MPI_Type	Type_set_name(MPI_ ling _set_name(datatype	, type_name, ierror)
34	MPI_Type TYPE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN	r, type_name, ierror) TENT(IN) :: datatype
34 35	MPI_Type TYPE CHAR.	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE	, type_name, ierror) TENT(IN) :: datatype NT(IN) :: type_name
34 35 36	MPI_Type TYPE CHAR.	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN	, type_name, ierror) TENT(IN) :: datatype NT(IN) :: type_name
34 35 36 37	MPI_Type TYPE CHAR.	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT	, type_name, ierror) TENT(IN) :: datatype NT(IN) :: type_name
34 35 36 37 38	MPI_Type TYPE CHAR INTE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT g	, type_name, ierror) TENT(IN) :: datatype NT(IN) :: type_name
34 35 36 37 38 39	MPI_Type TYPE CHAR INTE F bindin MPI_TYPE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT g	<pre>c, type_name, ierror) TENT(IN) :: datatype ENT(IN) :: type_name ENT(OUT) :: ierror C, TYPE_NAME, IERROR)</pre>
34 35 36 37 38 39 40	MPI_Type TYPE CHAR. INTE F bindin MPI_TYPE INTE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT g _SET_NAME(DATATYPE	<pre>e, type_name, ierror) TENT(IN) :: datatype NTT(IN) :: type_name ENT(OUT) :: ierror C, TYPE_NAME, IERROR) .OR</pre>
34 35 36 37 38 39 40 41 42	MPI_Type TYPE CHAR. INTE F bindin MPI_TYPE INTE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT g _SET_NAME(DATATYPE GER DATATYPE, IERR	<pre>e, type_name, ierror) TENT(IN) :: datatype NTT(IN) :: type_name ENT(OUT) :: ierror C, TYPE_NAME, IERROR) .OR</pre>
34 35 36 37 38 39 40 41 42 43	MPI_Type TYPE CHAR. INTE F bindin MPI_TYPE INTE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT g _SET_NAME(DATATYPE GER DATATYPE, IERR	<pre>e, type_name, ierror) TENT(IN) :: datatype NTT(IN) :: type_name ENT(OUT) :: ierror C, TYPE_NAME, IERROR) .OR</pre>
34 35 36 37 38 39 40 41 42 43 44	MPI_Type TYPE CHAR. INTE F bindin MPI_TYPE INTE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT g _SET_NAME(DATATYPE GER DATATYPE, IERR	<pre>e, type_name, ierror) TENT(IN) :: datatype NTT(IN) :: type_name ENT(OUT) :: ierror C, TYPE_NAME, IERROR) .OR</pre>
34 35 36 37 38 39 40 41 42 43 44 45	MPI_Type TYPE CHAR. INTE F bindin MPI_TYPE INTE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT g _SET_NAME(DATATYPE GER DATATYPE, IERR	<pre>e, type_name, ierror) TENT(IN) :: datatype NTT(IN) :: type_name ENT(OUT) :: ierror C, TYPE_NAME, IERROR) .OR</pre>
34 35 36 37 38 39 40 41 42 43 44 45 46	MPI_Type TYPE CHAR. INTE F bindin MPI_TYPE INTE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT g _SET_NAME(DATATYPE GER DATATYPE, IERR	<pre>e, type_name, ierror) TENT(IN) :: datatype NTT(IN) :: type_name ENT(OUT) :: ierror C, TYPE_NAME, IERROR) .OR</pre>
34 35 36 37 38 39 40 41 42 43 44 45	MPI_Type TYPE CHAR. INTE F bindin MPI_TYPE INTE	Type_set_name(MPI_ ling _set_name(datatype (MPI_Datatype), IN ACTER(LEN=*), INTE GER, OPTIONAL, INT g _SET_NAME(DATATYPE GER DATATYPE, IERR	<pre>e, type_name, ierror) TENT(IN) :: datatype NTT(IN) :: type_name ENT(OUT) :: ierror C, TYPE_NAME, IERROR) .OR</pre>

MPI_TYPE	_GET_NAME(datatype, type_	name, resultlen)	1
IN	datatype	datatype whose name is to be returned (handle)	2
OUT	type_name	the name previously stored on the datatype, or an empty string if no such name exists (string)	3 4 5
OUT	resultlen	length of returned name (integer)	6 7
C binding	r		8
-		datatype, char *type_name,	9 10 11
F08 bindi	ng		12
	get_name(datatype, type_n		13
	MPI_Datatype), INTENT(IN) CTER(LEN=MPI_MAX_OBJECT_N	• =	14 15
	ER, INTENT(OUT) :: resul	• -	16
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	17
F binding			18 19
	GET_NAME(DATATYPE, TYPE_N		20
	ER DATATYPE, RESULTLEN, I CTER*(*) TYPE_NAME	ERROR	21
			22
	l predefined datatypes have the VCHAR has the default name of the	e default names of the datatype name. For exam-	23 24
		setting and getting names of windows. The con-	24 25
	MAX_OBJECT_NAME also appl		26
			27
MPI_WIN_	SET_NAME(win, win_name)		28 29
INOUT	win	window whose identifier is to be set (handle)	30
IN	win_name	the character string which is remembered as the name	31
		(string)	32
			33 34
C binding	-		34 35
int MPI_W	in_set_name(MPI_Win win,	const char *win_name)	36
F08 bindi	ng		37
	et_name(win, win_name, ie		38
	MPI_Win), INTENT(IN) ::		39
	CTER(LEN=*), INTENT(IN) : ER, OPTIONAL, INTENT(OUT)		40 41
		:: leffor	42
F binding			43
	ET_NAME(WIN, WIN_NAME, IE	RRUR)	44
	ER WIN, IERROR CTER*(*) WIN_NAME		45
UNARA	OIDUL(") WINTNULL		46
			47

MPI_WIN_GET_NAME(win, win_name, resultlen)

```
2
       IN
                                            window whose name is to be returned (handle)
                win
3
       OUT
                win_name
                                            the name previously stored on the window, or an empty
4
                                            string if no such name exists (string)
5
6
       OUT
                resultlen
                                            length of returned name (integer)
7
8
     C binding
9
     int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
10
     F08 binding
11
     MPI_Win_get_name(win, win_name, resultlen, ierror)
12
         TYPE(MPI_Win), INTENT(IN) :: win
13
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
14
         INTEGER, INTENT(OUT) :: resultlen
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     F binding
18
     MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
19
         INTEGER WIN, RESULTLEN, IERROR
20
         CHARACTER*(*) WIN_NAME
21
```

6.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

6.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that 30 communicator must be free of side effects throughout execution of the subprogram: there 31 32 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 33 34so 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 35 interfere. Since we permit good implementations to create new communicators without 36 synchronization (such as by preallocated contexts on communicators), this does not impose 37 a significant overhead. 38

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.

42 43

44

22 23

24 25

26

27 28

29

1

6.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by having each executing process invoke the procedure. The invocation is a collective operation: it is executed by all processes in the execution group, and invocations are similarly ordered at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

Static Communicator Allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

Dynamic Communicator Allocation

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

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

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

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

- messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;
- 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.

Unofficial Draft for Comment Only

10 11

1

 $\mathbf{2}$

3

4

5

6 7

8

9

12

13

14

15

16

17 18

19 20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38 39

40

43

44

45

46

47

48

1			
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			
40			
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 (2×2) grid is as follows.

coord $(0,0)$:	$\operatorname{rank} 0$
coord $(0,1)$:	rank 1
coord $(1,0)$:	rank 2
coord $(1,1)$:	$\operatorname{rank} 3$

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

7.4 Overview of the Functions

43 MPI supports three topology types: Cartesian, graph, and distributed graph. The 44function MPI_CART_CREATE is used to create Cartesian topologies, the function 45

MPI_GRAPH_CREATE is used to create graph topologies, and the functions 46

MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE are used to cre-47

ate distributed graph topologies. These topology creation functions are collective. As with 48

1

2

3

4 $\mathbf{5}$

> 6 7

> > 8

9

10

11

12

13

14

15

25

27

35 36

37

38

39

40 41

other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. For MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical values on all processes of the group of comm_old. When calling MPI_GRAPH_CREATE, each process specifies all nodes and edges in the graph. In contrast, the functions MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator **comm_topol** is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an *n*-dimensional hypercube is an *n*-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

MPI defines functions to query a communicator for topology information. The function 22MPI_TOPO_TEST is used to query for the type of topology associated with a communicator. 23Depending on the topology type, different information can be extracted. For a graph topology, the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET return the values that were specified in the call to MPI_GRAPH_CREATE. Additionally, the functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to obtain 2728the neighbors of an arbitrary node in the graph. For a distributed graph topology, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS 2930 can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI_CARTDIM_GET and MPI_CART_GET return the values that were specified in the call to MPI_CART_CREATE. Additionally, the functions MPI_CART_RANK and 33 MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. 34The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group.

The two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP, are, in general, not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation.

The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the

4546nearest neighbors on the topology associated with the communicator. The nonblocking 47variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, 48 MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 $\overline{7}$

8 9

10 11

12

13

14

15

16

17

18

1920

21

 24

25

26

31

32

35

36 37

38

39

40 41

42

43

```
324
                                                    CHAPTER 7. PROCESS TOPOLOGIES
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 (array of positive integers)
14
15
       IN
                 periods
                                              logical array of size ndims specifying whether the grid
16
                                              is periodic (true) or not (false) in each dimension (ar-
17
                                              ray of logicals)
18
       IN
                 reorder
                                              ranking may be reordered (true) or not (false) (logical)
19
       OUT
                                              communicator with new Cartesian topology (handle)
                 comm_cart
20
21
     C binding
22
     int MPI_Cart_create(MPI_Comm comm_old, const int ndims, const int dims[],
23
                     int periods[], int reorder, MPI_Comm *comm_cart)
24
25
     F08 binding
26
     MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
27
          TYPE(MPI_Comm), INTENT(IN) :: comm_old
28
          INTEGER, INTENT(IN) :: ndims, dims(ndims)
29
          LOGICAL, INTENT(IN) :: periods(ndims), reorder
30
          TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
^{31}
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
32
     F binding
33
34
     MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
          INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
35
          LOGICAL PERIODS(*), REORDER
36
37
          MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian
38
     topology information is attached. If reorder = false then the rank of each process in the
39
     new group is identical to its rank in the old group. Otherwise, the function may reorder
40
     the processes (possibly so as to choose a good embedding of the virtual topology onto
41
     the physical machine). If the total size of the Cartesian grid is smaller than the size of
42
     the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to
43
     MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created.
44
     The call is erroneous if it specifies a grid that is larger than the group size or if ndims is
45
     negative.
46
47
48
```

7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI_COMM_WORLD's group) into an <i>n</i> -dimensional topology.					
MPI_DIMS_CREATE(nnodes, ndims, dims)					
		10			
	number of nodes in a grid (integer)	11			
IN ndims	number of Cartesian dimensions (integer)	12			
INOUT dims	integer array of size ndims specifying the number of	13 14			
	nodes in each dimension (array of positive integers)	15			
		16			
C binding					
<pre>int MPI_Dims_create(int nnodes, int ndims, int dims[])</pre>					
F08 binding					
MPI_Dims_create(nnodes, ndims, dims, ierror)					
INTEGER, INTENT(IN) :: nnodes, ndims INTEGER, INTENT(INOUT) :: dims(ndims) INTEGER OPTIONAL INTENT(OUT) :: isrner					
					INTEGER, OPTIONAL, INTENT(OUT) :: ierror
F binding MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)					
					INTEGER NNODES, NDIMS, DIMS(*), IERROR
The entries in the array dims a	re set to describe a Cartesian grid with ndims dimensions	28			
and a total of $nnodes$ nodes. The d	imensions are set to be as close to each other as possible,	29			
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 cal	i. [i] are erroneous. An error will occur if nnodes is not a	33 34			
multiple of	[1] are enoneous. An error win occur in modes is not a	35			
indiopio of	$\prod dims[i].$	36			
	$i,dims[i] \neq 0$	37			
For dims[i] set by the call dim	For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array dims				
is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is local.					
If ndims is zero and nnodes is one, MPI_DIMS_CREATE returns MPI_SUCCESS.					
		41			

Example 7.1

```
function call
1
                    dims
                                                                 dims
2
                    before call
                                                                 on return
3
                    (0,0)
                                MPI_DIMS_CREATE(6, 2, dims)
                                                                 (3,2)
                                MPI_DIMS_CREATE(7, 2, dims)
4
                    (0,0)
                                                                 (7,1)
                                MPI_DIMS_CREATE(6, 3, dims)
5
                    (0,3,0)
                                                                 (2,3,1)
6
                    (0,3,0)
                                MPI_DIMS_CREATE(7, 3, dims)
                                                                 erroneous call
7
8
            Graph Constructor
     7.5.3
9
10
11
     MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)
12
       IN
                                              input communicator (handle)
                 comm_old
13
14
       IN
                 nnodes
                                              number of nodes in graph (integer)
15
       IN
                 index
                                              array of integers describing node degrees (see below)
16
                                              (array of integers)
17
       IN
                                              array of integers describing graph edges (see below)
                 edges
18
                                              (array of non-negative integers)
19
                 reorder
       IN
                                             ranking may be reordered (true) or not (false) (logical)
20
21
       OUT
                                             communicator with graph topology added (handle)
                 comm_graph
22
23
     C binding
24
     int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],
25
                     const int edges[], int reorder, MPI_Comm *comm_graph)
26
27
     F08 binding
     MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
28
29
                     ierror)
30
          TYPE(MPI_Comm), INTENT(IN) :: comm_old
^{31}
          INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
32
          LOGICAL, INTENT(IN) ::
                                     reorder
33
          TYPE(MPI_Comm), INTENT(OUT) ::
                                               comm_graph
34
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
35
     F binding
36
     MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
37
                     IERROR)
38
          INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
39
          LOGICAL REORDER
40
41
          MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph
42
     topology information is attached. If reorder = false then the rank of each process in the
43
     new group is identical to its rank in the old group. Otherwise, the function may reorder the
44
     processes. If the size, nnodes, of the graph is smaller than the size of the group of comm_old,
45
     then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE
46
     and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL
47
     is returned in all processes. The call is erroneous if it specifies a graph that is larger than
48
     the group size of the input communicator.
```

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

Example 7.2

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

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

Then, the input arguments are:

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),

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

1	
	3. periods (periodicity information),
2	4. own_position (own position in grid, could also be computed from rank and
3	dims)
4	• For a graph topology:
5	
6	1. index,
7	2. edges,
8	which are the vectors defining the graph structure
9	which are the vectors defining the graph structure.
10	For a graph structure the number of nodes is equal to the number of processes in
11	the group. Therefore, the number of nodes does not have to be stored explicitly.
12	An additional zero entry at the start of array index simplifies access to the topology
13	information. (End of advice to implementors.)
14	
15	7.5.4 Distributed Graph Constructor
16	MDL CDADH CDEATE requires that each process reason the full (richal) communication
17	MPI_GRAPH_CREATE requires that each process passes the full (global) communication
18	graph to the call. This limits the scalability of this constructor. With the distributed graph
19	interface, the communication graph is specified in a fully distributed fashion. Each process
20	specifies only the part of the communication graph of which it is aware. Typically, this
21	could be the set of processes from which the process will eventually receive or get data,
22	or the set of processes to which the process will send or put data, or some combination of
23	such edges. Two different interfaces can be used to create a distributed graph topology.
24	MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with
25	each process specifying each of its incoming and outgoing (adjacent) edges in the logical
26	communication graph and thus requires minimal communication during creation.
27	MPI_DIST_GRAPH_CREATE provides full flexibility such that any process can indicate that
	communication will occur between any pair of processes in the graph.
28	To provide better possibilities for optimization by the MPI library, the distributed
29	graph constructors permit weighted communication edges and take an info argument that
30	can further influence process reordering or other optimizations performed by the MPI library.
31	For example, hints can be provided on how edge weights are to be interpreted, the quality
32	of the reordering, and/or the time permitted for the MPI library to process the graph.
33	of the reordering, and/or the time permitted for the twitt horary to process the graph.
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
40 47	
47 48	
40	

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph)

	degree, destinations, dest	tweights, into, reorder, comm_dist_graph)	-
IN	comm_old	input communicator (handle)	3
IN	indegree	size of sources and source weights arrays (non-negative integer)	4 5 6
IN	sources	ranks of processes for which the calling process is a destination (array of non-negative integers)	7 8
IN	sourceweights	weights of the edges into the calling process (array of non-negative integers)	9 10 11
IN	outdegree	size of destinations and destweights $\operatorname{arrays}(\operatorname{non-negative})$	11 12 13
IN	destinations	ranks of processes for which the calling process is a source (array of non-negative integers)	14 15
IN	destweights	weights of the edges out of the calling process (array of non-negative integers)	16 17
IN	info	hints on optimization and interpretation of weights (handle)	18 19 20
IN	reorder	the ranks may be reordered (true) or not (false) (logical)	21 22
OUT	comm_dist_graph	communicator with distributed graph topology (han- dle)	23 24
C bindin	g		25 26 27
	0	t(MPI_Comm comm_old, int indegree,	27 28

```
C
```

```
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
             const int sources[], const int sourceweights[], int outdegree,
             const int destinations[], const int destweights[],
             MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

F08 binding

```
33
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
                                                                                 34
             outdegree, destinations, destweights, info, reorder,
                                                                                 35
             comm_dist_graph, ierror)
                                                                                 36
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                 37
    INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),
                                                                                 38
    outdegree, destinations(outdegree), destweights(*)
                                                                                 39
    TYPE(MPI_Info), INTENT(IN) :: info
    LOGICAL, INTENT(IN) :: reorder
                                                                                 41
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
                                                                                 42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 43
F binding
                                                                                 44
                                                                                 45
```

```
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
             OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
             COMM_DIST_GRAPH, IERROR)
```

1

2

29

30

 31 32

40

46

INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR LOGICAL REORDER

MPI_DIST_GRAPH_CREATE_ADJACENT returns a handle to a new communicator 5to which the distributed graph topology information is attached. Each process passes all 6 information about its incoming and outgoing edges in the virtual distributed graph topology. 7 The calling processes must ensure that each edge of the graph is described in the source 8 and in the destination process with the same weights. If there are multiple edges for a given 9 (source, dest) pair, then the sequence of the weights of these edges does not matter. The 10 complete communication topology is the combination of all edges shown in the sources arrays 11 of all processes in **comm_old**, which must be identical to the combination of all edges shown 12in the destinations arrays. Source and destination ranks must be process ranks of comm_old. 13 This allows a fully distributed specification of the communication graph. Isolated processes 14(i.e., processes with no outgoing or incoming edges, that is, processes that have specified 15indegree and outdegree as zero and thus do not occur as source or destination rank in the 16graph specification) are allowed. 17

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 22remapping strategy and other internal MPI optimizations. For instance, approximate count 23arguments of later communication calls along specific edges could be used as their edge 24weights. Multiplicity of edges can likewise indicate more intense communication between 25pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 26standard and is left to the implementation. In C or Fortran, an application can supply 27the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have 28the same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some 29 but not all processes of comm_old. If the graph is weighted but indegree or outdegree is 30 zero, then MPI_WEIGHTS_EMPTY or any arbitrary array may be passed to sourceweights 31 or destweights respectively. Note that MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are 32 not special weight values; rather they are special values for the total array argument. In 33 Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects like MPI_BOTTOM (not 34usable for initialization or assignment). See Section 2.5.4. 35

Advice to users. In the case of an empty weights array argument passed while
 constructing a weighted graph, one should not pass NULL because the value of
 MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then
 be indistinguishable from MPI_UNWEIGHTED to the implementation. In this case
 MPI_WEIGHTS_EMPTY should be used instead. (End of advice to users.)

Advice to implementors. It is recommended that MPI_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)

Rationale. To ensure backward compatibility, MPI_UNWEIGHTED may still be implemented as NULL. See Annex B.3. (*End of rationale.*)

The meaning of the info and reorder arguments is defined in the description of the following routine.

1

2

3

4

42

43 44

45

LOGICAL REORDER

MPI_DIS	T_GRAPH_CREATE(comm_old order, comm_dist_graph)	, n, sources, degrees, destinations, weights, info, re-	1 2		
IN	comm_old	input communicator (handle)	$\frac{3}{4}$		
IN	n	number of source nodes for which this process specifies edges (integer)	4 5 6		
IN	sources	array containing the ${\sf n}$ source nodes for which this process specifies edges (array of non-negative integers)	7 8		
IN	degrees	array specifying the number of destinations for each source node in the source node array (array of non- negative integers)	9 10 11 12		
IN	destinations	destination nodes for the source nodes in the source node array (array of non-negative integers)	12 13 14		
IN	weights	weights for source to destination edges (array of non- negative integers)	15 16		
IN	info	hints on optimization and interpretation of weights (handle)	17 18 19		
IN	reorder	the ranks may be reordered (true) or not (false) (logical)	20 21		
OUT	comm_dist_graph	communicator with distributed graph topology added (handle)	22 23 24		
C bindin int MPI	_Dist_graph_create(MPI_Com const int degrees[],	<pre>m comm_old, int n, const int sources[], const int destinations[], MPI_Info info, int reorder, graph)</pre>	25 26 27 28 29		
TYPI INTI	z_graph_create(comm_old, n info, reorder, comm_ E(MPI_Comm), INTENT(IN) ::	0	30 31 32 33 34 35 36		
LOG] TYPI	E(MPI_Info), INTENT(IN) :: ICAL, INTENT(IN) :: reord E(MPI_Comm), INTENT(OUT) : EGER, OPTIONAL, INTENT(OUT	er : comm_dist_graph	37 38 39 40		
<pre>F binding MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,</pre>					

Unofficial Draft for Comment Only

1 MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the $\mathbf{2}$ distributed graph topology information is attached. Concretely, each process calls the con-3 structor with a set of directed (source, destination) communication edges as described below. 4 Every process passes an array of n source nodes in the sources array. For each source node, a $\mathbf{5}$ non-negative number of destination nodes is specified in the degrees array. The destination 6 nodes are stored in the corresponding consecutive segment of the destinations array. More $\overline{7}$ precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the 8 j-th such edge stored in destinations[degrees[0]+ \dots +degrees[i-1]+j]. The weight of this edge 9 is stored in weights[degrees[0]+ \dots +degrees[i-1]+i]. Both the sources and the destinations 10 arrays may contain the same node more than once, and the order in which nodes are listed 11as destinations or sources is not significant. Similarly, different processes may specify edges 12with the same source and destination nodes. Source and destination nodes must be pro-13cess ranks of comm_old. Different processes may specify different numbers of source and 14destination nodes, as well as different source to destination edges. This allows a fully dis-15tributed specification of the communication graph. Isolated processes (i.e., processes with 16no outgoing or incoming edges, that is, processes that do not occur as source or destination 17node 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.

²² If reorder = false, all processes will have the same rank in comm_dist_graph as in ²³ comm_old. If reorder = true then the MPI library is free to remap to other processes (of ²⁴ comm_old) in order to improve communication on the edges of the communication graph. ²⁵ The weight associated with each edge is a hint to the MPI library about the amount or ²⁶ intensity of communication on that edge, and may be used to compute a "best" reordering.

27Weights are specified as non-negative integers and can be used to influence the process 28remapping strategy and other internal MPI optimizations. For instance, approximate count 29arguments of later communication calls along specific edges could be used as their edge 30 weights. Multiplicity of edges can likewise indicate more intense communication between 31 pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 32 standard and is left to the implementation. In C or Fortran, an application can supply 33 the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the 34same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some but not 35 all processes of comm_old. If the graph is weighted but n = 0, then MPI_WEIGHTS_EMPTY 36 or any arbitrary array may be passed to weights. Note that MPI_UNWEIGHTED and 37 MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the 38 total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects 39 like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

 $40 \\ 41$

42

43

44

46

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.

⁴⁵ MPI_WEIGHTS_EMPTY should be used instead. (*End of advice to users.*)

47 Advice to implementors. It is recommended that MPI_UNWEIGHTED not be imple-48 mented as NULL. (End of advice to implementors.)

Unofficial Draft for Comment Only

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.

Advice to implementors. MPI implementations must document any additionally supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI_GRAPH_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors*.)

Example 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

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

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

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

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

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24

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	$1,\!1$

10 11 12

13

14

15

16

17

1

 $\mathbf{2}$

> 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:

```
/*
19
     Input:
                 dimensions P, Q
20
     Condition: number of processes equal to P*Q; otherwise only
21
                ranks smaller than P*Q participate
22
     */
23
     int rank, x, y;
^{24}
     int sources[1], degrees[1];
25
     int destinations[8], weights[8];
26
     MPI_Comm comm_dist_graph;
27
28
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
29
30
     /* get x and y dimension */
^{31}
     y=rank/P; x=rank%P;
32
33
     /* get my communication partners along x dimension */
34
     destinations[0] = P*y+(x+1)%P; weights[0] = 2;
35
     destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
36
37
     /* get my communication partners along y dimension */
38
     destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
39
     destinations[3] = P*((Q+y-1))(Q)+x; weights[3] = 2;
40
41
     /* get my communication partners along diagonals */
42
     destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[4] = 1;
     destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
43
     destinations[6] = P*((y+1))(Q)+(P+x-1)(P); weights[6] = 1;
44
     destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
45
46
47
     sources[0] = rank;
48
     degrees [0] = 8;
```

<pre>MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,</pre>					
7.5.5 Topology Inquiry Functions ⁴					
If a topolo	If a topology has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls. 7				
	/		8 9		
MPI_TOPO_TEST(comm, status)					
IN	comm	communicator (handle)	11		
OUT	status	topology type of communicator $comm$ (integer)	12 13		
			13		
C bindin	g fopo_test(MPI_Comm comm,	int *status)	15		
	-		16		
F08 bind	5	N	17		
-	_test(comm, status, ierro (MPI_Comm), INTENT(IN) ::		18		
	GER, INTENT(OUT) :: stat		19 20		
	GER, OPTIONAL, INTENT(OUT		20		
			22		
F binding	g _TEST(COMM, STATUS, IERRO	۵)	23		
	GER COMM, STATUS, IERROR		24		
			25		
		eturns the type of topology that is assigned to a	26		
communic The c	ator. output value status is one of th	a following:	27 28		
The c	Surput value status is one of th	le following.	28 29		
MPI_GR/	APH	graph topology	30		
MPI_CART		Cartesian topology	31		
	T_GRAPH	distributed graph topology	32		
MPI_UN	DEFINED	no topology	33		
			34		
MPL CRA	PHDIMS_GET(comm, nnodes,	nedges)	35		
	× ×	C ,	36 37		
IN	comm	communicator for group with graph structure (handle)	38		
OUT	nnodes	number of nodes in graph (same as number of pro- cesses in the group) (integer)	$\frac{39}{40}$		
OUT	nedges	number of edges in graph (integer)	41		
			42		
C bindin	g		43		
int MPI_(Graphdims_get(MPI_Comm co	mm, int *nnodes, int *nedges)	44		
F08 bind	ing		45		
	ndims_get(comm, nnodes, n	edges, ierror)	46 47		
TYPE	(MPI_Comm), INTENT(IN) ::	comm	48		

	, INTENT(OUT) :: nnode	s nodros				
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
³ F binding	F binding					
5 MPI_GRAPHDI	MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR) INTEGER COMM, NNODES, NEDGES, IERROR					
 ⁸ information t ⁹ The info 	Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topology information that was associated with a communicator by MPI_GRAPH_CREATE. The information provided by MPI_GRAPHDIMS_GET can be used to dimension the vectors index and edges correctly for the following call to MPI_GRAPH_GET.					
¹² 13 MPI_GRAPH	_GET(comm, maxindex, max	edges, index, edges)				
14	comm	communicator with graph structure (handle)				
15 IN m	naxindex	length of vector index in the calling program (integer)				
16 IN n	naxedges	length of vector edges in the calling program (integer)				
¹⁸ OUT in ¹⁹ ²⁰	ndex	array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE) (array of integers)				
21 22 OUT e 23	dges	array of integers containing the graph structure (array of non-negative integers)				
 C binding int MPI_Gray 	<pre>ph_get(MPI_Comm comm, i int edges[])</pre>	<pre>nt maxindex, int maxedges, int index[],</pre>				
²⁸ F08 binding	у 5					
	MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)					
I YPE (MP)	TYPE(MPI_Comm), INTENT(IN) :: comm					
0.0	INTEGER, INTENT(IN) :: maxindex, maxedges INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)					
³³ INTEGER	, OPTIONAL, INTENT(OUT)					
$_{35}^{34}$ F binding						
		DGES, INDEX, EDGES, IERROR)				
	INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR					
38 39						
	MPI_CARTDIM_GET(comm, ndims)					
	omm	communicator with Cartesian structure (handle)				
42 43 OUT n	idims	number of dimensions of the Cartesian structure (in-				
44		teger)				
45 Chinadiana						
0	C binding int MPI_Cartdim_get(MPI_Comm comm, int *ndims)					
48						

F08 bind	ing		1
	lim_get(comm, ndims, ierro	or)	2
TYPE(MPI_Comm), INTENT(IN) ::	comm	3
INTEG	ER, INTENT(OUT) :: ndim:	S	4
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	5
F binding	r		6
-) DIM_GET(COMM, NDIMS, IERR(OR)	7
	ER COMM, NDIMS, IERROR		8 9
			9 10
		and MPI_CART_GET return the Cartesian topol-	11
00		a communicator by MPI_CART_CREATE. If comm Cartesian topology, MPI_CARTDIM_GET returns	12
		all output arguments unchanged.	13
nunns—0 a		an output arguments unchanged.	14
			15
MPI_CAR	Γ_GET(comm, maxdims, dims,	periods, coords)	16
IN	comm	communicator with Cartesian structure (handle)	17
IN	maxdims	length of vectors dims, periods, and	18
		coords in the calling program (integer)	19
OUT	dims	number of processes for each Cartesian dimension (ar-	20
001	ums	ray of positive integers)	21 22
OUT	periods	periodicity (true/false) for each Cartesian dimension	23
	•	(array of logicals)	24
OUT	coords	coordinates of calling process in Cartesian structure	25
001	600143	(array of integers)	26
			27
C binding	.		28
		nt maxdims, int dims[], int periods[],	29 30
-	int coords[])		31
F00 L J	•		32
F08 bind	0	periods, coords, ierror)	33
	MPI_Comm), INTENT(IN) ::	-	34
	ER, INTENT(IN) :: maxdir		35
		(maxdims), coords(maxdims)	36
	AL, INTENT(OUT) :: perio		37
	ER, OPTIONAL, INTENT(OUT)		38
F hinding			39
F binding	-	PERIODS, COORDS, IERROR)	40
	ER COMM, MAXDIMS, DIMS,		41
	AL PERIODS(*)		42
20010			43 44
			44
			46
			47
			48

```
1
     MPI_CART_RANK(comm, coords, rank)
2
       IN
                  comm
                                               communicator with Cartesian structure (handle)
3
       IN
                 coords
                                               integer array (of size ndims) specifying the Cartesian
4
                                               coordinates of a process (array of integers)
5
6
       OUT
                 rank
                                               rank of specified process (non-negative integer)
7
8
      C binding
9
      int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
10
     F08 binding
11
     MPI_Cart_rank(comm, coords, rank, ierror)
12
          TYPE(MPI_Comm), INTENT(IN) :: comm
13
          INTEGER, INTENT(IN) :: coords(*)
14
          INTEGER, INTENT(OUT) :: rank
15
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
      F binding
18
     MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
19
          INTEGER COMM, COORDS(*), RANK, IERROR
20
          For a process group with Cartesian structure, the function MPI_CART_RANK trans-
21
     lates the logical process coordinates to process ranks as they are used by the point-to-point
22
     routines.
23
          For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that
24
     is, coords(i) < 0 or coords(i) \ge dims(i), it is shifted back to the interval
25
      0 \leq coords(i) < dims(i) automatically. Out-of-range coordinates are erroneous for non-
26
     periodic dimensions.
27
          If comm is associated with a zero-dimensional Cartesian topology, coords is not signif-
28
      icant and 0 is returned in rank.
29
30
^{31}
      MPI_CART_COORDS(comm, rank, maxdims, coords)
32
       IN
                                               communicator with Cartesian structure (handle)
33
                  comm
34
       IN
                  rank
                                               rank of a process within group of comm (non-negative
35
                                               integer)
36
       IN
                  maxdims
                                               length of vector coords in the calling program (integer)
37
       OUT
                 coords
                                               integer array (of size maxdims) containing the Carte-
38
39
                                               sian coordinates of specified process (array of integers)
40
41
     C binding
42
      int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
43
     F08 binding
44
     MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
45
          TYPE(MPI_Comm), INTENT(IN) :: comm
46
          INTEGER, INTENT(IN) :: rank, maxdims
47
          INTEGER, INTENT(OUT) :: coords(maxdims)
48
```

INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	1
F binding			2
0	COORDS(COMM, RANK, MAXDIM	S. COORDS. IERROR)	3
	ER COMM, RANK, MAXDIMS, C		4 5
TL - :		lineter toronalation in consciled has	6
	COORDS.	dinates translation is provided by	7
		dimensional Cartesian topology,	8
	be unchanged.	diffensional cartestair topology;	9
			10
			11
MPI_GRAP	H_NEIGHBORS_COUNT(com	m, rank, nneighbors)	12
IN	comm	communicator with graph topology (handle)	13
IN	rank	rank of process in group of comm (non-negative inte-	14
		ger)	15 16
OUT	nneighbors	number of neighbors of specified process (integer)	17
	- G		18
C binding			19
0	•	Comm comm, int rank, int *nneighbors)	20
F08 bindi		-	21
	ng _neighbors_count(comm, ra	nk nneighborg jerror)	22
-	MPI_Comm), INTENT(IN) ::	comm	23
	ER, INTENT(IN) :: rank		24
	ER, INTENT(OUT) :: nneig	hbors	25 26
	ER, OPTIONAL, INTENT(OUT)		20
F binding			28
0	_NEIGHBORS_COUNT(COMM, RA	NK NNETGHBORS TERROR)	29
	ER COMM, RANK, NNEIGHBORS		30
	,,,	,	31
			32
MPI_GRAP	H_NEIGHBORS(comm, rank,	maxneighbors, neighbors)	33
IN	comm	communicator with graph topology (handle)	34
			35
IN	rank	rank of process in group of comm (non-negative inte-	36 37
		ger)	38
IN	maxneighbors	size of array neighbors (integer)	39
OUT	neighbors	ranks of processes that are neighbors to specified pro-	40
		cess (array of non-negative integers)	41
			42
C binding	-		43
int MPI_G		comm, int rank, int maxneighbors,	44
	<pre>int neighbors[])</pre>		45
F08 bindi	0		$46 \\ 47$
MPI_Graph	_neighbors(comm, rank, ma	xneighbors, neighbors, ierror)	47

1	TYPE(MPI_Comm), INTENT(IN) :: comm
2	INTEGER, INTENT(IN) :: rank, maxneighbors
3	INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
4 5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6	F binding
7	MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
8	INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
9	MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency
10	information for a graph topology. The returned count and array of neighbors for the queried
11	rank will both include <i>all</i> neighbors and reflect the same edge ordering as was specified by
12	the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT
13	and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array
14	passed to MPI_GRAPH_CREATE (for the purpose of this example, we assume that index[-1]
15	is zero):
16	
17 18	• The number of neighbors (nneighbors) returned from
19	$\label{eq:model} MPI_GRAPH_NEIGHBORS_COUNT \ \mathrm{will} \ \mathrm{be} \ (index[rank] \ \text{-} \ index[rank-1]).$
20	• The neighbors array returned from MPI_GRAPH_NEIGHBORS will be edges[index[rank-
21	1]] through edges[index[rank]-1].
22	
23	Example 7.5
24	Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix (note
25	that some neighbors are listed multiple times):
26	
27	process neighbors
28	0 1, 1, 3
29 30	1 0, 0
31	
32	3 $0, 2, 2$
33	Thus, the input arguments to MPI_GRAPH_CREATE are:
34	
35	nnodes $= 4$
36	index = 3, 5, 6, 9
37	edges = 1, 1, 3, 0, 0, 3, 0, 2, 2
38	Therefore, calling MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS for
39	each of the 4 processes will return:
40	
41 42	Input rank Count Neighbors
42	$\begin{array}{cccccccccccccccccccccccccccccccccccc$
44	$egin{array}{cccccccccccccccccccccccccccccccccccc$
45	$\begin{array}{cccccccccccccccccccccccccccccccccccc$
46	
47	
48	Example 7.6

Suppose that **comm** is a communicator with a shuffle-exchange topology. The group has 2^n members. Each process is labeled by a_1, \ldots, a_n with $a_i \in \{0, 1\}$, and has three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), shuffle $(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$, and unshuffle $(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for n = 3.

r	ıode	exchange	shuffle	unshuffle
		neighbors(1)	neighbors(2)	neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

Suppose that the communicator **comm** has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
! assume: each process has stored a real number A.
! extract neighborhood information
        CALL MPI_COMM_RANK(comm, myrank, ierr)
        CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
! perform exchange permutation
        CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0, &
        neighbors(1), 0, comm, status, ierr)
! perform shuffle permutation
        CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, &
        neighbors(3), 0, comm, status, ierr)
! perform unshuffle permutation
        CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, &
        neighbors(3), 0, comm, status, ierr)
```

MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS provide adjacency information for a distributed graph topology.

 $\mathbf{2}$

 24

	-		
1	MPI_DIST	_GRAPH_NEIGHBORS_COUN	IT(comm, indegree, outdegree, weighted)
2 3 4	IN	comm	communicator with distributed graph topology (handle)
5 6	OUT	indegree	number of edges into this process (non-negative integer)
7 8	OUT	outdegree	number of edges out of this process (non-negative in- teger)
9 10 11	OUT	weighted	false if MPI_UNWEIGHTED was supplied during creation, true otherwise (logical)
12 13 14 15	C binding int MPI_D		t(MPI_Comm comm, int *indegree, *weighted)
16 17 18 19 20 21	TYPE(INTEG LOGIC	graph_neighbors_count(con MPI_Comm), INTENT(IN) ::	gree, outdegree nted
22 23 24 25 26 27 28	INTEG		M, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) GREE, IERROR
29 30	MPI_DIST	_GRAPH_NEIGHBORS(comm, destinations, destweights)	, maxindegree, sources, sourceweights, maxoutdegree,
31 32 33	IN	comm	communicator with distributed graph topology (handle)
33 34	IN	maxindegree	size of sources and sourceweights arrays (integer)
35 36	OUT	sources	processes for which the calling process is a destination (array of non-negative integers)
37 38	OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)
39 40	IN	maxoutdegree	size of destinations and destweights arrays (integer)
41 42	OUT	destinations	processes for which the calling process is a source (array of non-negative integers)
43 44 45	OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)
46 47 48	C binding	g	

F08 binding

<pre>MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,</pre>	
maxoutdegree, destinations, destweights, ierror)	
TYPE(MPI_Comm), INTENT(IN) :: comm	
INTEGER, INTENT(IN) :: maxindegree, maxoutdegree	
<pre>INTEGER, INTENT(OUT) :: sources(maxindegree),</pre>	
destinations(maxoutdegree)	
<pre>INTEGER :: sourceweights(*), destweights(*)</pre>	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	

F binding

```
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 (potentially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays. If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then for each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in **comm_old** in the creation call. If the communicator was created with MPI_DIST_GRAPH_CREATE then the only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT, then only the first part of the full list is returned.

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (*End of advice to implementors.*)

7.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation may be used ⁴⁴ along a coordinate direction to perform a shift of data. As input, MPI_SENDRECV takes ⁴⁵ the rank of a source process for the receive, and the rank of a destination process for the ⁴⁶ send. If the function MPI_CART_SHIFT is called for a Cartesian process group, it provides ⁴⁷ the calling process with the above identifiers, which then can be passed to MPI_SENDRECV. ⁴⁸

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

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
      The user specifies the coordinate direction and the size of the step (positive or negative).
\mathbf{2}
      The function is local.
3
4
      MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)
5
6
       IN
                  comm
                                               communicator with Cartesian structure (handle)
7
       IN
                  direction
                                               coordinate dimension of shift (integer)
8
       IN
                  disp
                                               displacement (> 0: upwards shift, < 0: downwards
9
                                               shift) (positive integer)
10
11
        OUT
                  rank_source
                                               rank of source process (non-negative integer)
12
        OUT
                  rank_dest
                                               rank of destination process (non-negative integer)
13
14
      C binding
15
      int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
16
                      int *rank_source, int *rank_dest)
17
18
     F08 binding
19
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
20
          TYPE(MPI_Comm), INTENT(IN) :: comm
21
          INTEGER, INTENT(IN) :: direction, disp
22
          INTEGER, INTENT(OUT) :: rank_source, rank_dest
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
      F binding
25
     MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
26
          INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
27
28
          The direction argument indicates the coordinate dimension to be traversed by the shift.
29
      The dimensions are numbered from 0 to ndims-1, where ndims is the number of dimensions.
30
          Depending on the periodicity of the Cartesian group in the specified coordinate direc-
^{31}
      tion, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case
32
      of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest,
33
      indicating that the source or the destination for the shift is out of range.
34
          It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or
35
      greater than or equal to the number of dimensions in the Cartesian communicator. This
36
      implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with
37
      a zero-dimensional Cartesian topology.
38
39
     Example 7.7
40
          The communicator, comm, has a two-dimensional, periodic, Cartesian topology associ-
41
      ated with it. A two-dimensional array of REALs is stored one element per process, in variable
42
      A. One wishes to skew this array, by shifting column i (vertically, i.e., along the column)
43
      by i steps.
44
45
46
47
48
```

CHAPTER 7. PROCESS TOPOLOGIES

			1
! find pr	ocess rank		2
-	L MPI_COMM_RANK(comm, ran	k, ierr)	3
! find Ca	rtesian coordinates		4
CAL	L MPI_CART_COORDS(comm, r	ank, maxdims, coords, ierr)	5
! compute	shift source and destina	tion	6
CAL	L MPI_CART_SHIFT(comm, 0,	<pre>coords(2), source, dest, ierr)</pre>	7
! skew ar	0		8
CAL		1, MPI_REAL, dest, 0, source, 0, comm, &	9
	st	atus, ierr)	10
Adaria	a to users In Fortran the di	mension indicated by $DIRECTION = i$ has $DIMS(i+1)$	11 12
		t was used to create the grid. In C, the dimension	12
		ension specified by dims[i]. (<i>End of advice to users.</i>)	14
maio			15
757 Par	titioning of Cartesian Structu	ires	16
1.0.1 1 41			17
			18
MPL CART		wcomm)	19
		,	20
IN	comm	communicator with Cartesian structure (handle)	21
IN	remain_dims	the i-th entry of $remain_dims$ specifies whether the i-th	22
		dimension is kept in the subgrid (true) or is dropped	23
		(false) (array of logicals)	24
OUT	newcomm	communicator containing the subgrid that includes	25
		the calling process (handle)	26
			27 28
C binding	5		20
int MPI_C	art_sub(MPI_Comm comm, co	onst int remain_dims[], MPI_Comm *newcomm)	30
F08 bindi	ng		31
	sub(comm, remain_dims, ne	wcomm. jerror)	32
	MPI_Comm), INTENT(IN) ::		33
	AL, INTENT(IN) :: remain		34
	MPI_Comm), INTENT(OUT) ::		35
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	36
F hinding			37
F binding			38
	SUB(COMM, REMAIN_DIMS, NE ER COMM, NEWCOMM, IERROR	WCOFM, IERROR)	39
	AL REMAIN_DIMS(*)		40
			41
		eated with MPI_CART_CREATE, the function	42
		on the communicator group into subgroups that	43
		ds, and to build for each subgroup a communicator	44
	0	opology. If all entries in remain_dims are false or dimensional Cartesian topology then newcomm is	45 46

Unofficial Draft for Comment Only

```
1
     Example 7.8
\mathbf{2}
          Assume that MPI_CART_CREATE(..., comm) has defined a (2 \times 3 \times 4) grid. Let
3
     remain_dims = (true, false, true). Then a call to
4
           MPI_CART_SUB(comm, remain_dims, comm_new);
5
6
      will create three communicators each with eight processes in a 2 \times 4 Cartesian topology.
7
      If remain_dims = (false, false, true) then the call to MPI_CART_SUB(comm, remain_dims,
8
      comm_new) will create six non-overlapping communicators, each with four processes, in a
9
      one-dimensional Cartesian topology.
10
11
             Low-Level Topology Functions
     7.5.8
12
13
      The two additional functions introduced in this section can be used to implement all other
14
      topology functions. In general they will not be called by the user directly, unless he or she
15
      is creating additional virtual topology capability other than that provided by MPI. The two
16
      calls are both local.
17
18
      MPI_CART_MAP(comm, ndims, dims, periods, newrank)
19
20
        IN
                  comm
                                               input communicator (handle)
21
        IN
                  ndims
                                               number of dimensions of Cartesian structure (integer)
22
                  dims
                                               integer array of size ndims specifying the number of
        IN
23
                                               processes in each coordinate direction (array of posi-
^{24}
                                               tive integers)
25
26
        IN
                  periods
                                               logical array of size ndims specifying the periodicity
27
                                               specification in each coordinate direction (array of log-
28
                                               icals)
29
        OUT
                  newrank
                                               reordered rank of the calling process;
30
                                               MPI_UNDEFINED if calling process does not belong
^{31}
                                               to grid (non-negative integer)
32
33
     C binding
34
     int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],
35
                      const int periods[], int *newrank)
36
37
      F08 binding
38
     MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
39
          TYPE(MPI_Comm), INTENT(IN) :: comm
40
          INTEGER, INTENT(IN) :: ndims, dims(ndims)
41
          LOGICAL, INTENT(IN) :: periods(ndims)
42
          INTEGER, INTENT(OUT) :: newrank
43
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                    ierror
44
     F binding
45
     MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
46
          INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
47
          LOGICAL PERIODS(*)
48
```

MPI_CART_MAP computes an "optimal" placement for the calling process on the physical machine. A possible implementation of this function is to always return the rank of the calling process, that is, not to perform any reordering.

The function MPI_CART_CREATE(comm, ndims, dims, Advice to implementors. 5periods, reorder, comm_cart), with reorder = true can be implemented by calling 6 MPI_CART_MAP(comm, ndims, dims, periods, newrank), then calling 7 MPI_COMM_SPLIT(comm, color, key, comm_cart), with color = 0 if newrank \neq 8 MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank. If ndims 9 is zero then a zero-dimensional Cartesian topology is created. 10 11 The function MPI_CART_SUB(comm, remain_dims, comm_new) can be implemented 12by a call to MPI_COMM_SPLIT(comm, color, key, comm_new), using a single number 13 encoding of the lost dimensions as color and a single number encoding of the preserved 14dimensions as key. 15All other Cartesian topology functions can be implemented locally, using the topology 16information that is cached with the communicator. (End of advice to implementors.) 17 18 The corresponding function for graph structures is as follows. 192021MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank) 22 IN comm input communicator (handle) 23IN nnodes number of graph nodes (integer) 24 25IN index integer array specifying the graph structure, see 26lushline MPI_GRAPH_CREATE (array of integers) 27IN edges integer array specifying the graph structure (array of 28non-negative integers) 29 30 OUT newrank reordered rank of the calling process; 31MPI_UNDEFINED if the calling process does not be-32 long to graph (non-negative integer) 33 34 C binding 35 int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[], 36 const int edges[], int *newrank) 37 F08 binding 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 44F binding 45MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) 46INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR 47

1 2

3

4

Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph), with reorder = true can be implemented by calling MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_graph), with color = 0 if newrank \neq MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

7.6 Neighborhood Collective Communication on Process Topologies

¹² MPI process topologies specify a communication graph, but they implement no commu-¹³ nication function themselves. Many applications require sparse nearest neighbor commu-¹⁴ nications that can be expressed as graph topologies. We now describe several collective ¹⁵ operations that perform communication along the edges of a process topology. All of these ¹⁶ functions are collective; i.e., they must be called by all processes in the specified commu-¹⁷ nicator. See Section 5 for an overview of other dense (global) collective communication ¹⁸ operations and the semantics of collective operations.

¹⁹ If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources ²⁰ and destinations containing 0, ..., n-1, where n is the number of processes in the group ²¹ of comm_old (i.e., the graph is fully connected and also includes an edge from each node ²² to itself), then the sparse neighborhood communication routine performs the same data ²³ exchange as the corresponding dense (fully-connected) collective operation. In the case of a ²⁴ Cartesian communicator, only nearest neighbor communication is provided, corresponding ²⁵ 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.)

33 For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the se-34quence of neighbors in the send and receive buffers at each process is defined as the sequence 35 returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources, respectively. For 36 a general graph topology, created with MPI_GRAPH_CREATE, the use of neighborhood col-37 lective communication is restricted to adjacency matrices, where the number of edges be-38 tween any two processes is defined to be the same for both processes (i.e., with a symmetric 39 adjacency matrix). In this case, the order of neighbors in the send and receive buffers is 40defined as the sequence of neighbors as returned by MPI_GRAPH_NEIGHBORS. Note that 41 general graph topologies should generally be replaced by the distributed graph topologies. 42For a Cartesian topology, created with MPI_CART_CREATE, the sequence of neigh-43 bors in the send and receive buffers at each process is defined by order of the dimensions,

⁴³ bors in the send and receive buffers at each process is defined by order of the dimensions, ⁴⁴ first the neighbor in the negative direction and then in the positive direction with dis-⁴⁵ placement 1. The numbers of sources and destinations in the communication routines are ⁴⁶ **2*ndims** with ndims defined in MPI_CART_CREATE. If a neighbor does not exist, i.e., at ⁴⁷ the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., ⁴⁸ periods[...]==false), then this neighbor is defined to be MPI_PROC_NULL.

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8 9

10 11

26

27

28

29

30

31

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

7.6.1 Neighborhood Gather

In this function, each process i gathers data items from each process j if an edge (j, i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i, j) exists. The send buffer is sent to each neighboring process and the *l*-th block in the receive buffer is received from the *l*-th neighbor.

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

	commy		15
IN	sendbuf	starting address of send buffer (choice)	16
IN	sendcount	number of elements sent to each neighbor (non-negative	17
		integer)	18
IN	sendtype	data type of send buffer elements (handle)	19
0.U.T		·-	20
OUT	recvbuf	starting address of receive buffer (choice)	21
IN	recvcount	number of elements received from each neighbor (non-	22
		negative integer)	23
IN	recvtype	data type of receive buffer elements (handle)	24
IIN	recvtype	data type of receive buller elements (nandle)	25
IN	comm	communicator with topology structure (handle)	26

C binding

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

F08 binding

MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, 34 recvtype, comm, ierror) 35 TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf 36 INTEGER, INTENT(IN) :: sendcount, recvcount 37 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 38 TYPE(*), DIMENSION(...) :: recvbuf 39 TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42

F binding

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

1

2

3

4

56

7

8

9

10

11 1213

14

27

28

29

30

3132

33

40

43

44

45

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);
6
     int *srcs=(int*)malloc(indegree*sizeof(int));
7
     int *dsts=(int*)malloc(outdegree*sizeof(int));
8
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
9
                                outdegree, dsts, MPI_UNWEIGHTED);
10
     int k,l;
11
12
     /* assume sendbuf and recvbuf are of type (char*) */
13
     for(k=0; k<outdegree; ++k)</pre>
14
       MPI_Isend(sendbuf, sendcount, sendtype,dsts[k],...);
15
16
     for(l=0; l<indegree; ++l)</pre>
17
       MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
18
                  srcs[1],...);
19
20
```

MPI_Waitall(...);

Figure 7.1 shows the neighborhood gather communication of one process with outgoing neighbors $d_0 \ldots d_3$ and incoming neighbors $s_0 \ldots s_5$. The process will send its sendbuf to all four destinations (outgoing neighbors) and it will receive the contribution from all six sources (incoming neighbors) into separate locations of its receive buffer.

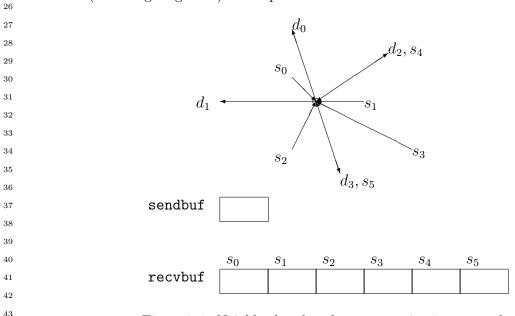


Figure 7.1: Neighborhood gather communication example.

⁴⁵ All arguments are significant on all processes and the argument comm must have iden ⁴⁶ tical values on all processes.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at all other processes. This implies

5

21 22

23

 24

25

that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

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

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

The vector variant of $\mathsf{MPI}_\mathsf{NEIGHBOR}_\mathsf{ALLGATHER}$ allows one to gather different numbers of elements from each neighbor.

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

)		14
IN	sendbuf	starting address of send buffer (choice)	15
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	$16 \\ 17$
IN	sendtype	data type of send buffer elements (handle)	18 19
OUT	recvbuf	starting address of receive buffer (choice)	20
IN	recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from each neighbor (array of non-negative integers)	21 22 23
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)	24 25 26
IN	recvtype	data type of receive buffer elements (handle)	27 28
IN	comm	communicator with topology structure (handle)	29

C binding

F08 binding

```
36
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                 37
             displs, recvtype, comm, ierror)
                                                                                 38
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 39
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
                                                                                 40
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 41
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 44
F binding
                                                                                 45
MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
                                                                                 46
                                                                                 47
```

```
DISPLS, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*)
```

1

 $\mathbf{2}$

3

4

5

6 7

8

9

10 11 12

13

30

31

32

33

34 35

```
1
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
\mathbf{2}
          IERROR
3
         This function supports Cartesian communicators, graph communicators, and distributed
4
     graph communicators as described in Section 7.6. If comm is a distributed graph commu-
5
     nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
6
     and receives from each of its incoming neighbors:
7
8
     MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
9
     int *srcs=(int*)malloc(indegree*sizeof(int));
10
     int *dsts=(int*)malloc(outdegree*sizeof(int));
11
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
12
                                 outdegree, dsts, MPI_UNWEIGHTED);
13
     int k,l;
14
15
     /* assume sendbuf and recvbuf are of type (char*) */
16
     for(k=0; k<outdegree; ++k)</pre>
17
       MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],...);
18
19
     for(l=0; l<indegree; ++1)</pre>
20
       MPI_Irecv(recvbuf+displs[1]*extent(recvtype), recvcounts[1], recvtype,
21
                   srcs[1],...);
22
23
```

```
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[l] = = 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 l-th neighbor is placed into recvbuf beginning at offset displs[l] elements (in terms of the recvtype).

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

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

35 Neighbor Alltoall 7.6.2 36

37 In this function, each process i receives data items from each process j if an edge (j,i)38 exists in the topology graph or Cartesian topology. Similarly, each process i sends data 39 items to all processes j where an edge (i, j) exists. This call is more general than

40MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor. 41 The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in 42the receive buffer is received from the *l*-th neighbor.

43 44

2425

26

27

28

29

30

 31

32

33

- 45
- 46
- 47
- 48

MPI_NE	IGHBOR_ALLTOALL(se	endbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	4 5 6
IN	sendtype	data type of send buffer elements (handle)	7
OUT	recvbuf	starting address of receive buffer (choice)	8
IN	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
C bindi int MPI F08 bin	_Neighbor_alltoall(MPI_Datatype MPI_Datatype	const void *sendbuf, int sendcount, sendtype, void *recvbuf, int recvcount, recvtype, MPI_Comm comm)	15 16 17 18 19 20
	0	buf, sendcount, sendtype, recvbuf, recvcount,	21
	recvtype, com	m, ierror)	22
		, INTENT(IN) :: sendbuf	23 24
		sendcount, recvcount	24 25
	E(MPI_Datatype), IN E(*), DIMENSION()	<pre>TENT(IN) :: sendtype, recvtype</pre>	26
	E(MPI_Comm), INTENT		27
	EGER, OPTIONAL, INT		28
F bindi	ng		29 30
	•	BUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, M. IERROR)	31 32
<ty< td=""><td>pe> SENDBUF(*), REC</td><td></td><td>33</td></ty<>	pe> SENDBUF(*), REC		33
INT	EGER SENDCOUNT, SEND	DTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	34
graph contractor,	ommunicators as describ	tesian communicators, graph communicators, and distributed bed in Section 7.6. If comm is a distributed graph commu- ch process executed sends to each of its outgoing neighbors coming neighbors:	35 36 37 38 39
MPI_Dis	t_graph_neighbors c	ount(comm, &indegree, &outdegree, &weighted);	40
	cs=(int*)malloc(inde		41
		<pre>degree*sizeof(int));</pre>	42
MPI_Dis		omm, indegree, srcs, MPI_UNWEIGHTED,	43 44
int k,l		utdegree, dsts, MPI_UNWEIGHTED);	44 45
1110 A,I	,		46
	me sendbuf and recv ; k <outdegree; ++k)<="" td=""><td>buf are of type (char*) */</td><td>47 48</td></outdegree;>	buf are of type (char*) */	47 48

Unofficial Draft for Comment Only

```
1
        MPI_Isend(sendbuf+k*sendcount*extent(sendtype), sendcount, sendtype,
\mathbf{2}
                     dsts[k],...);
3
4
      for(l=0; l<indegree; ++1)</pre>
5
        MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
6
                     srcs[1],...);
7
8
     MPI_Waitall(...);
9
10
          The type signature associated with sendcount, sendtype, at a process must be equal to
      the type signature associated with recvcount, recvtype at any other process. This implies
11
      that the amount of data sent must be equal to the amount of data received, pairwise between
12
      every pair of communicating processes. Distinct type maps between sender and receiver are
13
      still allowed.
14
          The "in place" option is not meaningful for this operation.
15
16
           All arguments are significant on all processes and the argument comm must have iden-
17
      tical values on all processes.
          The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different
18
      numbers of elements to and from each neighbor.
19
20
21
      MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
22
                      rdispls, recvtype, comm)
23
^{24}
        IN
                  sendbuf
                                                 starting address of send buffer (choice)
25
                  sendcounts
        IN
                                                 non-negative integer array (of length outdegree) speci-
26
                                                 fying the number of elements to send to each neighbor
27
                                                 (array of non-negative integers)
28
        IN
                  sdispls
                                                 integer array (of length outdegree). Entry j specifies
29
                                                 the displacement (relative to sendbuf) from which to
30
                                                 send the outgoing data to neighbor j (array of integers)
^{31}
        IN
                  sendtype
                                                 data type of send buffer elements (handle)
32
33
        OUT
                  recvbuf
                                                 starting address of receive buffer (choice)
34
        IN
                                                 non-negative integer array (of length indegree) speci-
                   recvcounts
35
                                                 fying the number of elements that are received from
36
                                                 each neighbor (array of non-negative integers)
37
        IN
                   rdispls
                                                 integer array (of length indegree). Entry i specifies the
38
39
                                                 displacement (relative to recvbuf) at which to place the
40
                                                 incoming data from neighbor i (array of integers)
41
        IN
                                                 data type of receive buffer elements (handle)
                   recvtype
42
                                                 communicator with topology structure (handle)
        IN
                  comm
43
44
      C binding
45
      int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[],
46
                      const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
47
48
```

```
1
              const int recvcounts[], const int rdispls[],
                                                                                     2
              MPI_Datatype recvtype, MPI_Comm comm)
                                                                                     3
F08 binding
                                                                                     4
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                     5
              recvcounts, rdispls, recvtype, comm, ierror)
                                                                                     6
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                     7
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                     8
    rdispls(*)
                                                                                     9
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                     10
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                     11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     13
                                                                                     14
F binding
                                                                                     15
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
                                                                                     16
              RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
                                                                                     17
    <type> SENDBUF(*), RECVBUF(*)
                                                                                     18
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                     19
    RECVTYPE, COMM, IERROR
                                                                                     20
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                     21
graph communicators as described in Section 7.6. If comm is a distributed graph commu-
                                                                                     22
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
                                                                                     23
and receives from each of its incoming neighbors:
                                                                                     24
                                                                                     25
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
                                                                                     26
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                     27
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                     28
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                                                                     29
                           outdegree, dsts, MPI_UNWEIGHTED);
                                                                                     30
int k,l;
                                                                                     31
```

```
MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process ⁴³ i must be equal to the type signature associated with recvcounts[l], recvtype with srcs[l]==i ⁴⁴ at process j. This implies that the amount of data sent must be equal to the amount of ⁴⁵ data received, pairwise between every pair of communicating processes. Distinct type maps ⁴⁶ between sender and receiver are still allowed. The data in the sendbuf beginning at offset ⁴⁷ sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data ⁴⁸

Unofficial Draft for Comment Only

32

33

34

35

36 37

38

39

40 41

1		om the l-th incoming neighbor in terms of the recvtype).	r is placed into $recvbuf$ beginning at offset $rdispls[l]$
3	```	in place" option is not meaning	ngful for this operation.
4			processes and the argument comm must have iden-
5	tical values	s on all processes.	
6	MPI_N	NEIGHBOR_ALLTOALLW allow	ws one to send and receive with different datatypes
7	to and from	n each neighbor.	
8			
9 10 11	MPI_NEIG	HBOR_ALLTOALLW(sendbuf, rdispls, recvtypes, comm)	sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
12	IN	sendbuf	starting address of send buffer (choice)
13 14 15	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
16 17 18 19	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)
20	IN	sendtypes	array of datatypes (of length outdegree). Entry j spec-
21		schutypes	ifies the type of data to send to neighbor j (array of
22			handles)
23 24	OUT	recvbuf	starting address of receive buffer (choice)
25 26 27	IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor
28 29 30 31	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)
32 33 34 35	IN	recvtypes	array of datatypes (of length indegree). Entry i spec- ifies the type of data received from neighbor i (array of handles)
36	IN	comm	communicator with topology structure (handle)
37			
38	C binding	р Э	
39	int MPI_N	0	<pre>void* sendbuf, const int sendcounts[],</pre>
40		-	<pre>ls[], const MPI_Datatype sendtypes[],</pre>
41		void* recvbuf, const	-
42 43		const MP1_Aint rdisp MPI_Comm comm)	<pre>ls[], const MPI_Datatype recvtypes[],</pre>
43			
45	F08 bindi	5	
46	MPI_Neigh		endcounts, sdispls, sendtypes, recvbuf,
47		recvcounts, rdispls, *), DIMENSION(), INTEN	recvtypes, comm, ierror) T(IN) :: sendbuf
48	IIFE(···, DIRENSTUN(), INTEN.	

```
1
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                         \mathbf{2}
    INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
                                                                                         3
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                         4
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                         5
                                                                                         6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                         7
F binding
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                                                                                         9
               RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
                                                                                         10
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         11
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
                                                                                         12
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
                                                                                         13
                IERROR
                                                                                         14
                                                                                         15
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                         16
graph communicators as described in Section 7.6. If comm is a distributed graph commu-
                                                                                         17
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
                                                                                         18
and receives from each of its incoming neighbors:
                                                                                         19
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
                                                                                        20
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                        21
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                         22
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                                                                         23
                            outdegree, dsts, MPI_UNWEIGHTED);
                                                                                         24
int k,l;
                                                                                         25
                                                                                         26
/* assume sendbuf and recvbuf are of type (char*) */
                                                                                         27
for(k=0; k<outdegree; ++k)</pre>
                                                                                         28
  MPI_Isend(sendbuf+sdispls[k], sendcounts[k], sendtypes[k], dsts[k],...);
                                                                                         29
                                                                                         30
for(l=0; l<indegree; ++1)</pre>
                                                                                         31
  MPI_Irecv(recvbuf+rdispls[1], recvcounts[1], recvtypes[1], srcs[1],...);
                                                                                         32
                                                                                         33
MPI_Waitall(...);
                                                                                        34
                                                                                        35
    The type signature associated with sendcounts[k], sendtypes[k] with dsts[k]==j at pro-
                                                                                        36
cess i must be equal to the type signature associated with recvcounts[1], recvtypes[1] with
                                                                                        37
srcs[I] == i at process j. This implies that the amount of data sent must be equal to the
                                                                                         38
amount of data received, pairwise between every pair of communicating processes. Distinct
                                                                                         39
type maps between sender and receiver are still allowed.
                                                                                         40
    The "in place" option is not meaningful for this operation.
                                                                                         41
    All arguments are significant on all processes and the argument comm must have iden-
                                                                                         42
tical values on all processes.
                                                                                         43
                                                                                         44
```

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

Unofficial Draft for Comment Only

45

 $46 \\ 47$

7.7.1 N	onblocking Neighborh	ood Gather
MPI_INE	IGHBOR_ALLGATHER comm, request)	(sendbuf, sendcount, sendtype, recvbuf, recvcount, recv
IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-neighbor) integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	number of elements received from each neighbor negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
OUT	request	communication request (handle)
C bindin int MPI_ F08 bind	Ineighbor_allgather MPI_Datatype MPI_Datatype	c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount,
int MPI_ F08 bin MPI_Inei	Ineighbor_allgather MPI_Datatype MPI_Datatype ding .ghbor_allgather(ser recvtype, com	c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request ndbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror)
int MPI_ F08 bin MPI_Inei TYPE	Ineighbor_allgather MPI_Datatype MPI_Datatype ding ghbor_allgather(ser recvtype, com E(*), DIMENSION()	c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request adbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf
int MPI_ F08 bind MPI_Inei TYPE TYPE	Ineighbor_allgathen MPI_Datatype MPI_Datatype ding ghbor_allgather(sen recvtype, com E(*), DIMENSION(); E(*), DIMENSION();	c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request ndbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror)
int MPI_ F08 bin MPI_Inei TYPE TYPE INTE TYPE	Ineighbor_allgathen MPI_Datatype MPI_Datatype ghbor_allgather(sen recvtype, com E(*), DIMENSION() E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INT	<pre>c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request adbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf sendcount, recvcount TENT(IN) :: sendtype, recvtype</pre>
int MPI F08 bind MPI_Inei TYPE TYPE INTE TYPE TYPE	Ineighbor_allgather MPI_Datatype MPI_Datatype ding .ghbor_allgather(ser recvtype, com E(*), DIMENSION() E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INTENT E(MPI_Comm), INTENT	c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request adbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf sendcount, recvcount TENT(IN) :: sendtype, recvtype (IN) :: comm
int MPI F08 bind MPI_Inei TYPE TYPE INTE TYPE TYPE TYPE	Ineighbor_allgathen MPI_Datatype MPI_Datatype ghbor_allgather(sen recvtype, com E(*), DIMENSION() E(*), DIMENSION() EGER, INTENT(IN) :: E(MPI_Datatype), INT	<pre>c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request adbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf sendcount, recvcount TENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: request</pre>
int MPI_ F08 bind MPI_Inei TYPE TYPE TYPE TYPE TYPE INTE F bindin	Ineighbor_allgathen MPI_Datatype MPI_Datatype ding .ghbor_allgather(sen recvtype, com E(*), DIMENSION(); E(*), DIMENSION(); E(MPI_Comm), INTENT(*); E(MPI_Comm), INTENT(*); E(MPI_Request), INTENT(*); E(*), OPTIONAL, INTENT(*); E(*), DIMENSION(); E(*), DIMENSION(); E(MPI_Comm), INTENT(*); E(MPI_Request), INTENT(*); E(*), OPTIONAL, INTENT	<pre>c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request adbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf sendcount, recvcount TENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: request ENT(OUT) :: ierror</pre>
int MPI_ F08 bind MPI_Inei TYPE TYPE INTE TYPE INTE F bindin MPI_INEJ <typ< td=""><td><pre>Ineighbor_allgathen MPI_Datatype MPI_Datatype ding .ghbor_allgather(sen recvtype, com E(*), DIMENSION(), E(*), DIMENSION(), INTENT E(*), DIMENSION(), DIMENSION(), DIMENSION(), DIMENSION(), DIMENSION()</pre></td><td><pre>c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request adbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf sendcount, recvcount TENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: request ENT(OUT) :: ierror NDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUN M, REQUEST, IERROR) /BUF(*)</pre></td></typ<>	<pre>Ineighbor_allgathen MPI_Datatype MPI_Datatype ding .ghbor_allgather(sen recvtype, com E(*), DIMENSION(), E(*), DIMENSION(), INTENT E(*), DIMENSION(), DIMENSION(), DIMENSION(), DIMENSION(), DIMENSION()</pre>	<pre>c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request adbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf sendcount, recvcount TENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: request ENT(OUT) :: ierror NDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUN M, REQUEST, IERROR) /BUF(*)</pre>
int MPI_ F08 bind MPI_Inei TYPE TYPE TYPE TYPE TYPE INTE F bindin MPI_INEJ <typ INTE</typ 	Ineighbor_allgathen MPI_Datatype MPI_Datatype ding .ghbor_allgather(sen recvtype, com E(*), DIMENSION(), E(*), DIMENSION(<pre>c(const void* sendbuf, int sendcount, sendtype, void* recvbuf, int recvcount, recvtype, MPI_Comm comm, MPI_Request *request ndbuf, sendcount, sendtype, recvbuf, recvcoun m, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf sendcount, recvcount TENT(IN) :: sendtype, recvtype (IN) :: comm ENT(OUT) :: request ENT(OUT) :: ierror NDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUN M, REQUEST, IERROR)</pre>

CHAPTER 7. PROCESS TOPOLOGIES

MPI_INE	IGHBOR_ALLGATHER recvtype, comm,	/(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, request)	1 2			
IN	sendbuf	starting address of send buffer (choice)	3			
IN	sendcount	number of elements sent to each neighbor (non-negative	4 5			
		integer)	6			
IN	sendtype	data type of send buffer elements (handle)	7			
OUT	recvbuf	starting address of receive buffer (choice)	8 9			
IN	recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from each neighbor	10 11 12			
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	13 14 15			
IN	recvtype	data type of receive buffer elements (handle)	16 17			
IN	comm	communicator with topology structure (handle)	18			
OUT	request	communication request (handle)	19			
F08 bin	0	-	25 26 27			
	0	endbuf, sendcount, sendtype, recvbuf, recvcounts,				
III 1_1IIC.	• •	ype, comm, request, ierror)	28 29			
TYP		INTENT(IN), ASYNCHRONOUS :: sendbuf	30			
	E(*), DIMENSION(),		31			
	EGER, INTENT(IN) ::		32			
		YNCHRONOUS :: recvcounts(*), displs(*) ENT(IN) :: sendtype, recvtype	33			
	E(MPI_Comm), INTENT(34 35			
	E(MPI_Request), INTE		36			
INT	EGER, OPTIONAL, INTE	NT(OUT) :: ierror	37			
F bindi	ng		38			
	•	NDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,	39			
		YPE, COMM, REQUEST, IERROR)	40 41			
• •	pe> SENDBUF(*), RECV		42			
TNT	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR					
			44			
This	s call starts a nonblocki	ng variant of MPI_NEIGHBOR_ALLGATHERV.	45			
			46 47			

	360	CHAPTER 7. PROCESS TOPOLOGIES				
1 2 3	7.7.2 Nonblocking Neighborhood Alltoall					
4 5	MPI_INEI	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)			
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	C binding int MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) F08 binding MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: sendtype, recvtype TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVEOUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVTYPE, COMM, REQUEST, IERROR This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALL.</type>					
46 47 48						

$MPI_INEIGHBOR_ALLTOALLV(sendbuf, \ sendcounts, \ sdispls, \ sendtype, \ recvbuf, \ recvcounts, \ \ ^1$						
	rdispls, recvtype, comm,	request)	2 3			
IN	sendbuf	starting address of send buffer (choice)	4			
IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor	5			
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j	7 8 9			
IN	sendtype	data type of send buffer elements (handle)	10			
OUT	recvbuf	starting address of receive buffer (choice)	11 12			
IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor	13 14 15			
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	16 17 18 19			
IN	recvtype	data type of receive buffer elements (handle)	20			
IN	comm	communicator with topology structure (handle)	21			
OUT request		communication request (handle)	22 23 24			
C binding int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)						
F08 bind	ling		30			
MPI_Inei	-	sendcounts, sdispls, sendtype, recvbuf,	31			
TVDE	recvcounts, rdispls, (*), DIMENSION(), INTEN	recvtype, comm, request, ierror)	32 33			
	(*), DIMENSION(), INTEN (*), DIMENSION(), ASYNC	-	34			
		NOUS :: sendcounts(*), sdispls(*),	35			
recvcounts(*), rdispls(*)						
TYPE	(MPI_Datatype), INTENT(IN	-	37			
	TYPE(MPI_Comm), INTENT(IN) :: comm					
TYPE(MPI_Request), INTENT(OUT) :: request 33						
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4						

```
F binding
```

```
      42

      MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
      43

      <type> SENDBUF(*), RECVBUF(*)
      44

      INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
RECVTYPE, COMM, REQUEST, IERROR
      45
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.

41

1 2						
3	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	C binding 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[].					
36		MPI_Comm comm, MPI_Request *request)				
37 38	F08 binding					
39	III I_IIIEI8	<pre>MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request, ierror)</pre>				
40	TYPE(TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf				
41	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf					
42 43		INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)				
44		<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*), rdispls(*)</pre>				
45	TYPE(TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),				
46	recvtypes(*)					
47 48	TYPE(MPI_Comm), INTENT(IN) :: comm					
010	TYPE(MPI_Request), INTENT(OUT) :: request					

Unofficial Draft for Comment Only

INTE	GER, OPTIONAL, INTENT(OUT) :: ierror	1
<tyr INTE INTE</tyr 	GHBOR_ALLTOALLW(SENDBU RECVCOUNTS, RDISH De> SENDBUF(*), RECVBUF GER(KIND=MPI_ADDRESS_K GER SENDCOUNTS(*), SEN REQUEST, IERROR	F, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, PLS, RECVTYPES, COMM, REQUEST, IERROR) (*) IND) SDISPLS(*), RDISPLS(*) DTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, ariant of MPI_NEIGHBOR_ALLTOALLW.	2 3 4 5 6 7 8 9 10 11
7.8 Pe	ersistent Neighborhood	d Communication on Process Topologies	12 13
mance be similar to	enefits for programs with r	nood collective operations can offer significant perfor- epetitive communication patterns. The semantics are tions as described in Section 5.13. ther	14 15 16 17 18 19 20
MPI_NEI	GHBOR_ALLGATHER_INIT recvtype, comm, info	(sendbuf, sendcount, sendtype, recvbuf, recvcount, , request)	20 21 22 23
IN	sendbuf	starting address of send buffer (choice)	24
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	25 26 27
IN	sendtype	data type of send buffer elements (handle)	28
OUT	recvbuf	starting address of receive buffer (choice)	29
IN	recvcount	number of elements received from each neighbor (non- negative integer)	30 31 32
IN	recvtype	data type of receive buffer elements (handle)	33
IN	comm	communicator with topology structure (handle)	34
IN	info	info argument (handle)	35
OUT	request	communication request (handle)	36 37 38
	C binding int MPI_Neighbor_allgather_init(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)		
F08 binding MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf			

1 2 3 4 5 6 7 8 9 10 11 12 13 14	INTEC TYPE TYPE TYPE INTEC F binding MPI_NEIGH	HBOR_ALLGATHER_INIT(SENDB RECVCOUNT, RECVTYPE, > SENDBUF(*), RECVBUF(*)	<pre>ount, recvcount) :: sendtype, recvtype comm info) :: request) :: ierror UF, SENDCOUNT, SENDTYPE, RECVBUF, , COMM, INFO, REQUEST, IERROR)</pre>	
15 16 17 18	operation.	-	munication request for the neighborhood allgather	
19 20	MPI_NEIG	displs, recvtype, comm, i	endbuf, sendcount, sendtype, recvbuf, recvcounts, nfo, request)	
21	IN	sendbuf	starting address of send buffer (choice)	
22 23 24	IN	sendcount	number of elements sent to each neighbor (non-negative integer)	
25	IN	sendtype	data type of send buffer elements (handle)	
26	OUT	recvbuf	starting address of receive buffer (choice)	
27 28 29 30	IN	recvcounts	non-negative integer array (of length indegree) con- taining the number of elements that are received from each neighbor	
31 32 33	IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	
34	IN	recvtype	data type of receive buffer elements (handle)	
35 36	IN	comm	communicator with topology structure (handle)	
37	IN	info	info argument (handle)	
38 39	OUT	request	communication request (handle)	
40 41 42 43 44 45 46 47	<pre>int MPI_Neighbor_allgatherv_init(const void* sendbuf, int sendcount,</pre>			
48		recocounts, aispis,	recvtype, comm, info, request, ierror)	

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 1 $\mathbf{2}$ TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 3 INTEGER, INTENT(IN) :: sendcount 4 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*) TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 56 TYPE(MPI_Comm), INTENT(IN) :: comm 7 TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10 F binding 11 MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 12RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 13 <type> SENDBUF(*), RECVBUF(*) 14INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 15INFO, REQUEST, IERROR 1617Creates a persistent collective communication request for the neighborhood allgathery 18

7.8.2 Persistent Neighborhood Alltoall

MPI_NEIGHBOR_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)	26
IN	sendcount	number of elements sent to each neighbor (non-negative	27
		integer)	28
IN	sendtype	data type of send buffer elements (handle)	29
Ουτ	recvbuf	starting address of receive buffer (choice)	30 31
			32
IN	recvcount	number of elements received from each neighbor (non- negative integer)	33
IN	recvtype	data type of receive buffer elements (handle)	34
	recvtype		35
IN	comm	communicator with topology structure (handle)	36
IN	info	info argument (handle)	37
Ουτ	request	communication request (handle)	38
001	i cquest	communication request (nandic)	39

C binding

operation.

<pre>int MPI_Neighbor_alltoall_init(const void* sendbuf, int sendcount,</pre>
<pre>MPI_Datatype sendtype, void* recvbuf, int recvcount,</pre>
MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
MPI_Request *request)
F08 binding
<pre>MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,</pre>

recvcount, recvtype, comm, info, request, ierror)

Unofficial Draft for Comment Only

19 20

21 22 23

24

25

40

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	TYPE INTE TYPE TYPE TYPE INTE F binding MPI_NEIG	<pre>(*), DIMENSION(), ASY GER, INTENT(IN) :: sen (MPI_Datatype), INTENT() (MPI_Comm), INTENT(IN) (MPI_Info), INTENT(IN) (MPI_Request), INTENT(O) GER, OPTIONAL, INTENT(O) g HBOR_ALLTOALL_INIT(SEND) RECVCOUNT, RECVTYP e> SENDBUF(*), RECVBUF(;</pre>	dcount, recvcount IN) :: sendtype, recvtype :: comm :: info UT) :: request UT) :: ierror BUF, SENDCOUNT, SENDTYPE, RECVBUF, E, COMM, INFO, REQUEST, IERROR)
16		-	ommunication request for the neighborhood alltoall
17 18	operation.		
19			
20	MPI_NEIG		endbuf, sendcounts, sdispls, sendtype, recvbuf,
21 22			vtype, comm, info, request)
22	IN	sendbuf	starting address of send buffer (choice)
24 25	IN	sendcounts	non-negative integer array (of length outdegree) speci- fying the number of elements to send to each neighbor
26 27 28	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j
29	IN	sendtype	data type of send buffer elements (handle)
$30 \\ 31$	OUT	recvbuf	starting address of receive buffer (choice)
32 33 34	IN	recvcounts	non-negative integer array (of length indegree) speci- fying the number of elements that are received from each neighbor
35 36 37	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
38 39	IN	recvtype	data type of receive buffer elements (handle)
40	IN	comm	communicator with topology structure (handle)
41	IN	info	info argument (handle)
42	OUT	request	communication request (handle)
43 44			
45	C bindin	-	
46	int MPI_N	Neighbor_alltoallv_init	
47			<pre>ts[], const int sdispls[], ype, void* recvbuf, const int recvcounts[],</pre>
48		in i_Datatype Sellut	JPC, VOIGH TECHDAI, CONSt IND TECHCOUNDEL],

<pre>const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MDI_Info_infoMDI_Boguest_traguest)</pre>	1 2
MPI_Info info, MPI_Request *request)	3
F08 binding	4
<pre>MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,</pre>	5
recvbuf, recvcounts, rdispls, recvtype, comm, info, request,	6
ierror)	7
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	8
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	9
<pre>INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),</pre>	9 10
recvcounts(*), rdispls(*)	
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	11
TYPE(MPI_Comm), INTENT(IN) :: comm	12
TYPE(MPI_Info), INTENT(IN) :: info	13
TYPE(MPI_Request), INTENT(OUT) :: request	14
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	15
	16
F binding	17
MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,	18
RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,	19
IERROR)	20
<type> SENDBUF(*), RECVBUF(*)</type>	21
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	22
RECVTYPE, COMM, INFO, REQUEST, IERROR	23
	24
Creates a persistent collective communication request for the neighborhood alltoally operation.	25
operation.	26
	27
	28
	29
	30
	31
	32
	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47

$\frac{1}{2}$	MPI_NEIG	•	dbuf, sendcounts, sdispls, sendtypes, recvbuf, rpes, comm, info, request)	
3	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	IN	info	info argument (handle)	
29 30	OUT	request	communication request (handle)	
 31 32 33 34 35 36 37 38 32 	C binding int MPI_Neighbor_alltoallw_init(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_Info info,			
 39 40 41 42 43 44 45 46 47 48 	<pre>F08 binding MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,</pre>			

```
TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
              recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) ::
                                   comm
    TYPE(MPI_Info), INTENT(IN) ::
                                   info
    TYPE(MPI_Request), INTENT(OUT) ::
                                       request
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                       ierror
F binding
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
             RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
              INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the neighborhood all toallw operation.

7.9 An Application Example

Example 7.9 The example in Figures 7.2-7.5 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine **relax**.

In each relaxation step each process computes new values for the solution grid function at the points u(1:100,1:100) owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the newly calculated values in u(1,1:100) must be sent into the halo cells u(101,1:100) of the left-hand neighbor with coordinates (own_coord(1)-1,own_coord(2)).

1

 $\mathbf{2}$

3

4

5

6

7

9

10

11

12

13

14

15 16

17

18 19

20 21 22

23

 24

25

26

27

28

29

30

31

32

```
1
2
3
4
5
6
7
     INTEGER ndims, num_neigh
8
     LOGICAL reorder
9
    PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
10
     INTEGER comm, comm_size, comm_cart, dims(ndims), ierr
11
     INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
12
    LOGICAL periods(ndims)
13
    REAL u(0:101,0:101), f(0:101,0:101)
14
    DATA dims / ndims * 0 /
15
     comm = MPI_COMM_WORLD
16
     CALL MPI_COMM_SIZE(comm, comm_size, ierr)
17
         Set process grid size and periodicity
     I.
18
     CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
19
     periods(1) = .TRUE.
20
     periods(2) = .TRUE.
21
         Create a grid structure in WORLD group and inquire about own position
     !
22
     CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
23
                           comm_cart, ierr)
24
    CALL MPI_CART_GET(comm_cart, ndims, dims, periods, own_coords, ierr)
25
     i = own_coords(1)
26
     j = own_coords(2)
27
     ! Look up the ranks for the neighbors. Own process coordinates are (i,j).
28
     ! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
29
     CALL MPI_CART_SHIFT(comm_cart, 0,1, neigh_rank(1), neigh_rank(2), ierr)
30
     CALL MPI_CART_SHIFT(comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
31
     ! Initialize the grid functions and start the iteration
32
     CALL init(u, f)
33
    DO it=1,100
34
        CALL relax(u, f)
35
            Exchange data with neighbor processes
     !
36
        CALL exchange(u, comm_cart, neigh_rank, num_neigh)
37
     END DO
38
     CALL output(u)
39
40
41
        Figure 7.2: Set-up of process structure for two-dimensional parallel Poisson solver.
42
43
44
45
46
47
48
```

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
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)
sndbuf(1:100,4) = u(1:100,100)
CALL MPI_NEIGHBOR_ALLTOALL(sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
                           comm_cart, ierr)
! instead of
! DO i=1,num_neigh
    CALL MPI_IRECV(rcvbuf(1,i), 100, MPI_REAL, neigh_rank(i),..., &
!
                   rq(2*i-1), ierr)
!
    CALL MPI_ISEND(sndbuf(1,i), 100, MPI_REAL, neigh_rank(i),..., &
!
                   rq(2*i ), ierr)
!
! END DO
! CALL MPI_WAITALL(2*num_neigh, rq, statuses, ierr)
u( 0,1:100) = rcvbuf(1:100,1)
u(101,1:100) = rcvbuf(1:100,2)
u(1:100, 0) = rcvbuf(1:100,3)
u(1:100,101) = rcvbuf(1:100,4)
END
```

Figure 7.3: Communication routine with local data copying and sparse neighborhood all-to-all.

Unofficial Draft for Comment Only

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

2 3 4 56 SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh) 7 IMPLICIT NONE 8 USE MPI 9 REAL u(0:101,0:101) 10 INTEGER comm_cart, num_neigh, neigh_rank(num_neigh) 11 INTEGER sndcounts(num_neigh), sndtypes(num_neigh) 12INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh) INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal 13 INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh) 14 INTEGER type_vec, ierr 15 ! The following initialization need to be done only once 16! before the first call of exchange. 17CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr) 18 CALL MPI_TYPE_VECTOR(100, 1, 102, MPI_REAL, type_vec, ierr) 19CALL MPI_TYPE_COMMIT(type_vec, ierr) sndtypes(1:2) = type_vec 20sndcounts(1:2) = 121sndtypes(3:4) = MPI_REAL 22 sndcounts(3:4) = 10023 rcvtypes = sndtypes 24rcvcounts = 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)sdispls(3) = (1 + 1*102) * size of real ! first element of u(1:100, 1)27) sdispls(4) = (1 + 100*102) * sizeofreal ! first element of u(1:100,100) 28 rdispls(1) = (0 + 1*102) * sizeofreal ! first element of u(0 , 1:100) 29 rdispls(2) = (101 + 1*102) * sizeofreal ! first element of u(101 1:100)30 rdispls(3) = (1 + 0*102) * size of real ! first element of u(1:100, 0)) 31 rdispls(4) = (1 + 101*102) * sizeofreal ! first element of u(1:100,101) 32 ! the following communication has to be done in each call of exchange 33 CALL MPI_NEIGHBOR_ALLTOALLW(u, sndcounts, sdispls, sndtypes, & 34 u, rcvcounts, rdispls, rcvtypes, & comm_cart, ierr) 35! The following finalizing need to be done only once 36 ! after the last call of exchange. 37 CALL MPI_TYPE_FREE(type_vec, ierr) 38 END 39 4041Figure 7.4: Communication routine with sparse neighborhood all-to-all-w and without local 42data copying. 43 44 45

1

```
INTEGER ndims, num_neigh
                                                                                    1
LOGICAL reorder
                                                                                    2
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
                                                                                    3
INTEGER comm, comm_size, comm_cart, dims(ndims), it, ierr
                                                                                    4
LOGICAL periods(ndims)
                                                                                    5
REAL u(0:101,0:101), f(0:101,0:101)
                                                                                    6
DATA dims / ndims * 0 /
                                                                                    7
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
                                                                                    8
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
                                                                                    9
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
                                                                                    10
INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
                                                                                    11
INTEGER type_vec, request, status
                                                                                    12
comm = MPI_COMM_WORLD
                                                                                    13
CALL MPI_COMM_SIZE(comm, comm_size, ierr)
                                                                                    14
    Set process grid size and periodicity
                                                                                    15
CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
                                                                                    16
periods(1) = .TRUE.
                                                                                    17
periods(2) = .TRUE.
                                                                                    18
    Create a grid structure in WORLD group
!
                                                                                    19
CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
                                                                                   20
                      comm_cart, ierr)
                                                                                   21
! Create datatypes for the neighborhood communication
                                                                                   22
!
                                                                                   23
! Insert code from example in Figure 7.4 to create and initialize
                                                                                   24
! sndcounts, sdispls, sndtypes, rcvcounts, rdispls, and rcvtypes
                                                                                   25
Ţ
                                                                                    26
! Initialize the neighborhood all-to-all-w operation
                                                                                   27
CALL MPI_NEIGHBOR_ALLTOALLW_INIT(u, sndcounts, sdispls, sndtypes, &
                                                                                   28
                                   u, rcvcounts, rdispls, rcvtypes, &
                                                                                   29
                                   comm_cart, info, request, ierr)
                                                                                   30
! Initialize the grid functions and start the iteration
                                                                                    31
CALL init(u, f)
                                                                                    32
DO it=1,100
                                                                                    33
       Start data exchange with neighbor processes
!
                                                                                   34
   CALL MPI_START(request, ierr)
                                                                                   35
       Compute inner cells
ļ
                                                                                   36
   CALL relax_inner (u, f)
                                                                                   37
!
       Check on completion of neighbor exchange
                                                                                    38
   CALL MPI_WAIT(request, status, ierr)
                                                                                    39
       Compute edge cells
!
                                                                                    40
   CALL relax_edges(u, f)
                                                                                    41
END DO
                                                                                    42
CALL output(u)
                                                                                    43
CALL MPI_REQUEST_FREE(request, ierr)
                                                                                    44
CALL MPI_TYPE_FREE(type_vec, ierr)
                                                                                    45
                                                                                    46
```

Figure 7.5: Two-dimensional parallel Poisson solver with persistent sparse neighborhood all-to-all-w and without local data copying.

47

Chapter 8

MPI Environmental Management

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

6

9

15

16

17

1819

2021

22

23

 24 25

26

2728

29

30

Implementation Information 8.1

8.1.1 Version Inquiries

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

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

#define MPI_VERSION 3 #define MPI_SUBVERSION 1

in Fortran,	31
in rortran,	
INTEGER :: MPI_VERSION, MPI_SUBVERSION	33
PARAMETER (MPI_VERSION = 3)	34
PARAMETER (MPI_SUBVERSION = 1)	35
The mustice of the section of the se	36
For runtime determination,	37
	38
MPI_GET_VERSION(version, subversion)	39
	40
OUTversionversion number (integer)	41
OUTsubversionsubversion number (integer)	42
	43
C binding	44
int MPI_Get_version(int *version, int *subversion)	45
	46
F08 binding	47
MPI_Get_version(version, subversion, ierror)	48

1		version subversion			
2	INTEGER, INTENT(OUT) :: version, subversion INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
3					
4	F binding				
5 6	MPI_GET_VERSION(VERSION, SUBV INTEGER VERSION, SUBVERS	-			
7 8 9 10 11	MPI_GET_VERSION can be called before MPI_INIT and after MPI_FINALIZE. This function must always be thread-safe, as defined in Section 12.4. Valid (MPI_VERSION, MPI_SUBVERSION) pairs in this and previous versions of the MPI standard are $(3,1)$, $(3,0)$, $(2,2)$, $(2,1)$, $(2,0)$, and $(1,2)$.				
12 13	MPI_GET_LIBRARY_VERSION(ve	rsion, resultlen)			
14	OUT version	version number (string)			
15	OUT resultlen	Length (in printable characters) of the result returned			
16	oor resulten	in version (integer)			
17					
18 19	C binding				
20	8	char *version, int *resultlen)			
21	F08 binding				
22	MPI_Get_library_version(vers:	ion, resultlen, ierror)			
23	·	BRARY_VERSION_STRING), INTENT(OUT) :: version			
24	INTEGER, INTENT(OUT) :: resultlen				
25	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
26 27	F binding				
28	MPI_GET_LIBRARY_VERSION(VERS	ION, RESULTLEN, IERROR)			
29	CHARACTER*(*) VERSION				
30	INTEGER RESULTLEN, IERROR				
31	This routine returns a string	representing the version of the MPI library. The version			
32	argument is a character string for				
33					
34	Advice to implementors. An implementation of MPI should return a different string				
35 36	for every change to its source code or build that could be visible to the user. (End of				
37	advice to implementors.)				
38	The argument version must re-	epresent storage that is			
39	8	ING characters long. MPI_GET_LIBRARY_VERSION may			
40	write up to this many characters i	nto version.			
41		ally written is returned in the output argument, resultlen.			
42		y stored at version[resultlen]. The value of resultlen cannot			
43	0	Z-VERSION_STRING - 1. In Fortran, version is padded on			
44 45	the right with blank characters. MPI_MAX_LIBRARY_VERSION_STR	The value of resultien cannot be larger than			
46		N can be called before MPI_INIT and after			
		TA CALL DE CALLER DELOTE IVIT I_ITATI ALLE ALLEL			
47	MPI_FINALIZE. This function mus	st always be thread-safe, as defined in Section 12.4.			

8.1.2 Environmental Inquiries ¹
A set of attributes that describe the execution environment are attached to the communi- cator MPI_COMM_WORLD when MPI is initialized. The values of these attributes can be inquired by using the function MPI_COMM_GET_ATTR described in Section 6.7 and in Section 17.2.7. It is erroneous to delete these attributes, free their keys, or change their values.
The list of predefined attribute keys include 8
MPI_TAG_UB Upper bound for tag value. 9
MPI_HOST Host process rank, if such exists, MPI_PROC_NULL, otherwise.
MPI_IO rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter. 12 13 14
MPI_WTIME_IS_GLOBAL Boolean variable that indicates whether clocks are synchronized.
Vendors may add implementation-specific parameters (such as node number, real memory size, virtual memory size, etc.) 17 ory size, virtual memory size, etc.) 18 These predefined attributes do not change value between MPI initialization (MPI_INIT) 19 and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users. 20 Advice to users. 21 Advice to users. 21
is a <i>pointer</i> to an int containing the requested value. (<i>End of advice to users.</i>) 23
The required parameter values are discussed in more detail below: 25
Tag Values 27
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 at least 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.28 29 30 30 31 32 33 33 34The attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.34
Host Rank
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. 41
IO Rank 43
The value returned for MPI_IO is the rank of a processor that can provide language-standard I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., OPEN, REWIND, WRITE). For C, this means that all of the ISO C I/O operations are supported (e.g., fopen, fprintf, lseek).

1 If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE $\mathbf{2}$ will be returned. Otherwise, if the calling process can provide language-standard I/O, 3 then its rank will be returned. Otherwise, if some process can provide language-standard 4 I/O then the rank of one such process will be returned. The same value need not be $\mathbf{5}$ returned by all processes. If no process can provide language-standard I/O, then the value 6 MPI_PROC_NULL will be returned. 7 Advice to users. Note that input is not collective, and this attribute does not indicate 8 9 which process can or does provide input. (End of advice to users.) 10 11Clock Synchronization 12The value returned for MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in 13 MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered 14 synchronized if explicit effort has been taken to synchronize them. The expectation is that 15 the variation in time, as measured by calls to MPI_WTIME, will be less then one half the 16 round-trip time for an MPI message of length zero. If time is measured at a process just 17before a send and at another process just after a matching receive, the second time should 18 be always higher than the first one. 19 The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not 20synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This 21attribute may be associated with communicators other then MPI_COMM_WORLD. 22 The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of 23MPI_COMM_WORLD. 2425Inquire Processor Name 262728MPI_GET_PROCESSOR_NAME(name, resultlen) 2930 OUT name A unique specifier for the actual (as opposed to vir- 31 tual) node. (string) 32 OUT resultlen Length (in printable characters) of the result returned 33 in name (integer) 34 35 C binding 36 int MPI_Get_processor_name(char *name, int *resultlen) 37 38 F08 binding 39 MPI_Get_processor_name(name, resultlen, ierror) 40 CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name 41 INTEGER, INTENT(OUT) :: resultlen 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 F binding 44MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR) 45CHARACTER*(*) NAME 46 INTEGER RESULTLEN, IERROR 4748

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

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

This function allows MPI implementations that do process migration to Rationale. 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-	35
		ger)	36
		0)	37
IN	info	info argument (handle)	38
OUT	baseptr	pointer to beginning of memory segment allocated	39

C binding

int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

F08 binding

MPI_Alloc_mem(size, info, baseptr, ierror) 45USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 46INTEGER(KIND=MPI_ADDRESS_KIND, INTENT(IN) :: size 47TYPE(MPI_Info), INTENT(IN) :: info 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 5

6 $\overline{7}$

8 9

10 11

12

13

14

15

16 17

18

19

20

2122 23

 24 25

26

27

28

29

30

31

32 33

34

42

43

```
1
         TYPE(C_PTR), INTENT(OUT) :: baseptr
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     F binding
4
     MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
5
          INTEGER(KIND=MPI_ADDRESS_KIND SIZE
6
         INTEGER INFO, IERROR
7
         INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR
8
9
         If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
10
     be provided in the mpi module and should be provided in mpif.h through overloading,
11
     i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
12
     BASEPTR, but with a different specific procedure name:
13
14
     INTERFACE MPI_ALLOC_MEM
15
         SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
16
              IMPORT :: MPI_ADDRESS_KIND
17
              INTEGER INFO, IERROR
18
              INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
19
         END SUBROUTINE
         SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
20
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
21
22
              IMPORT :: MPI_ADDRESS_KIND
23
              INTEGER :: INFO, IERROR
^{24}
              INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
25
              TYPE(C_PTR) :: BASEPTR
26
         END SUBROUTINE
27
     END INTERFACE
28
         The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The
29
     implied specific procedure names are described in Section 17.1.5.
30
         The info argument can be used to provide directives that control the desired location
^{31}
     of the allocated memory. Such a directive does not affect the semantics of the call. Valid
32
     info values are implementation-dependent; a null directive value of info = MPI_INFO_NULL
33
34
     is always valid.
         The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
35
     to indicate it failed because memory is exhausted.
36
37
38
     MPI_FREE_MEM(base)
39
       IN
                 base
                                            initial address of memory segment allocated by
40
                                            MPI_ALLOC_MEM (choice)
41
42
43
     C binding
44
     int MPI_Free_mem(void *base)
45
     F08 binding
46
     MPI_Free_mem(base, ierror)
47
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: base
48
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 F binding 3 MPI_FREE_MEM(BASE, IERROR) 4 <type> BASE(*) 5 INTEGER IERROR 6 $\overline{7}$ The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to 8 indicate an invalid base argument. 9 *Rationale.* The C bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar 10 to the bindings for the malloc and free C library calls: a call to 11 MPI_Alloc_mem(..., &base) should be paired with a call to MPI_Free_mem(base) (one 12less level of indirection). Both arguments are declared to be of same type 13 void^{*} so as to facilitate type casting. The Fortran binding is consistent with the C 14bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the TYPE(C_PTR) 1516pointer 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 17stored at that location. (End of rationale.) 18 19 Advice to implementors. If MPI_ALLOC_MEM allocates special memory, then a 20design similar to the design of C malloc and free functions has to be used, in order 21to find out the size of a memory segment, when the segment is freed. If no special 22 memory is used, MPI_ALLOC_MEM simply invokes malloc, and MPI_FREE_MEM 23invokes free. 2425A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate mem-26ory in a shared memory segment. (End of advice to implementors.) 2728 Example of use of MPI_ALLOC_MEM, in Fortran with Example 8.1 29 TYPE(C_PTR) pointers. We assume 4-byte REALs. 30 31(not guaranteed with INCLUDE 'mpif.h') USE mpi_f08 ! or USE mpi 32 USE, INTRINSIC :: ISO_C_BINDING 33 TYPE(C_PTR) :: p 34 REAL, DIMENSION(:,:), POINTER :: a ! no memory is allocated 35 INTEGER, DIMENSION(2) :: shape 36 INTEGER(KIND=MPI_ADDRESS_KIND) :: size 37 shape = (/100, 100/)38 ! assuming 4 bytes per REAL size = 4 * shape(1) * shape(2)39 CALL MPI_Alloc_mem(size, MPI_INFO_NULL, p, ierr) ! memory is allocated and 40 41

```
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.

Unofficial Draft for Comment Only

42

43 44

 $45 \\ 46$

```
1
       REAL A
2
                                     ! no memory is allocated
       POINTER (P, A(100,100))
3
       INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
4
       SIZE = 4*100*100
5
       CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
6
        ! memory is allocated
7
        . . .
8
       A(3,5) = 2.71;
9
10
       CALL MPI_FREE_MEM(A, IERR) ! memory is freed
11
          This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this
12
     code or need a special option, e.g., the GNU gFortran compiler needs -fcray-pointer.
13
14
           Advice to implementors. Some compilers map Cray-pointers to address-sized integers,
15
           some to TYPE(C_PTR) pointers (e.g., Cray Fortran, version 7.3.3). From the user's
16
           viewpoint, this mapping is irrelevant because Examples 8.2 should work correctly
17
           with an MPI-3.0 (or later) library if Cray-pointers are available. (End of advice to
18
           implementors.)
19
20
21
     Example 8.3 Same example, in C.
22
       float (* f)[100][100];
23
24
       /* no memory is allocated */
       MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
25
26
       /* memory allocated */
27
        . . .
        (*f)[5][3] = 2.71;
28
29
30
       MPI_Free_mem(f);
^{31}
32
     8.3
            Error Handling
33
```

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

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as *may* be handled. More background information about how MPI treats errors can be found in Section 2.8.

⁴¹ A user can associate error handlers to three types of objects: communicators, windows, ⁴² and files. The specified error handling routine will be used for any MPI exception that occurs ⁴³ during a call to MPI for the respective object. MPI calls that are not related to any objects ⁴⁴ are considered to be attached to the communicator MPI_COMM_SELF. The attachment of ⁴⁵ error handlers to objects is purely local: different processes may attach different error ⁴⁶ handlers to corresponding objects.

```
<sup>47</sup> Several predefined error handlers are available in MPI:
```

48

34

35

36

- **MPI_ERRORS_ARE_FATAL** The handler, when called, causes the program to abort all connected processes. This is similar to calling **MPI_ABORT** using a communicator containing all connected processes with an implementation-specific value as the **errorcode** argument.
- MPI_ERRORS_ABORT The handler, when called, is invoked on a communicator in a manner similar to calling MPI_ABORT on that communicator. If the error handler is invoked on an window or a file, it is similar to calling MPI_ABORT using a communicator containing the group of MPI processes associated with the window or file, respectively. In either case, the value that would be provided as the errorcode argument to MPI_ABORT is implementation-specific.
- **MPI_ERRORS_RETURN** The handler has no effect other than returning the error code to the user.

Advice to implementors. The implementation-specific error information resulting from MPI_ERRORS_ARE_FATAL and MPI_ERRORS_ABORT provided to the invoking environment should be meaningful to the end-user, for example a predefined error class. (End of advice to implementors.)

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

After initialization, the error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by

MPI_COMM_GET_PARENT (if any). Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non-trivial MPI error handler.

After an error is detected, MPI will provide the user as much information as possible about that error using error classes. Some errors might prevent MPI from completing further API calls successfully and those functions will continue to report errors until the cause of the error is corrected or the user terminates the application. The user can make the determination of whether or not to attempt to continue after detecting such an error.

Advice to users. For example, users may be unable to correct errors corresponding to some error classes, such as MPI_ERR_INTERN. Such errors may cause subsequent MPI calls to complete in error. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors and available recovery actions. (End of advice to implementors.)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test

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
     which error handler is associated with an object. C has distinct typedefs for user defined
\mathbf{2}
     error handling callback functions that accept communicator, file, and window arguments.
3
     In Fortran there are three user routines.
4
         An error handler object is created by a call to MPI_XXX_CREATE_ERRHANDLER,
5
     where XXX is, respectively, COMM, WIN, or FILE.
6
         An error handler is attached to a communicator, window, or file by a call to
7
     MPI_XXX_SET_ERRHANDLER. The error handler must be either a predefined error han-
8
     dler, or an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER,
9
     with matching XXX. The predefined error handlers MPI_ERRORS_RETURN and
10
     MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, and files.
11
         The error handler currently associated with a communicator, window, or file can be
12
     retrieved by a call to MPI_XXX_GET_ERRHANDLER.
13
         The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that
14
     was created by a call to MPI_XXX_CREATE_ERRHANDLER.
15
         MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler ob-
16
     ject is created. That is, once the error handler is no longer needed,
17
     MPI_ERRHANDLER_FREE should be called with the error handler returned from
18
     MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation.
19
     This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.
20
          Advice to implementors. High-quality implementations should raise an error when
21
          an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER is
22
          attached to an object of the wrong type with a call to MPI_YYY_SET_ERRHANDLER.
23
          To do so, it is necessary to maintain, with each error handler, information on the
24
          typedef of the associated user function. (End of advice to implementors.)
25
26
         The syntax for these calls is given below.
27
28
     8.3.1 Error Handlers for Communicators
29
     JMS See comment in IAT_FX
30
^{31}
32
33
     MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)
34
       IN
                comm_errhandler_fn
                                            user defined error handling procedure (function)
35
       OUT
                errhandler
                                            MPI error handler (handle)
36
37
38
     C binding
39
     int
40
                    MPI_Comm_create_errhandler(MPI_Comm_errhandler_function *comm_errhandler_fn,
41
                    MPI_Errhandler *errhandler)
42
     F08 binding
43
     MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
44
         PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
45
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
F binding
```

MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL COMM_ERRHANDLER_FN INTEGER ERRHANDLER, IERROR

Creates an error handler that can be attached to communicators.

The user routine should be, in C, a function of type $\mathsf{MPI_Comm_errhandler_function},$ which is defined as

```
typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use. The second is the error code to be returned by the MPI routine that raised the error. If the routine would have returned MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused the error handler to be invoked. The remaining arguments are "varargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran. With the Fortran mpi_f08 module, the user routine comm_errhandler_fn should be of the form:

ABSTRACT INTERFACE

```
SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
   TYPE(MPI_Comm) :: comm
   INTEGER :: error_code
```

With the Fortran mpi module and mpif.h, the user routine COMM_ERRHANDLER_FN should be of the form: SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)

INTEGER COMM, ERROR_CODE

Rationale. The variable argument list is provided because it provides an ISOstandard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments. (*End of rationale.*)

Advice to users. A newly created communicator inherits the error handler that is associated with the "parent" communicator. In particular, the user can specify a "global" error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization. (End of advice to users.)

MPI_COMM_SET_ERRHANDLER(comm, errhandler)			38
	M_SET_ERRHANDLER(COMM	i, ermandier)	39
INOUT	comm	communicator (handle)	40
IN	errhandler	new error handler for communicator (handle)	41
			42
C binding			43
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)			44
Int MF1_Comm_Set_effnandter(MF1_Comm comm, MF1_Effnandter effnandter)			45
F08 binding			46
MPI_Comm_	MPI_Comm_set_errhandler(comm, errhandler, ierror)		
TYPE(MPI_Comm), INTENT(IN) :: comm			48

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

1		(MPI_Errhandler), INTENT() EER, OPTIONAL, INTENT(OUT		
3) lellor	
45	F binding MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR) INTEGER COMM, ERRHANDLER, IERROR			
7 8 9 10	a predefin		communicator. The error handler must be either r handler created by a call to	
11 12	MPI_COMM_GET_ERRHANDLER(comm, errhandler)			
13	IN	comm	communicator (handle)	
14 15 16	OUT	errhandler	error handler currently associated with communicator (handle)	
17 18 19	C binding int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)			
20 21 22 23 24 25	F08 binding MPI_Comm_get_errhandler(comm, errhandler, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
26 27 28	F binding MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)			
29 30 31 32 33	For ex for a comr	xample, a library function may	y associated with a communicator. register at its entry point the current error handler e error handler for this communicator, and restore r.	
34 35	8.3.2 Error Handlers for Windows			
36 37 38	JMS	See comment	in IAT_EX	
39	MPI_WIN_	_CREATE_ERRHANDLER(win	_errhandler_fn, errhandler)	
40	IN	win_errhandler_fn	user defined error handling procedure (function)	
41 42	OUT	errhandler	MPI error handler (handle)	
43 44	C binding	g		
45 46 47 48	int	MPI_Win_create_errha MPI_Errhandler *errh	ndler(MPI_Win_errhandler_function *win_errhandler_fn, andler)	

			1
F08 bind			2
		n_errhandler_fn, errhandler, ierror)	3
		dler_function) :: win_errhandler_fn	4
	-	TENT(OUT) :: errhandler	5
INTE	GER, OPTIONAL, INTEN	1(001) :: 1error	6
F binding	g		7
MPI_WIN_(CREATE_ERRHANDLER(WI	N_ERRHANDLER_FN, ERRHANDLER, IERROR)	8
EXTE	RNAL WIN_ERRHANDLER_	FN	9
INTE	GER ERRHANDLER, IERR	OR	10
Croat	os an orror handlor tha	t can be attached to a window object. The user routine	11
		Ū Ū	12
should be, in C, a function of type MPI_Win_errhandler_function which is defined as typedef void MPI_Win_errhandler_function(MPI_Win *, int *,);			13
cypeder	voiu Mri_win_erinanu	Ter_runction(Mrr_win *, int *,),	14
The f	irst argument is the win	dow in use, the second is the error code to be returned.	15
With the l	Fortran mpi_f08 module	e, the user routine win_errhandler_fn should be of the form:	16
ABSTRACT	INTERFACE		17
SUBROU	<pre>FINE MPI_Win_errhand</pre>	<pre>ler_function(win, error_code)</pre>	18
TYI	PE(MPI_Win) :: win		19
IN	TEGER :: error_code		20
With the l	Fortran mpi module and	<pre>mpif.h, the user routine WIN_ERRHANDLER_FN should</pre>	21
be of the f	-		22
		NCTION(WIN, ERROR_CODE)	23
	GER WIN, ERROR_CODE		24
			25
			26
MPI W/IN	_SET_ERRHANDLER(w	in errhandler)	27
		,	28
INOUT	win	window (handle)	29
IN	errhandler	new error handler for window (handle)	30
			31
C bindin	g		32
int MPI_N	Win_set_errhandler(M	PI_Win win, MPI_Errhandler errhandler)	33
	•		34
F08 bind	0		35
	set_errhandler(win,		36
	(MPI_Win), INTENT(IN		37
		TENT(IN) :: errhandler	38
	GER, OPTIONAL, INTEN		39
F binding	g		40
MPI_WIN_S	SET ERRHANDLER (WIN	ERRHANDLER, IERROR)	41 42
INTE			
	GER WIN, ERRHANDLER,	IERROR	
Attac	GER WIN, ERRHANDLER,		43
	GER WIN, ERRHANDLER,	r to a window. The error handler must be either a pre-	43 44
defined er	GER WIN, ERRHANDLER, thes a new error handler ror handler, or an err	r to a window. The error handler must be either a pre- or handler created by a call to	43 44 45
defined er	GER WIN, ERRHANDLER,	r to a window. The error handler must be either a pre- or handler created by a call to	43 44 45 46
defined er	GER WIN, ERRHANDLER, thes a new error handler ror handler, or an err	r to a window. The error handler must be either a pre- or handler created by a call to	43 44 45

1 MPI_WIN_GET_ERRHANDLER(win, errhandler) 2 IN window (handle) win 3 OUT errhandler error handler currently associated with window (han-4 dle) 56 C binding $\overline{7}$ 8 int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler) 9 F08 binding 10 MPI_Win_get_errhandler(win, errhandler, ierror) 11 TYPE(MPI_Win), INTENT(IN) :: win 12TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler 13 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1415F binding 16MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR) 17 INTEGER WIN, ERRHANDLER, IERROR 18 Retrieves the error handler currently associated with a window. 19 208.3.3 Error Handlers for Files 21JMS See comment in LAT_FX 2223 24 25MPI_FILE_CREATE_ERRHANDLER(file_errhandler_fn, errhandler) 26file_errhandler_fn IN user defined error handling procedure (function) 27OUT errhandler MPI error handler (handle) 282930 C binding 31int 32 MPI_File_create_errhandler(MPI_File_errhandler_function *file_errhandler_fn, 33 MPI_Errhandler *errhandler) 34F08 binding 35 MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror) 36 PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn 37 TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler 38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 39 40F binding 41 MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR) 42EXTERNAL FILE_ERRHANDLER_FN 43 INTEGER ERRHANDLER, IERROR 44 Creates an error handler that can be attached to a file object. The user routine should 45 be, in C, a function of type MPI_File_errhandler_function, which is defined as 46 typedef void MPI_File_errhandler_function(MPI_File *, int *, ...); 4748

With the F ABSTRACT SUBROUT TYF	Fortran mpi_f08 module, the u INTERFACE	e, the second is the error code to be returned. user routine file_errhandler_fn should be of the form: function(file, error_code)	1 2 3 4 5 6 7
be of the for subrouting		h, the user routine FILE_ERRHANDLER_FN should ON(FILE, ERROR_CODE)	8 9 10 11 12
MPI_FILE_	_SET_ERRHANDLER(file, errh	andler)	13 14
INOUT	file	file (handle)	15
IN	errhandler	new error handler for file (handle)	16
			17 18
C binding	רי ס		19
int MPI_F	'ile_set_errhandler(MPI_F	ile file, MPI_Errhandler errhandler)	20
F08 bind	ing		21
	<pre>set_errhandler(file, err)</pre>		22 23
	<pre>MPI_File), INTENT(IN) :: MPI_Errhandler), INTENT(</pre>		24
	ER, OPTIONAL, INTENT(OUT		25
		,	26
F binding	g SET_ERRHANDLER(FILE, ERR		27
	ER FILE, ERRHANDLER, IER		28 29
			30
		file. The error handler must be either a predefined ed by a call to MPI_FILE_CREATE_ERRHANDLER.	31
ciror nana	ier, of an error handler create		32
			33
MPI_FILE_	_GET_ERRHANDLER(file, errh	handler)	34
IN	file	file (handle)	35 36
OUT	errhandler	error handler currently associated with file (handle)	37
			38
C binding			39
int MPI_F	'ile_get_errhandler(MPI_F	ile file, MPI_Errhandler *errhandler)	40
F08 bind	ing		41
	get_errhandler(file, err		42 43
	MPI_File), INTENT(IN) ::		44
	MPI_Errhandler), INTENT(ER, OPTIONAL, INTENT(OUT		45
		/ 161101	46
F binding	-		47
MFT_LTFF	GET_ERRHANDLER(FILE, ERR	TANDLER, IERRUR)	48

1 INTEGER FILE, ERRHANDLER, IERROR $\mathbf{2}$ Retrieves the error handler currently associated with a file. 3 4 8.3.4 Freeing Errorhandlers and Retrieving Error Strings 56 7 MPI_ERRHANDLER_FREE(errhandler) 8 9 INOUT errhandler MPI error handler (handle) 10 11 C binding 12int MPI_Errhandler_free(MPI_Errhandler *errhandler) 13 14F08 binding 15MPI_Errhandler_free(errhandler, ierror) 16TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler 17INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 F binding 19 MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR) 20INTEGER ERRHANDLER, IERROR 2122 Marks the error handler associated with errhandler for deallocation and sets errhandler 23to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all the objects 24 associated with it (communicator, window, or file) have been deallocated. 2526MPI_ERROR_STRING(errorcode, string, resultlen) 2728IN errorcode Error code returned by an MPI routine (integer) 29OUT string Text that corresponds to the errorcode (string) 30 OUT resultlen Length (in printable characters) of the result returned 31 in string (integer) 32 33 34C binding int MPI_Error_string(int errorcode, char *string, int *resultlen) 35 36 F08 binding 37 MPI_Error_string(errorcode, string, resultlen, ierror) 38 INTEGER, INTENT(IN) :: errorcode 39 CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string 40 INTEGER, INTENT(OUT) :: resultlen 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4243 F binding 44MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR) 45INTEGER ERRORCODE, RESULTLEN, IERROR 46CHARACTER*(*) STRING 4748

Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER*(*) can not be referenced in, for example, a PRINT statement. (*End of rationale.*)

8.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

To make it possible for an application to interpret an error code, the routine MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called *error classes*. Valid error classes are shown in Table 8.1 and Table 8.2.

The error classes are a subset of the error codes: an MPI function may return an error class number; and the function MPI_ERROR_STRING can be used to compute the error string associated with an error class. The values defined for MPI error classes are valid MPI error codes.

The error codes satisfy,

$$0 = MPI_SUCCESS < MPI_ERR_... \le MPI_ERR_LASTCODE.$$

Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.

Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known LASTCODE is often a nice sanity check as well. (*End of rationale.*)

	00	
MPI_ERROR_CLASS(errorcode, errorclass)		
IN error code returned by an MPI routine (integer)	37	
	38	
OUTerrorclassError class associated with errorcode (integer)	39	
	40	
C binding		
<pre>int MPI_Error_class(int errorcode, int *errorclass)</pre>	42	
F08 binding	43	
MPI_Error_class(errorcode, errorclass, ierror)	44	
INTEGER. INTENT(IN) :: errorcode	45	
	46	
INTEGER, INTENT(OUT) :: errorclass	47	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48	

 $\overline{7}$

 24

1		
2		No error
3	MPI_SUCCESS MPI_ERR_BUFFER	Invalid buffer pointer
4	MPI_ERR_COUNT	Invalid count argument
5	MPI_ERR_TYPE	Invalid datatype argument
6		
7	MPI_ERR_TAG	Invalid tag argument Invalid communicator
8	MPI_ERR_COMM	Invalid communicator
9	MPI_ERR_RANK	
10	MPI_ERR_REQUEST	Invalid request (handle) Invalid root
11	MPI_ERR_ROOT	
12	MPI_ERR_GROUP	Invalid group
13	MPI_ERR_OP	Invalid operation
14	MPI_ERR_TOPOLOGY	Invalid topology
15	MPI_ERR_DIMS	Invalid dimension argument
16	MPI_ERR_ARG	Invalid argument of some other kind
17	MPI_ERR_UNKNOWN	Unknown error
18	MPI_ERR_TRUNCATE	Message truncated on receive
19	MPI_ERR_OTHER	Known error not in this list
20	MPI_ERR_INTERN	Internal MPI (implementation) error
21	MPI_ERR_IN_STATUS	Error code is in status
22	MPI_ERR_PENDING	Pending request
23	MPI_ERR_KEYVAL	Invalid keyval has been passed
24	MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory
25		is exhausted
26	MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM
27	MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY
28	MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL
29	MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE
30	MPI_ERR_SPAWN	Error in spawning processes
31	MPI_ERR_PORT	Invalid port name passed to
32		MPI_COMM_CONNECT
33	MPI_ERR_SERVICE	Invalid service name passed to
34		MPI_UNPUBLISH_NAME
35	MPI_ERR_NAME	Invalid service name passed to
36		MPI_LOOKUP_NAME
37	MPI_ERR_WIN	Invalid win argument
38	MPI_ERR_SIZE	Invalid size argument
39	MPI_ERR_DISP	Invalid disp argument
40	MPI_ERR_INFO	Invalid info argument
41	MPI_ERR_LOCKTYPE	Invalid locktype argument
42	MPI_ERR_ASSERT	Invalid assert argument
43	MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
44	MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls
45		
46	Table 8	1: Error classes (Part 1)
47	14510 0.	
48		

		4
MPI_ERR_RMA_RANGE	Target memory is not part of the win-	5
	dow (in the case of a window created	6
	with MPI_WIN_CREATE_DYNAMIC, tar-	7
	get memory is not attached)	8
MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because	9
	of resource exhaustion)	10
MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some pro-	11
	cess in the group of the specified commu-	12
	nicator cannot expose shared memory)	13
MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the	14
	called function	15
MPI_ERR_FILE	Invalid file handle	16
MPI_ERR_NOT_SAME	Collective argument not identical on all	17
	processes, or collective routines called in	18
	a different order by different processes	19
MPI_ERR_AMODE	Error related to the amode passed to	20
	MPI_FILE_OPEN	21
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to	22
	MPI_FILE_SET_VIEW	23
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	24
	a file which supports sequential access only	25
MPI_ERR_NO_SUCH_FILE	File does not exist	26
MPI_ERR_FILE_EXISTS	File exists	27
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	28
MPI_ERR_ACCESS	Permission denied	29
MPI_ERR_NO_SPACE	Not enough space	30
MPI_ERR_QUOTA	Quota exceeded	31
MPI_ERR_READ_ONLY	Read-only file or file system	32
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	33
	the file is currently open by some process	34
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	35
	tered because a data representation identi-	36
	fier that was already defined was passed to	37
	MPI_REGISTER_DATAREP	38
MPI_ERR_CONVERSION	An error occurred in a user supplied data	39
	conversion function.	40
MPI_ERR_IO	Other I/O error	
 MPI_ERR_LASTCODE	Last error code	41
		42
		43
Table 8.2: Err	cor classes (Part 2)	44
		45
		46

1	F binding
2	MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
3	INTEGER ERRORCODE, ERRORCLASS, IERROR
4	INTEGEN ENRORODE, ERROROLASS, TEMION
5	The function MPI_ERROR_CLASS maps each standard error code (error class) onto
6	itself.
7	
8 9	8.5 Error Classes, Error Codes, and Error Handlers
10 11 12	Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on MPI, see Chapter 13. For this purpose, functions are needed to:
13 14	1. add a new error class to the ones an MPI implementation already knows.
15 16	2. associate error codes with this error class, so that MPI_ERROR_CLASS works.
17	3. associate strings with these error codes, so that MPI_ERROR_STRING works.
18 19	4. invoke the error handler associated with a communicator, window, or object.
20 21 22 23 24	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.
25	MPI_ADD_ERROR_CLASS(errorclass)
26 27	OUTerrorclassvalue for the new error class (integer)
28	C binding
29	int MPI_Add_error_class(int *errorclass)
30	Int Mr1_Add_error_class(int *errorclass)
31	F08 binding
32	MPI_Add_error_class(errorclass, ierror)
33	INTEGER, INTENT(OUT) :: errorclass
34	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35	F binding
36	MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
37	INTEGER ERRORCLASS, IERROR
38 39	
39 40	Creates a new error class and returns the value for it.
41	Rationale. To avoid conflicts with existing error codes and classes, the value is set
42	by the implementation and not by the user. (<i>End of rationale.</i>)
43	
44	Advice to implementors. A high-quality implementation will return the value for
45	a new errorclass in the same deterministic way on all processes. (End of advice to
46	implementors.)
47	
48	

Advice to users. Since a call to MPI_ADD_ERROR_CLASS is local, the same errorclass may not be returned on all processes that make this call. Thus, it is not safe to assume that registering a new error on a set of processes at the same time will yield the same errorclass on all of the processes. However, if an implementation returns the new errorclass in a deterministic way, and they are always generated in the same order on the same set of processes (for example, all processes), then the value will be the same. However, even if a deterministic algorithm is used, the value can vary across processes. This can happen, for example, if different but overlapping groups of processes make a series of calls. As a result of these issues, getting the "same" error on multiple processes may not cause the same value of error code to be generated. (*End of advice to users.*)

The value of MPI_ERR_LASTCODE is a constant value and is not affected by new userdefined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different processes. The value returned by this key is always greater than or equal to MPI_ERR_LASTCODE.

Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class/code. In a multithreaded environment, the user must take extra care in assuming this value has not changed. Note that error codes and error classes are not necessarily dense. A user may not assume that each error class below MPI_LASTUSEDCODE is valid. (*End of advice to users.*)

MPI_ADD_ERROR_CODE(errorclass, errorcode)

Whit_ADD_ERROR_CODE(enorclass, enorcode)			
IN	errorclass	error class (integer)	
OUT	errorcode	new error code to be associated with errorclass (integer)	
C binding	dd_error_code(int errorcl	ass, int *errorcode)	
INTEG INTEG	ng rror_code(errorclass, err ER, INTENT(IN) :: errorc ER, INTENT(OUT) :: error ER, OPTIONAL, INTENT(OUT)	lass code	
	RROR_CODE(ERRORCLASS, ERR ER ERRORCLASS, ERRORCODE,	•	

Creates new error code associated with errorclass and returns its value in errorcode.

Rationale. To avoid conflicts with existing error codes and classes, the value of the 47 new error code is set by the implementation and not by the user. (End of rationale.) 48

Unofficial Draft for Comment Only

 $\overline{7}$

 24

```
1
           Advice to implementors.
                                      A high-quality implementation will return the value for
\mathbf{2}
           a new errorcode in the same deterministic way on all processes. (End of advice to
3
           implementors.)
4
5
6
     MPI_ADD_ERROR_STRING(errorcode, string)
7
8
       IN
                 errorcode
                                              error code or class (integer)
9
       IN
                 string
                                              text corresponding to errorcode (string)
10
11
     C binding
12
     int MPI_Add_error_string(int errorcode, const char *string)
13
14
     F08 binding
15
     MPI_Add_error_string(errorcode, string, ierror)
16
          INTEGER, INTENT(IN) :: errorcode
17
          CHARACTER(LEN=*), INTENT(IN) :: string
18
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
19
     F binding
20
     MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
21
          INTEGER ERRORCODE, IERROR
22
          CHARACTER*(*) STRING
23
^{24}
          Associates an error string with an error code or class. The string must be no more
25
     than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the
26
     calling language. The length of the string does not include the null terminator in C. Trailing
27
     blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING for an errorcode that
28
     already has a string will replace the old string with the new string. It is erroneous to call
29
     MPI_ADD_ERROR_STRING for an error code or class with a value \leq MPI_ERR_LASTCODE.
30
          If MPI_ERROR_STRING is called when no string has been set, it will return a empty
^{31}
     string (all spaces in Fortran, "" in C).
32
          Section 8.3 describes the methods for creating and associating error handlers with
33
     communicators, files, and windows.
34
35
     MPI_COMM_CALL_ERRHANDLER(comm, errorcode)
36
37
       IN
                                              communicator with error handler (handle)
                 comm
38
       IN
                 errorcode
                                              error code (integer)
39
40
     C binding
41
     int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
42
43
     F08 binding
44
     MPI_Comm_call_errhandler(comm, errorcode, ierror)
45
          TYPE(MPI_Comm), INTENT(IN) :: comm
46
          INTEGER, INTENT(IN) :: errorcode
47
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

R	2 3	
$dler$ assigned to the communicator with the error $L_SUCCESS$ in C and the same value in IERROR if	5 6 7	
s preode)		
, 1	1	
n win, int errorcode)	7	
1	8	
	9	
-	0	
- iorror		
	2	
CUDE, IERRUR)		
2		
	27	
CCESS in C and the same value in $IERROR$ if the $$_2$$	8	
suming the process is not aborted and the error $_2$	9	
3	0	
aisatana tha dafault annan handlan fan windawa ia	1	
· · · · · · · · · · · · · · · · · · ·	2	
,		
code)		
file with error handler (handle) 3		
error code (integer) 3	9	
4	0	
4	1	
ile fh, int errorcode) 4	2	
4		
code, ierror)		
fh		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48		
	<pre>ORCODE, IERROR) R fler assigned to the communicator with the error L_SUCCESS in C and the same value in IERROR if (assuming the process is not aborted and the error orcode) window with error handler (handle) error code (integer) n win, int errorcode) code, ierror) win ode :: ierror CODE, IERROR) dler assigned to the window with the error code CCESS in C and the same value in IERROR if the suming the process is not aborted and the error nicators, the default error handler for windows is advice to users.) code file with error handler (handle) error code (integer) ile fh, int errorcode) code, ierror) fh ode ierror</pre>	

FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR) INTEGER FH, ERRORCODE, IERROR
This function invokes the error handler assigned to the file with the error code supplied. function returns MPI_SUCCESS in C and the same value in IERROR if the error handler successfully called (assuming the process is not aborted and the error handler returns).
Advice to users. Unlike errors on communicators and windows, the default behavior for files is to have MPI_ERRORS_RETURN. (<i>End of advice to users.</i>)
Advice to users. Users are warned that handlers should not be called recursively with MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER is called inside an error handler.
Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (<i>End of advice to users.</i>)
Timers and Synchronization
defines a timer. A timer is specified even though it is not "message-passing," because ag parallel programs is important in "performance debugging" and because existing rs (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either incon- ent or do not provide adequate access to high resolution timers. See also Section 2.6.4.
_WTIME()
nding le MPI_Wtime(void)
<pre>binding LE PRECISION MPI_Wtime()</pre>
nding LE PRECISION MPI_WTIME()
MPI_WTIME returns a floating-point number of seconds, representing elapsed wall- time since some time in the past. The "time in the past" is guaranteed not to change during the life of the process.

```
{
    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()

C binding double MPI_Wtick(void)

F08 binding DOUBLE PRECISION MPI_Wtick()

F binding

DOUBLE PRECISION MPI_WTICK()

MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI_WTICK should be 10^{-3} .

8.7 Startup

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine MPI_INIT.

MPI_INIT()

C binding int MPI_Init(int *argc, char ***argv)

F08 binding

```
MPI_Init(ierror)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

 31

MPI_INIT(IERROR)

INTEGER IERROR

All MPI programs must contain exactly one call to an MPI initialization routine: MPI_INIT or MPI_INIT_THREAD. Subsequent calls to any initialization routines are erroneous. The only MPI functions that may be invoked before the MPI initialization routines are called are MPI_GET_VERSION, MPI_GET_LIBRARY_VERSION, MPI_INITIALIZED, MPI_FINALIZED, and any function with the prefix MPI_T_ (within the constraints for functions with this prefix listed in Section ??). The version for ISO C accepts the argc and argv that are provided by the arguments to main or NULL:

```
11
      int main(int argc, char *argv[])
12
      {
13
          MPI_Init(&argc, &argv);
14
15
          /* parse arguments */
16
          /* main program
                                 */
17
18
                                  /* see below */
          MPI_Finalize();
19
          return 0;
20
      }
21
      The Fortran version takes only IERROR.
22
          Conforming implementations of MPI are required to allow applications to pass NULL
23
      for both the argc and argv arguments of main in C.
^{24}
          After MPI is initialized, the application can access information about the execution
25
26
      environment by querying the predefined info object MPI_INFO_ENV. The following keys are
      predefined for this object, corresponding to the arguments of MPI_COMM_SPAWN or of
27
      mpiexec:
28
29
      command Name of program executed.
30
^{31}
      argv Space separated arguments to command.
32
      maxprocs Maximum number of MPI processes to start.
33
34
      soft Allowed values for number of processors.
35
36
      host Hostname.
37
      arch Architecture name.
38
39
      wdir Working directory of the MPI process.
40
^{41}
      file Value is the name of a file in which additional information is specified.
42
      thread_level Requested level of thread support, if requested before the program started exe-
43
           cution.
44
45
          Note that all values are strings. Thus, the maximum number of processes is represented
46
      by a string such as "1024" and the requested level is represented by a string such as
47
      "MPI_THREAD_SINGLE".
48
```

1

 $\mathbf{2}$

3

4

5

6

7

8

9

 $\mathbf{2}$

 $\mathbf{5}$

The info object MPI_INFO_ENV need not contain a (key,value) pair for each of these predefined keys; the set of (key,value) pairs provided is implementation-dependent. Implementations may provide additional, implementation specific, (key,value) pairs.

In case where the MPI processes were started with MPI_COMM_SPAWN_MULTIPLE or, equivalently, with a startup mechanism that supports multiple process specifications, then the values stored in the info object MPI_INFO_ENV at a process are those values that affect the local MPI process.

Example 8.4 If MPI is started with a call to

mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos

Then the first 5 processes will have have in their MPI_INFO_ENV object the pairs (command, ocean), (maxprocs, 5), and (arch, sun). The next 10 processes will have in MPI_INFO_ENV (command, atmos), (maxprocs, 10), and (arch, rs6000)

Advice to users. The values passed in MPI_INFO_ENV are the values of the arguments passed to the mechanism that started the MPI execution — not the actual value provided. Thus, the value associated with maxprocs is the number of MPI processes requested; it can be larger than the actual number of processes obtained, if the soft option was used. (*End of advice to users.*)

Advice to implementors. High-quality implementations will provide a (key,value) pair for each parameter that can be passed to the command that starts an MPI program. (End of advice to implementors.)

MPI_FINALIZE()

C binding

int MPI_Finalize(void)

F08 binding

```
MPI_Finalize(ierror)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

MPI_FINALIZE(IERROR) INTEGER IERROR

This routine cleans up all MPI state. If an MPI program terminates normally (i.e., not due to a call to MPI_ABORT or an unrecoverable error) then each process must call MPI_FINALIZE before it exits.

Before an MPI process invokes MPI_FINALIZE, the process must perform all MPI calls needed to complete its involvement in MPI communications: It must locally complete all MPI operations that it initiated and must execute matching calls needed to complete MPI communications initiated by other processes. For example, if the process executed a nonblocking send, it must eventually call MPI_WAIT, MPI_TEST, MPI_REQUEST_FREE, or any derived function; if the process is the target of a send, then it must post the matching receive; if it is part of a group executing a collective operation, then it must have completed its participation in the operation.

Unofficial Draft for Comment Only

```
1
          The call to MPI_FINALIZE does not free objects created by MPI calls; these objects are
\mathbf{2}
     freed using MPI_XXX_FREE calls.
3
          MPI_FINALIZE is collective over all connected processes. If no processes were spawned,
4
     accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective
\mathbf{5}
     over the union of all processes that have been and continue to be connected, as explained
6
     in Section 10.5.4.
7
          The following examples illustrates these rules
8
     Example 8.5 The following code is correct
9
10
              Process 0
                                           Process 1
11
               _____
                                           _____
12
              MPI_Init();
                                           MPI_Init();
13
              MPI_Send(dest=1);
                                           MPI_Recv(src=0);
14
              MPI_Finalize();
                                           MPI_Finalize();
15
16
17
     Example 8.6 Without a matching receive, the program is erroneous
18
19
              Process 0
                                           Process 1
               _____
                                            _____
20
              MPI_Init();
                                           MPI_Init();
21
              MPI_Send (dest=1);
22
23
              MPI_Finalize();
                                           MPI_Finalize();
^{24}
25
     Example 8.7 This program is correct: Process 0 calls MPI_Finalize after it has executed
26
     the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call
27
     that completes the matching receive operation before it calls MPI_Finalize.
28
29
       Process 0
                                          Proces 1
30
        _____
                                          _____
31
       MPI_Init();
                                          MPI_Init();
32
       MPI_Isend(dest=1);
                                          MPI_Recv(src=0);
33
       MPI_Request_free();
                                          MPI_Finalize();
34
       MPI_Finalize();
                                          exit();
35
     exit();
36
37
     Example 8.8 This program is correct. The attached buffer is a resource allocated by the
38
     user, not by MPI; it is available to the user after MPI is finalized.
39
40
         Process 0
                                           Process 1
41
         _____
                                            _____
42
         MPI_Init();
                                          MPI_Init();
43
         buffer = malloc(1000000);
                                          MPI_Recv(src=0);
44
         MPI_Buffer_attach();
                                          MPI_Finalize();
45
         MPI_Send(dest=1));
                                          exit();
46
         MPI_Finalize();
47
         free(buffer);
48
         exit();
```

Example 8.9 This program is correct. The cancel operation must succeed, since the send cannot complete normally. The wait operation, after the call to MPI_Cancel, is local — no matching MPI call is required on process 1. Cancelling a send request by calling MPI_CANCEL is deprecated.

Process 0	Process 1
<pre>MPI_Issend(dest=1);</pre>	<pre>MPI_Finalize();</pre>
<pre>MPI_Cancel();</pre>	
<pre>MPI_Wait();</pre>	
<pre>MPI_Finalize();</pre>	

Advice to implementors. Even though a process has executed all MPI calls needed to complete the communications it is involved with, such communication may not yet be completed from the viewpoint of the underlying MPI system. For example, a blocking send may have returned, even though the data is still buffered at the sender in an MPI buffer; an MPI process may receive a cancel request for a message it has completed receiving. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication 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 ??).

Although it is not required that all processes return from MPI_FINALIZE, it is required that at least process 0 in MPI_COMM_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, users may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Example 8.10 The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
   resultfile = fopen("outfile", "w");
   dump_results(resultfile);
   fclose(resultfile);
}
exit(0);
```

 31

```
1
     MPI_INITIALIZED(flag)
2
       OUT
                 flag
                                              Flag is true if MPI_INIT has been called and flags oth-
3
                                              erwise (logical)
4
5
     C binding
6
     int MPI_Initialized(int *flag)
7
8
     F08 binding
9
     MPI_Initialized(flag, ierror)
10
          LOGICAL, INTENT(OUT) ::
                                       flag
11
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     F binding
13
     MPI_INITIALIZED(FLAG, IERROR)
14
          LOGICAL FLAG
15
          INTEGER IERROR
16
17
          This routine may be used to determine whether MPI_INIT has been called.
18
     MPI_INITIALIZED returns true if the calling process has called MPI_INIT. Whether
19
     MPI_FINALIZE has been called does not affect the behavior of MPI_INITIALIZED. It is one
20
     of the few routines that may be called before MPI_INIT is called. This function must always
21
     be thread-safe, as defined in Section 12.4.
22
23
     MPI_ABORT(comm, errorcode)
^{24}
25
       IN
                 comm
                                              communicator of tasks to abort (handle)
26
       IN
                 errorcode
                                              error code to return to invoking environment (integer)
27
28
     C binding
29
     int MPI_Abort(MPI_Comm comm, int errorcode)
30
^{31}
     F08 binding
32
     MPI_Abort(comm, errorcode, ierror)
33
          TYPE(MPI_Comm), INTENT(IN) :: comm
34
          INTEGER, INTENT(IN) :: errorcode
35
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
36
     F binding
37
     MPI_ABORT(COMM, ERRORCODE, IERROR)
38
          INTEGER COMM, ERRORCODE, IERROR
39
40
          This routine makes a "best attempt" to abort all tasks in the group of comm. This
41
     function does not require that the invoking environment take any action with the error
42
     code. However, a Unix or POSIX environment should handle this as a return errorcode
43
     from the main program.
44
          It may not be possible for an MPI implementation to abort only the processes repre-
45
     sented by comm if this is a subset of the processes. In this case, the MPI implementation
46
     should attempt to abort all the connected processes but should not abort any unconnected
47
```

processes. If no processes were spawned, accepted, or connected then this has the effect of aborting all the processes associated with MPI_COMM_WORLD.

Advice to implementors. After aborting a subset of processes, a high quality implementation should be able to provide error handling for communicators, windows, and files involving both aborted and non-aborted processes. As an example, if the user changes the error handler for MPI_COMM_WORLD to MPI_ERRORS_RETURN or a custom error handler, when a subset of MPI_COMM_WORLD is aborted, the remaining processes in MPI_COMM_WORLD should be able to continue communicating with each other and receive appropriate error codes when attempting communication with an aborted process. (*End of advice to implementors.*)

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

8.7.1 Allowing User Functions at Process Termination

There are times in which it would be convenient to have actions happen when an MPI process finishes. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that being terminated in the case of dynamically created processes) is finished. This can be accomplished in MPI by attaching an attribute to MPI_COMM_SELF with a callback function. When MPI_FINALIZE is called, it will first execute the equivalent of an MPI_COMM_FREE on MPI_COMM_SELF. This will cause the delete callback function to be executed on all keys associated with MPI_COMM_SELF, in the reverse order that they were set on MPI_COMM_SELF. If no key has been attached to MPI_COMM_SELF, then no callback is invoked. The "freeing" of MPI_COMM_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.

Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. Implementations that use the attribute delete callback on MPI_COMM_SELF internally should register their internal callbacks before returning from MPI_INIT / MPI_INIT_THREAD, so that libraries or applications will not have portions of the MPI implementation shut down before the application-level callbacks are made. (*End of advice to implementors.*)

8.7.2 Determining Whether MPI Has Finished

One of the goals of MPI was to allow for layered libraries. In order for a library to do this cleanly, it needs to know if MPI is active. In MPI the function MPI_INITIALIZED was provided to tell if MPI had been initialized. The problem arises in knowing if MPI has been 48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11 12

13

14

15 16 17

18

19

20 21

22 23

 24

25

26

27

28

29

30

 31

32

33

34

35 36

37

38

39

40

41

42

43 44

1 finalized. Once MPI has been finalized it is no longer active and cannot be restarted. A $\mathbf{2}$ library needs to be able to determine this to act accordingly. To achieve this the following 3 function is needed: 4 5MPI_FINALIZED(flag) 6 $\overline{7}$ OUT flag true if MPI was finalized (logical) 8 9 C binding 10 int MPI_Finalized(int *flag) 11 F08 binding 12MPI_Finalized(flag, ierror) 13 LOGICAL, INTENT(OUT) :: flag 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516F binding 17MPI_FINALIZED(FLAG, IERROR) 18 LOGICAL FLAG 19INTEGER IERROR 20This routine returns true if MPI_FINALIZE has completed. It is valid to call 21MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE. This function must always be 22 thread-safe, as defined in Section 12.4. 23 24 Advice to users. MPI is "active" and it is thus safe to call MPI functions if MPI_INIT 25has completed and MPI_FINALIZE has not completed. If a library has no other 26way of knowing whether MPI is active or not, then it can use MPI_INITIALIZED and 27MPI_FINALIZED to determine this. For example, MPI is "active" in callback functions 28that are invoked during MPI_FINALIZE. (End of advice to users.) 29 30 31 Portable MPI Process Startup 8.8 32 A number of implementations of MPI provide a startup command for MPI programs that 33 34is of the form 35 mpirun <mpirun arguments> <program> <program arguments> 36 37 Separating the command to start the program from the program itself provides flexibility, 38 particularly for network and heterogeneous implementations. For example, the startup 39 script need not run on one of the machines that will be executing the MPI program itself. 40 Having a standard startup mechanism also extends the portability of MPI programs one 41 step further, to the command lines and scripts that manage them. For example, a validation 42suite script that runs hundreds of programs can be a portable script if it is written using such 43a standard starup mechanism. In order that the "standard" command not be confused with 44existing practice, which is not standard and not portable among implementations, instead 45of 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

Unofficial Draft for Comment Only

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></th></maxproc<>	s>
-soft	<	>
-host	<	>
-arch	<	>
-wdir	<	>
-path	<	>
-file	<	>
<commai< td=""><td>nd line></td><td></td></commai<>	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><</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 $\langle \rangle$.
4	
5	Example 8.11 Start 16 instances of myprog on the current or default machine:
6	Example 6.11 Start to instances of myprog on the current of default machine.
7	mpiexec -n 16 myprog
8	mbrexec -u 10 mybrog
9	
10	Example 8.12 Start 10 processes on the machine called ferrari:
11	Example 6.12 Start to processes on the machine caned refrart.
12	mpievec - n 10 - heat ferreri munrer
	mpiexec -n 10 -host ferrari myprog
13	
14	Example 8.13 Start three copies of the same program with different command-line
15	arguments:
16	alguments.
17	muisuss muusus infils1 , muusus infils0 , muusus 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	
22	RS/6000's:
23	mainers of Europhism second and 10 such as COOO stars
24	mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
25	
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	Example 8.15 Start the ocean program on five Suns and the atmos program on 10
30	RS/6000's (Form B):
31	
32	<pre>mpiexec -configfile myfile</pre>
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	
44	
45	
46	
47	
48	

Chapter 9

The Info Object

Many of the routines in MPI take an argument info. info is an opaque object with a handle of type MPI_Info in C and Fortran with the mpi_f08 module, and INTEGER in Fortran with the mpi module or the include file mpif.h. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI. Some info hints allow the MPI library to restrict its support for certain operations in order to improve performance or resource utilization. If an application provides such an info hint, it must be compatible with any changes in the behavior of the MPI library that are allowed by the info hint.

An implementation must support info objects as caches for arbitrary (key,value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET must retain all (key,value) pairs so that layered functionality can also use the Info object.

Keys have an implementation-defined maximum length of MPI_MAX_INFO_KEY, which is at least 32 and at most 255. Values have an implementation-defined maximum length of MPI_MAX_INFO_VAL. In Fortran, leading and trailing spaces are stripped from both. Returned values will never be larger than these maximum lengths. Both key and value are case sensitive.

Rationale. Keys have a maximum length because the set of known keys will always be finite and known to the implementation and because there is no reason for keys to be complex. The small maximum size allows applications to declare keys of size MPI_MAX_INFO_KEY. The limitation on value sizes is so that an implementation is not forced to deal with arbitrarily long strings. (*End of rationale.*)

Advice to users. MPI_MAX_INFO_VAL might be very large, so it might not be wise to declare a string of that size. (End of advice to users.)

When info is used as an argument to a nonblocking routine, it is parsed before that routine returns, so that it may be modified or freed immediately after return.

When the descriptions refer to a key or value as being a boolean, an integer, or a list, they mean the string representation of these types. An implementation may define its own

7 8 9

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38

39

40

41 42

43

44

 $45 \\ 46$

47

1 rules for how info value strings are converted to other types, but to ensure portability, every $\mathbf{2}$ implementation must support the following representations. Valid values for a boolean must 3 include the strings "true" and "false" (all lowercase). For integers, valid values must include 4 string representations of decimal values of integers that are within the range of a standard $\mathbf{5}$ integer type in the program. (However it is possible that not every integer is a valid value 6 for a given key.) On positive numbers, + signs are optional. No space may appear between $\overline{7}$ a + or - sign and the leading digit of a number. For comma separated lists, the string 8 must contain valid elements separated by commas. Leading and trailing spaces are stripped 9 automatically from the types of info values described above and for each element of a comma 10 separated list. These rules apply to all info values of these types. Implementations are free 11to specify a different interpretation for values of other info keys. 1213 MPI_INFO_CREATE(info) 1415OUT info info object created (handle) 1617C binding 18 int MPI_Info_create(MPI_Info *info) 19 F08 binding 20MPI_Info_create(info, ierror) 21TYPE(MPI_Info), INTENT(OUT) :: info 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23 24 F binding 25MPI_INFO_CREATE(INFO, IERROR) 26INTEGER INFO, IERROR 27MPI_INFO_CREATE creates a new info object. The newly created object contains no 28key/value pairs. 2930 31 MPI_INFO_SET(info, key, value) 32 INOUT info info object (handle) 33 34IN key key (string) 35 IN value value (string) 36 37 C binding 38 int MPI_Info_set(MPI_Info info, const char *key, const char *value) 39 40F08 binding 41 MPI_Info_set(info, key, value, ierror) 42TYPE(MPI_Info), INTENT(IN) :: info 43 CHARACTER(LEN=*), INTENT(IN) :: key, value 44INTEGER, OPTIONAL, INTENT(OUT) :: ierror 45F binding 46MPI_INFO_SET(INFO, KEY, VALUE, IERROR) 47INTEGER INFO, IERROR 48

1 CHARACTER*(*) KEY, VALUE $\mathbf{2}$ MPI_INFO_SET adds the (key, value) pair to info, and overrides the value if a value for 3 the same key was previously set. key and value are null-terminated strings in C. In Fortran, 4 leading and trailing spaces in key and value are stripped. If either key or value are larger 5than the allowed maximums, the errors MPI_ERR_INFO_KEY or MPI_ERR_INFO_VALUE are 6 raised, respectively. 7 8 9 MPI_INFO_DELETE(info, key) 10 INOUT info info object (handle) 11 IN key (string) key 1213 14C binding 15int MPI_Info_delete(MPI_Info info, const char *key) 16F08 binding 17MPI_Info_delete(info, key, ierror) 18 TYPE(MPI_Info), INTENT(IN) :: info 19 CHARACTER(LEN=*), INTENT(IN) :: key 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122 F binding 23MPI_INFO_DELETE(INFO, KEY, IERROR) 24INTEGER INFO, IERROR 25CHARACTER*(*) KEY 26MPI_INFO_DELETE deletes a (key,value) pair from info. If key is not defined in info, 27the call raises an error of class MPI_ERR_INFO_NOKEY. 28 29 30 MPI_INFO_GET(info, key, valuelen, value, flag) 31IN info info object (handle) 32 33 IN key key (string) 34 valuelen IN length of value arg (integer) 35 OUT value value (string) 36 37 OUT flag true if key defined, false if not (logical) 38 39 C binding 40 int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value, 41 int *flag) 42F08 binding 43 MPI_Info_get(info, key, valuelen, value, flag, ierror) 44TYPE(MPI_Info), INTENT(IN) :: info 45CHARACTER(LEN=*), INTENT(IN) :: key 46 INTEGER, INTENT(IN) :: valuelen 47CHARACTER(LEN=*), INTENT(OUT) :: value 48

Unofficial Draft for Comment Only

```
1
          LOGICAL, INTENT(OUT) :: flag
\mathbf{2}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     F binding
4
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
5
          INTEGER INFO, VALUELEN, IERROR
6
          CHARACTER*(*) KEY, VALUE
7
          LOGICAL FLAG
8
9
          This function retrieves the value associated with key in a previous call to
10
     MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value,
11
     otherwise it sets flag to false and leaves value unchanged. valuelen is the number of characters
12
     available in value. If it is less than the actual size of the value, the value is truncated. In
13
     C, valuelen should be one less than the amount of allocated space to allow for the null
14
     terminator.
15
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
16
17
     MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
18
19
       IN
                                               info object (handle)
                 info
20
       IN
                  key
                                              key (string)
21
       OUT
                 valuelen
                                              length of value arg (integer)
22
23
       OUT
                 flag
                                               true if key defined, false if not (logical)
^{24}
25
     C binding
26
     int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
27
                     int *flag)
28
     F08 binding
29
     MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
30
          TYPE(MPI_Info), INTENT(IN) :: info
31
          CHARACTER(LEN=*), INTENT(IN) :: key
32
          INTEGER, INTENT(OUT) :: valuelen
33
34
          LOGICAL, INTENT(OUT) :: flag
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     F binding
37
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
38
          INTEGER INFO, VALUELEN, IERROR
39
          CHARACTER*(*) KEY
40
          LOGICAL FLAG
41
42
          Retrieves the length of the value associated with key. If key is defined, valuelen is set to
     the length of its associated value and flag is set to true. If key is not defined, valuelen is not
43
     touched and flag is set to false. The length returned in C does not include the end-of-string
44
45
     character.
          If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
46
47
48
```

MPI INF	O_GET_NKEYS(info, nkeys)		1
IN	info	info object (handle)	2
		,	3
OUT	nkeys	number of defined keys (integer)	4
			5
C bindi	ng _Info_get_nkeys(MPI_Info i	nfo int ankows)	6 7
		iio, iiit *iikeys)	8
F08 bin	0	、 、	9
	<pre>_get_nkeys(info, nkeys, i E(MDL Info) INTENT(IN)</pre>		10
	E(MPI_Info), INTENT(IN) :: EGER, INTENT(OUT) :: nkey		11
	EGER, OPTIONAL, INTENT(OUT		12
		,	13
F bindi	-		14 15
	D_GET_NKEYS(INFO, NKEYS, I EGER INFO, NKEYS, IERROR	ERRUR)	16
			17
MPI	_INFO_GET_NKEYS returns the set of the set o	ne number of currently defined keys in info.	18
			19
MPI_INF	O_GET_NTHKEY(info, n, key)		20
IN	info	info object (handle)	21
IN		key number (integer)	22 23
	n		23
OUT	key	key (string)	25
			26
C bindi	ng _Info_get_nthkey(MPI_Info	info int n char *key)	27
		into, int n, chai *key)	28
F08 bin	0		29
	<pre>b_get_nthkey(info, n, key, p(wpt_t_c) = INTERNT(IN)</pre>		30
	E(MPI_Info), INTENT(IN) :: EGER, INTENT(IN) :: n	info	31 32
	RACTER(LEN=*), INTENT(OUT)	:: kev	33
	EGER, OPTIONAL, INTENT(OUT		34
F bindi	2.7		35
	ng D_GET_NTHKEY(INFO, N, KEY,	TERROR)	36
	EGER INFO, N, IERROR		37
	RACTER*(*) KEY		38
Thi	function noturns the nth define	d have in info Kava are numbered 0. N 1 where	39 40
		ed key in info. Keys are numbered $0 \dots N - 1$ where _GET_NKEYS. All keys between 0 and $N - 1$ are	41
	-	of a given key does not change as long as info is not	42
0	with MPI_INFO_SET or $MPI_$		43
			4.4

```
1
     MPI_INFO_DUP(info, newinfo)
\mathbf{2}
       IN
                 info
                                              info object (handle)
3
       OUT
                 newinfo
                                              info object (handle)
4
5
6
     C binding
\overline{7}
     int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
8
     F08 binding
9
     MPI_Info_dup(info, newinfo, ierror)
10
          TYPE(MPI_Info), INTENT(IN) :: info
11
          TYPE(MPI_Info), INTENT(OUT) :: newinfo
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     F binding
15
     MPI_INFO_DUP(INFO, NEWINFO, IERROR)
16
          INTEGER INFO, NEWINFO, IERROR
17
          MPI_INFO_DUP duplicates an existing info object, creating a new object, with the
18
     same (key,value) pairs and the same ordering of keys.
19
20
21
     MPI_INFO_FREE(info)
22
       INOUT
                 info
                                              info object (handle)
23
24
     C binding
25
     int MPI_Info_free(MPI_Info *info)
26
27
     F08 binding
28
     MPI_Info_free(info, ierror)
29
          TYPE(MPI_Info), INTENT(INOUT) :: info
30
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
31
     F binding
32
     MPI_INFO_FREE(INFO, IERROR)
33
          INTEGER INFO, IERROR
34
35
          This function frees info and sets it to MPI_INFO_NULL. The value of an info argument is
36
     interpreted each time the info is passed to a routine. Changes to an info after return from
37
     a routine do not affect that interpretation.
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

10.2.1 Starting Processes

 $^{25}_{26}$ MPI applications may start new processes through an interface to an external process manager.

²⁷ MPI_COMM_SPAWN starts MPI processes and establishes communication with them,
 ²⁸ returning an intercommunicator. MPI_COMM_SPAWN_MULTIPLE starts several different
 ³⁰ binaries (or the same binary with different arguments), placing them in the same
 ³¹ MPI_COMM_WORLD and returning an intercommunicator.

MPI uses the group abstraction to represent processes. A process is identified by a (group, rank) pair.

³⁴₃₅ 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:

 $40 \\ 41$

42

43

44

45

46

32

33

• MPP managed by a batch queueing system. Batch queueing systems generally allocate resources before an application begins, enforce limits on resource use (CPU time, memory use, etc.), and do not allow a change in resource allocation after a job begins. Moreover, many MPPs have special limitations or extensions, such as a limit on the number of processes that may run on one processor, or the ability to gang-schedule processes of a parallel application.

47 48

1

2

3

4 5

> 6 7

8

9

10

11

12

13

14 15

16 17

18

19

20

- Network of workstations with PVM. PVM (Parallel Virtual Machine) allows a user to create a "virtual machine" out of a network of workstations. An application may extend the virtual machine or manage processes (create, kill, redirect output, etc.) through the PVM library. Requests to manage the machine or processes may be intercepted and handled by an external resource manager.
- Network of workstations managed by a load balancing system. A load balancing system may choose the location of spawned processes based on dynamic quantities, such as load average. It may transparently migrate processes from one machine to another when a resource becomes unavailable.
- Large SMP with Unix. Applications are run directly by the user. They are scheduled at a low level by the operating system. Processes may have special scheduling characteristics (gang-scheduling, processor affinity, deadline scheduling, processor locking, etc.) and be subject to OS resource limits (number of processes, amount of memory, etc.).

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide "operating system" services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.).

Complex interaction of an MPI application with its runtime environment should be done through an environment-specific API. An example of such an API would be the PVM task and machine management routines — pvm_addhosts, pvm_config, pvm_tasks, etc., possibly modified to return an MPI (group, rank) when possible. A Condor or PBS API would be another possibility.

At some low level, obviously, MPI must be able to interact with the runtime system, but the interaction is not visible at the application level and the details of the interaction are not specified by the MPI standard.

In many cases, it is impossible to keep environment-specific information out of the MPI interface without seriously compromising MPI functionality. To permit applications to take advantage of environment-specific functionality, many MPI routines take an info argument that allows an application to specify environment-specific information. There is a tradeoff between functionality and portability: applications that make use of environment-specific info are not portable.

MPI does not require the existence of an underlying "virtual machine" model, in which there is a consistent global view of an MPI application and an implicit "operating system" managing resources and processes. For instance, processes spawned by one task may not be visible to another; additional hosts added to the runtime environment by one process may not be visible in another process; tasks spawned by different processes may not be automatically distributed over available resources.

Interaction between MPI and the runtime environment is limited to the following areas:

- A process may start new processes with MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE.
- When a process spawns a child process, it may optionally use an info argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.

Unofficial Draft for Comment Only

1 2

3

4

5

6

 $\overline{7}$

8

9

10 11

12

13

14

15

16 17

18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38

39

40

41

42 43

44

45

46

47

• An attribute MPI_UNIVERSE_SIZE (See Section 10.5.1) on MPI_COMM_WORLD tells a program how "large" the initial runtime environment is, namely how many processes can usefully be started in all. One can subtract the size of MPI_COMM_WORLD from this value to find out how many processes might usefully be started in addition to those already running.

10.3 Process Manager Interface

10.3.1 Processes in MPI

A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a
 unique process but a process does not determine a unique (group, rank) pair, since a process
 may belong to several groups.

 $14 \\ 15$

19

20

21

22

23

1

 $\mathbf{2}$

3

4

5

6 7

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

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes)

28			
29 30	IN	command	name of program to be spawned (string, significant only at root)
31 32	IN	argv	arguments to command (array of strings, significant only at root)
33 34 35	IN	maxprocs	maximum number of processes to start (integer, sig- nificant only at root)
36 37 38	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)
39 40	IN	root	rank of process in which previous arguments are ex- amined (non-negative integer)
41 42	IN	comm	intracommunicator containing group of spawning processes (handle)
43 44 45	OUT	intercomm	intercommunicator between original group and the newly spawned group (handle)
46 47	OUT	array_of_errcodes	one code per process (array of integer)

⁴⁸ C binding

F08 binding

MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
array_of_errcodes, ierror)
CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
INTEGER, INTENT(IN) :: maxprocs, root
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Comm), INTENT(OUT) :: intercomm
<pre>INTEGER :: array_of_errcodes(*)</pre>
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
F binding

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 sys-

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

CHAPTER 10. PROCESS CREATION AND MANAGEMENT

1	tem with a global file system might look first in the working directory of the spawning		
2	process, or might search the directories in a PATH environment variable as do Unix		
3	shells. An implementation on top of PVM would use PVM's rules for finding exe-		
4	cutables (usually in \$HOME/pvm3/bin/\$PVM_ARCH). An MPI implementation running		
5	under POE on an IBM SP would use POE's method of finding executables. An imple-		
6	mentation should document its rules for finding executables and determining working		
7	directories, and a high-quality implementation should give the user some control over		
8	these rules. (End of advice to implementors.)		
9			
10	If the program named in command does not call MPI_INIT, but instead forks a process		
11	that calls MPI_INIT, the results are undefined. Implementations may allow this case to		
12	work but are not required to.		
13			
14	Advice to users. MPI does not say what happens if the program you start is a		
15	shell script and that shell script starts a program that calls MPI_INIT. Though some		
16	implementations may allow you to do this, they may also have restrictions, such as		
17	requiring that arguments supplied to the shell script be supplied to the program, or		
18	requiring that certain parts of the environment not be changed. (End of advice to		
19	users.)		
20			
21	The argv argument argv is an array of strings containing arguments that are passed to		
22	the program. The first element of argv is the first argument passed to command, not, as		
23	is conventional in some contexts, the command itself. The argument list is terminated by		
24	NULL in C and an empty string in Fortran. In Fortran, leading and trailing spaces are		
25	always stripped, so that a string consisting of all spaces is considered an empty string. The		
26	constant MPI_ARGV_NULL may be used in C and Fortran to indicate an empty argument		
27	list. In C this constant is the same as NULL.		
28			
29	Example 10.1 Examples of argv in C and Fortran		
30	To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:		
31			
32	char command[] = "ocean";		
33	<pre>char *argv[] = {"-gridfile", "ocean1.grd", NULL};</pre>		
34	<pre>MPI_Comm_spawn(command, argv,);</pre>		
35	or, if not everything is known at compile time:		
36			
37	char *command;		
38	char **argv;		
39	<pre>command = "ocean";</pre>		
40	<pre>argv=(char **)malloc(3 * sizeof(char *));</pre>		
41	<pre>argv[0] = "-gridfile";</pre>		
42	<pre>argv[1] = "ocean1.grd";</pre>		
43	argv[2] = NULL;		
44	<pre>MPI_Comm_spawn(command, argv,);</pre>		
45	In Dentucation		
46	In Fortran:		
47			
48			

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

1 number of user-specified (key,value) pairs. key and value are strings (null-terminated char* $\mathbf{2}$ in C, character*(*) in Fortran). Routines to create and manipulate the info argument are 3 described in Chapter 9.

4 For the SPAWN calls, info provides additional (and possibly implementation-dependent) $\mathbf{5}$ instructions to MPI and the runtime system on how to start processes. An application may 6 pass MPI_INFO_NULL in C or Fortran. Portable programs not requiring detailed control over 7process locations should use MPI_INFO_NULL.

8 MPI does not specify the content of the info argument, except to reserve a number of 9 special key values (see Section 10.3.4). The info argument is quite flexible and could even 10 be used, for example, to specify the executable and its command-line arguments. In this 11case the command argument to MPI_COMM_SPAWN could be empty. The ability to do this 12follows from the fact that MPI does not specify how an executable is found, and the info 13argument can tell the runtime system where to "find" the executable "" (empty string). Of 14course a program that does this will not be portable across MPI implementations.

15

16The root argument All arguments before the root argument are examined only on the 17process whose rank in comm is equal to root. The value of these arguments on other 18 processes is ignored.

19

20The array_of_errcodes argument The array_of_errcodes is an array of length maxprocs in 21which MPI reports the status of each process that MPI was requested to start. If all maxprocs 22processes were spawned, $\operatorname{array_of}_{\operatorname{errcodes}}$ is filled in with the value MPI_SUCCESS. If only m 23 $(0 \le m \le maxprocs)$ processes are spawned, m of the entries will contain MPI_SUCCESS and 24the rest will contain an implementation-specific error code indicating the reason MPI could 25not start the process. MPI does not specify which entries correspond to failed processes. 26An implementation may, for instance, fill in error codes in one-to-one correspondence with 27a detailed specification in the info argument. These error codes all belong to the error class 28MPI_ERR_SPAWN if there was no error in the argument list. In C or Fortran, an application 29may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes. 30

Advice to implementors. MPI_ERRCODES_IGNORE in Fortran is a special type of constant, like MPI_BOTTOM. See the discussion in Section 2.5.4. (End of advice to *implementors.*)

36 MPI_COMM_GET_PARENT(parent) 37

OUT

38 39

 31

32 33

34 35

the parent communicator (handle)

C binding 40

int MPI_Comm_get_parent(MPI_Comm *parent) 41

42F08 binding 43 MPI_Comm_get_parent(parent, ierror) 44 TYPE(MPI_Comm), INTENT(OUT) :: parent 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4647F binding

MPI_COMM_GET_PARENT(PARENT, IERROR) 48

parent

INTEGER PARENT, IERROR

If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, MPI_COMM_GET_PARENT returns the "parent" intercommunicator of the current process. This parent intercommunicator is created implicitly inside of MPI_INIT and is the same intercommunicator returned by SPAWN in the parents.

If the process was not spawned, MPI_COMM_GET_PARENT returns MPI_COMM_NULL. After the parent communicator is freed or disconnected, MPI_COMM_GET_PARENT returns MPI_COMM_NULL.

Advice to users. MPI_COMM_GET_PARENT returns a handle to a single intercommunicator. Calling MPI_COMM_GET_PARENT a second time returns a handle to the same intercommunicator. Freeing the handle with MPI_COMM_DISCONNECT or MPI_COMM_FREE will cause other references to the intercommunicator to become invalid (dangling). Note that calling MPI_COMM_FREE on the parent communicator is not useful. (*End of advice to users.*)

Rationale. The desire of the Forum was to create a constant MPI_COMM_PARENT similar to MPI_COMM_WORLD. Unfortunately such a constant cannot be used (syntactically) as an argument to MPI_COMM_DISCONNECT, which is explicitly allowed. (*End of rationale.*)

10.3.3 Starting Multiple Executables and Establishing Communication

While MPI_COMM_SPAWN is sufficient for most cases, it does not allow the spawning of multiple binaries, or of the same binary with multiple sets of arguments. The following routine spawns multiple binaries or the same binary with multiple sets of arguments, establishing communication with them and placing them in the same MPI_COMM_WORLD.

ry with multiple sets of a he same binary with mul ad placing them in the same $\mathbf{2}$

```
424
```

1

 $\mathbf{2}$

3

4

5

6 7

8

9

```
MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs,
                    array_of_info, root, comm, intercomm, array_of_errcodes)
       IN
                 count
                                             number of commands (integer, significant only at root)
       IN
                 array_of_commands
                                             programs to be executed (array of strings, significant
                                             only at root)
       IN
                 array_of_argv
                                             arguments for commands (array of array of array of
                                             stringss, significant only at root)
       IN
                 array_of_maxprocs
                                             maximum number of processes to start for each com-
10
                                             mand (non-negative integer, significant only at root)
11
       IN
                 array_of_info
                                             info objects telling the runtime system where and how
12
                                             to start processes (handle, significant only at root)
13
14
       IN
                 root
                                             rank of process in which previous arguments are ex-
15
                                             amined (non-negative integer)
16
       IN
                 comm
                                             intracommunicator containing group of spawning pro-
17
                                             cesses (handle)
18
       OUT
                 intercomm
                                             intercommunicator between original group and the newly
19
                                             spawned group (handle)
20
21
       OUT
                 array_of_errcodes
                                             one code per process (integer)
22
23
     C binding
^{24}
     int MPI_Comm_spawn_multiple(int count, char* array_of_commands[],
25
                    char** array_of_argv[], const int array_of_maxprocs[],
26
                    const MPI_Info array_of_info[], int root, MPI_Comm comm,
27
                    MPI_Comm *intercomm, int array_of_errcodes[])
28
     F08 binding
29
     MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
30
                    array_of_maxprocs, array_of_info, root, comm, intercomm,
^{31}
                    array_of_errcodes, ierror)
32
          INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
33
          CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
34
          array_of_argv(count,*)
35
          TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
36
          TYPE(MPI_Comm), INTENT(IN) :: comm
37
          TYPE(MPI_Comm), INTENT(OUT) ::
                                              intercomm
38
          INTEGER :: array_of_errcodes(*)
39
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
40
     F binding
42
     MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
43
                    ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
44
                    ARRAY_OF_ERRCODES, IERROR)
45
          INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,
46
          INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
47
          CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(*)
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 formal 50 difficult, we include	9
1. a means a	10
2. a:b means $a, a + 1, a + 2,, b$	11
3. a:b:c means $a, a + c, a + 2c, \ldots, a + ck$, where for $c > 0, k$ is the largest integer	12 r
for which $a + ck \le b$ and for $c < 0$, k is the largest integer for which $a + ck \ge b$	10
If $b > a$ then c must be positive. If $b < a$ then c must be negative.	• 14 15
Examples:	16
1. a:b gives a range between a and b	17 18
2. 0:N gives full "soft" functionality	19
	20
3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows a power-of-two number of processes.	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.	23 24
0. 2.10.2,1 allows 2, 1, 0, 1, 0, 01 10 processes.	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 33
<pre>int world_size, universe_size, *universe_sizep, flag; MDL Comm account of the intervention of the intervention of the second secon</pre>	34
MPI_Comm everyone; /* intercommunicator */	35
char worker_program[100];	36
<pre>MPI_Init(&argc, &argv);</pre>	37
MPI_Comm_size(MPI_COMM_WORLD, &world_size);	38
In I_comm_Size(In I_conn_wolled, wwolld_Size);	39
if (world_size != 1) error("Top heavy with management");	40
(***_****** _/ ********************	41
MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,	42
&universe_sizep, &flag);	43
if (!flag) {	44
$printf("This MPI does not support UNIVERSE_SIZE. How many\n$	45
processes total?");	46
<pre>scanf("%d", &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

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

- 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. 1. Applications that do not rely on the ability to publish names are the most portable. Typically the port_name must be transferred "by hand" from server to client. 2. Applications that use the MPI_PUBLISH_NAME mechanism are completely portable among implementations that provide this service. To be portable among all implementations, these applications should have a fall-back mechanism that can be used when names are not published. 3. Applications may ignore MPI's name publishing functionality and use their own mechanism (possibly system-supplied) to publish names. This allows arbitrary flexibility but is not portable. 10.4.2 Server Routines A server makes itself available with two routines. First it must call MPI_OPEN_PORT to establish a port at which it may be contacted. Secondly it must call MPI_COMM_ACCEPT to accept connections from clients.
- 39 40 41

MPI_OPEN_PORT(info, port_name)

42IN info implementation-specific information on how to estab-43 lish an address (handle) 44OUT newly established port (string) port_name 4546 C binding 47int MPI_Open_port(MPI_Info info, char *port_name) 48

F08 binding MPI_Open_port(info, port_name, ierror) TYPE(MPI_Info), INTENT(IN) :: info CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_OPEN_PORT(INFO, PORT_NAME, IERROR) INTEGER INFO, IERROR CHARACTER*(*) PORT_NAME

This function establishes a network address, encoded in the port_name string, at which the server will be able to accept connections from clients. port_name is supplied by the system, possibly using information in the info argument.

MPI copies a system-supplied port name into port_name. port_name identifies the newly opened port and can be used by a client to contact the server. The maximum size string that may be supplied by the system is MPI_MAX_PORT_NAME.

Advice to users. The system copies the port name into port_name. The application must pass a buffer of sufficient size to hold this value. (End of advice to users.)

port_name is essentially a network address. It is unique within the communication universe to which it belongs (determined by the implementation), and may be used by any client within that communication universe. For instance, if it is an internet (host:port) address, it will be unique on the internet. If it is a low level switch address on an IBM SP, it will be unique to that SP.

Advice to implementors. These examples are not meant to constrain implementations. A port_name could, for instance, contain a user name or the name of a batch job, as long as it is unique within some well-defined communication domain. The larger the communication domain, the more useful MPI's client/server functionality will be. (End of advice to implementors.)

The precise form of the address is implementation-defined. For instance, an internet address may be a host name or IP address, or anything that the implementation can decode into an IP address. A port name may be reused after it is freed with MPI_CLOSE_PORT and released by the system.

Advice to implementors. Since the user may type in port_name by hand, it is useful to choose a form that is easily readable and does not have embedded spaces. (End of advice to implementors.)

info may be used to tell the implementation how to establish the address. It may, and usually will, be MPI_INFO_NULL in order to get the implementation defaults.

MPI_CL	.OSE_PORT(port_name)	
IN	port_name	

a port (string)

C binding

```
1
     int MPI_Close_port(const char *port_name)
\mathbf{2}
     F08 binding
3
     MPI_Close_port(port_name, ierror)
4
          CHARACTER(LEN=*), INTENT(IN) :: port_name
5
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
     F binding
8
     MPI_CLOSE_PORT(PORT_NAME, IERROR)
9
          CHARACTER*(*) PORT_NAME
10
          INTEGER IERROR
11
12
13
     MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
14
       IN
                 port_name
                                             port name (string, significant only at root)
15
16
       IN
                 info
                                             implementation-dependent information (handle, sig-
17
                                             nificant only at root)
18
       IN
                                             rank in comm of root node (non-negative integer)
                 root
19
                                             intracommunicator over which call is collective (han-
       IN
                 comm
20
                                             dle)
21
22
       OUT
                 newcomm
                                             intercommunicator with client as remote group (han-
23
                                             dle)
^{24}
25
     C binding
26
     int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
27
                    MPI_Comm comm, MPI_Comm *newcomm)
28
     F08 binding
29
     MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
30
          CHARACTER(LEN=*), INTENT(IN) :: port_name
^{31}
          TYPE(MPI_Info), INTENT(IN) :: info
32
          INTEGER, INTENT(IN) :: root
33
          TYPE(MPI_Comm), INTENT(IN) ::
                                             comm
34
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
35
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     F binding
38
     MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
39
          CHARACTER*(*) PORT_NAME
40
          INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
41
          MPI_COMM_ACCEPT establishes communication with a client. It is collective over the
42
     calling communicator. It returns an intercommunicator that allows communication with the
43
     client.
44
         The port_name must have been established through a call to MPI_OPEN_PORT.
45
         info can be used to provide directives that may influence the behavior of the ACCEPT
46
     call.
47
48
```

10.4.3 Cl	ient Routines		1
There is only one routine on the client side.		2	
	ly one routine on the cheft si		3
		,	4 5
MPI_COMI	M_CONNECT(port_name, info	, root, comm, newcomm)	6
IN	port_name	network address (string, significant only at root)	7
IN	info	implementation-dependent information (handle, sig- nificant only at root)	8 9
IN	root	rank in comm of root node (non-negative integer)	10
IN	comm	intracommunicator over which call is collective (han- dle)	11 12 13
OUT	newcomm	intercommunicator with server as remote group (han-dle)	13 14 15
			16
C binding	-		17
int MPI_C	-	ort_name, MPI_Info info, int root,	18
	MPI_Comm comm, MPI_Co	mm *newcomm)	19 20
F08 bindi	0		20 21
	-	root, comm, newcomm, ierror)	22
	CHARACTER(LEN=*), INTENT(IN) :: port_name		
TYPE(MPI_Info), INTENT(IN) :: info			24
INTEGER, INTENT(IN) :: root TYPE(MPI_Comm), INTENT(IN) :: comm			25
TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(OUT) :: newcomm			26
	ER, OPTIONAL, INTENT(OUT)		27
			28
F binding			29 30
	CUNNECT(PORT_NAME, INFO, CTER*(*) PORT_NAME	ROOT, COMM, NEWCOMM, IERROR)	31
	ER INFO, ROOT, COMM, NEWC		32
			33
		tion with a server specified by port_name. It is	34
	9	and returns an intercommunicator in which the	35
0	up participated in an MPI_CC	has been closed), MPI_COMM_CONNECT raises	36
	class MPI_ERR_PORT.	has been closed), with COMMECT taises	37
		a pending MPI_COMM_ACCEPT, the connection	38
-	· ,	in implementation-defined time, or succeed when	39
-		n the case of a time out, MPI_COMM_CONNECT	40 41
raises an er	ror of class MPI_ERR_PORT .		42
Advie	e to implementors. The ti	me out period may be arbitrarily short or long.	43
		tation will try to queue connection attempts so	44
that a	a server can handle simultane	eous requests from several clients. A high-quality a mechanism, through the info arguments to	45 46

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.*) 48

Unofficial Draft for Comment Only

¹ MPI provides no guarantee of fairness in servicing connection attempts. That is, connec-² tion attempts are not necessarily satisfied in the order they were initiated and competition ³ from other connection attempts may prevent a particular connection attempt from being ⁴ satisfied.

port_name is the address of the server. It must be the same as the name returned
 by MPI_OPEN_PORT on the server. Some freedom is allowed here. If there are equivalent
 forms of port_name, an implementation may accept them as well. For instance, if port_name
 is (hostname:port), an implementation may accept (ip_address:port) as well.

10.4.4 Name Publishing

The routines in this section provide a mechanism for publishing names. A (service_name, 12port_name) pair is published by the server, and may be retrieved by a client using the 13 service_name only. An MPI implementation defines the scope of the service_name, that 14is, the domain over which the service_name can be retrieved. If the domain is the empty 15set, that is, if no client can retrieve the information, then we say that name publishing 16is not supported. Implementations should document how the scope is determined. High-17quality implementations will give some control to users through the info arguments to name 18 publishing functions. Examples are given in the descriptions of individual functions. 19

```
20
21
```

22

9 10

11

```
MPI_PUBLISH_NAME(service_name, info, port_name)
```

```
IN
                 service_name
                                             a service name to associate with the port (string)
23
^{24}
       IN
                 info
                                             implementation-specific information (handle)
25
       IN
                 port_name
                                             a port name (string)
26
27
     C binding
28
     int MPI_Publish_name(const char *service_name, MPI_Info info,
29
                     const char *port_name)
30
^{31}
     F08 binding
32
     MPI_Publish_name(service_name, info, port_name, ierror)
33
          CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
34
          TYPE(MPI_Info), INTENT(IN) :: info
35
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
36
     F binding
37
     MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
38
          CHARACTER*(*) SERVICE_NAME, PORT_NAME
39
          INTEGER INFO, IERROR
40
41
          This routine publishes the pair (port_name, service_name) so that an application may
42
     retrieve a system-supplied port_name using a well-known service_name.
43
          The implementation must define the scope of a published service name, that is, the
^{44}
     domain over which the service name is unique, and conversely, the domain over which 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.

Unofficial Draft for Comment Only

MPI permits publishing more than one service_name for a single port_name. On the other hand, if service_name has already been published within the scope determined by info, the behavior of MPI_PUBLISH_NAME is undefined. An MPI implementation may, through a mechanism in the info argument to MPI_PUBLISH_NAME, provide a way to allow multiple servers with the same service in the same scope. In this case, an implementation-defined policy will determine which of several port names is returned by MPI_LOOKUP_NAME.

Note that while service_name has a limited scope, determined by the implementation, port_name always has global scope within the communication universe used by the implementation (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.*)

			29
MPI_UI	NPUBLISH_NAME(service)	_name, info, port_name)	30
IN	service_name	a service name (string)	31
IN	info	implementation-specific information (handle)	32
IIN	inte	implementation specific information (nature)	33
IN	port_name	a port name (string)	34
			35
C bind	ling		36
int MP	I_Unpublish_name(const	t char *service_name, MPI_Info info,	37
	const char *po	rt_name)	38
F09 L:	a dia a		39
F08 bi	0		40
-	-	name, info, port_name, ierror)	41
		<pre>[(IN) :: service_name, port_name</pre>	42
	PE(MPI_Info), INTENT(1		43
IN	TEGER, OPTIONAL, INTEN	VT(OUT) :: ierror	44
F bind	ing		45
MPI_UN	PUBLISH_NAME(SERVICE_N	NAME, INFO, PORT_NAME, IERROR)	46
CH	ARACTER*(*) SERVICE_NA	AME, PORT_NAME	47
	TEGER INFO, IERROR		48

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

```
1
          This routine unpublishes a service name that has been previously published. Attempt-
\mathbf{2}
     ing to unpublish a name that has not been published or has already been unpublished is
3
     erroneous and is indicated by the error class MPI_ERR_SERVICE.
4
          All published names must be unpublished before the corresponding port is closed and
\mathbf{5}
     before the publishing process exits. The behavior of MPI_UNPUBLISH_NAME is implemen-
6
     tation dependent when a process tries to unpublish a name that it did not publish.
7
          If the info argument was used with MPI_PUBLISH_NAME to tell the implementation
8
     how to publish names, the implementation may require that info passed to
9
     MPI_UNPUBLISH_NAME contain information to tell the implementation how to unpublish
10
     a name.
11
12
     MPI_LOOKUP_NAME(service_name, info, port_name)
13
14
       IN
                 service_name
                                             a service name (string)
15
       IN
                 info
                                             implementation-specific information (handle)
16
       OUT
                 port_name
                                             a port name (string)
17
18
19
     C binding
     int MPI_Lookup_name(const char *service_name, MPI_Info info,
20
21
                     char *port_name)
22
     F08 binding
23
     MPI_Lookup_name(service_name, info, port_name, ierror)
^{24}
          CHARACTER(LEN=*), INTENT(IN) :: service_name
25
          TYPE(MPI_Info), INTENT(IN) :: info
26
          CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
27
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     F binding
30
     MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
^{31}
          CHARACTER*(*) SERVICE_NAME, PORT_NAME
32
          INTEGER INFO, IERROR
33
          This function retrieves a port_name published by MPI_PUBLISH_NAME with
34
     service_name. If service_name has not been published, it raises an error in the error class
35
     MPI_ERR_NAME. The application must supply a port_name buffer large enough to hold the
36
     largest possible port name (see discussion above under MPI_OPEN_PORT).
37
          If an implementation allows multiple entries with the same service_name within the
38
     same scope, a particular port_name is chosen in a way determined by the implementation.
39
          If the info argument was used with MPI_PUBLISH_NAME to tell the implementation
40
     how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME.
41
42
     10.4.5 Reserved Key Values
43
44
     The following key values are reserved. An implementation is not required to interpret these
45
     key values, but if it does interpret the key value, it must provide the functionality described.
46
47
     ip_port Value contains IP port number at which to establish a port. (Reserved for
48
           MPI_OPEN_PORT only).
```

ip_address Value contains IP address at which to establish a port. If the address is not a valid IP address of the host on which the MPI_OPEN_PORT call is made, the results are undefined. (Reserved for MPI_OPEN_PORT only).

10.4.6 Client/Server Examples

Simplest Example — Completely Portable.

The following example shows the simplest way to use the client/server interface. It does not use service names at all.

On the server side:

```
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;
/* ... */
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);
MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */
```

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:

```
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
```

Ocean/Atmosphere — Relies on Name Publishing

In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere climate model. It assumes that the MPI implementation publishes names.

MPI_Open_port(MPI_INFO_NULL, port_name); MPI_Publish_name("ocean", MPI_INFO_NULL, port_name); MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm); /* do something with intercomm */ MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name); On the client side: MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);

Unofficial Draft for Comment Only

¹ Simple Client-Server Example

This is a simple example; the server accepts only a single connection at a time and serves that connection until the client requests to be disconnected. The server is a single process. Here is the server. It accepts a single connection and then processes data until it

receives a message with tag 1. A message with tag 0 tells the server to exit.

```
7
     #include "mpi.h"
8
     int main(int argc, char *argv[])
9
     {
10
         MPI_Comm client;
11
         MPI_Status status;
12
         char port_name[MPI_MAX_PORT_NAME];
13
         double buf[MAX_DATA];
14
         int
                 size, again;
15
16
         MPI_Init(&argc, &argv);
17
         MPI_Comm_size(MPI_COMM_WORLD, &size);
18
         if (size != 1) error(FATAL, "Server too big");
19
         MPI_Open_port(MPI_INFO_NULL, port_name);
20
         printf("server available at %s\n", port_name);
21
         while (1) {
22
              MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
23
                                &client);
24
              again = 1;
25
              while (again) {
26
                  MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
27
                            MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
28
                  switch (status.MPI_TAG) {
29
                       case 0: MPI_Comm_free(&client);
30
                                MPI_Close_port(port_name);
31
                                MPI_Finalize();
32
                                return 0;
33
                       case 1: MPI_Comm_disconnect(&client);
34
                                again = 0;
35
                                break;
36
                       case 2: /* do something */
37
                       . . .
38
                       default:
39
                                /* Unexpected message type */
40
                                MPI_Abort(MPI_COMM_WORLD, 1);
41
                       }
42
                  }
43
              }
44
     }
45
         Here is the client.
46
47
     #include "mpi.h"
48
```

```
1
int main(int argc, char **argv)
                                                                                        \mathbf{2}
{
                                                                                        3
    MPI_Comm server;
    double buf [MAX_DATA];
                                                                                        4
    char port_name[MPI_MAX_PORT_NAME];
                                                                                        5
                                                                                        6
    MPI_Init(&argc, &argv);
                                                                                        7
    strcpy(port_name, argv[1]);/* assume server's name is cmd-line arg */
                                                                                        8
                                                                                        9
                                                                                       10
    MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                                                                                       11
                       &server);
                                                                                       12
    while (!done) {
                                                                                       13
                                                                                       14
        tag = 2; /* Action to perform */
                                                                                       15
        MPI_Send(buf, n, MPI_DOUBLE, 0, tag, server);
                                                                                       16
        /* etc */
                                                                                       17
                                                                                       18
    MPI_Send(buf, 0, MPI_DOUBLE, 0, 1, server);
                                                                                       19
    MPI_Comm_disconnect(&server);
                                                                                       20
    MPI_Finalize();
                                                                                       21
    return 0;
}
                                                                                       22
                                                                                       23
```

10.5 Other Functionality

10.5.1 Universe Size

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 34 the application to obtain this information in a portable manner. This attribute indicates 35the total number of processes that are expected. In Fortran, the attribute is the integer 36 value. In C, the attribute is a pointer to the integer value. An application typically subtracts 37 the size of MPI_COMM_WORLD from MPI_UNIVERSE_SIZE to find out how many processes it 38 should spawn. MPI_UNIVERSE_SIZE is initialized in MPI_INIT and is not changed by MPI. If 39 defined, it has the same value on all processes of MPI_COMM_WORLD. MPI_UNIVERSE_SIZE 40 is determined by the application startup mechanism in a way not specified by MPI. (The 41 size of MPI_COMM_WORLD is another example of such a parameter.) 42

Possibilities for how $\mathsf{MPI_UNIVERSE_SIZE}$ might be set include

- \bullet A <code>-universe_size</code> argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.

Unofficial Draft for Comment Only

24

25 26

27

28

29

30

31

32

33

43 44

 $45 \\ 46$

47

1

2

3 4

5

6

7

8

9

10

- An environment variable set by the user.
- Extra information passed to MPI_COMM_SPAWN through the info argument.

An implementation must document how MPI_UNIVERSE_SIZE is set. An implementation may not support the ability to set MPI_UNIVERSE_SIZE, in which case the attribute MPI_UNIVERSE_SIZE is not set.

MPI_UNIVERSE_SIZE is a recommendation, not necessarily a hard limit. For instance, some implementations may allow an application to spawn 50 processes per processor, if they are requested. However, it is likely that the user only wants to spawn one process per processor.

11 MPI_UNIVERSE_SIZE is assumed to have been specified when an application was started, 12and is in essence a portable mechanism to allow the user to pass to the application (through 13 the MPI process startup mechanism, such as **mpiexec**) a piece of critical runtime informa-14tion. Note that no interaction with the runtime environment is required. If the runtime 15environment changes size while an application is running, MPI_UNIVERSE_SIZE is not up-16dated, and the application must find out about the change through direct communication 17with the runtime system.

18 19

20

26

4243

44

45

48

10.5.2 Singleton MPI_INIT

21A 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 22 a process can then connect to other MPI processes using the MPI_COMM_ACCEPT and 23MPI_COMM_CONNECT routines, or spawn other MPI processes. MPI does not mandate 24this behavior, but strongly encourages it where technically feasible. 25

Advice to implementors. To start MPI processes belonging to the same

27MPI_COMM_WORLD requires some special coordination. The processes must be started 28at the "same" time, they must have a mechanism to establish communication, etc. 29 Either the user or the operating system must take special steps beyond simply starting 30 processes. 31

When an application enters MPI_INIT, clearly it must be able to determine if these 32 special steps were taken. If a process enters MPI_INIT and determines that no 33 34 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 pro-35 gram, that is, one in which MPI_COMM_WORLD has size 1. 36

37 In some implementations, MPI may not be able to function without an "MPI environ-38 ment." For example, MPI may require that daemons be running or MPI may not be 39 able to work at all on the front-end of an MPP. In this case, an MPI implementation 40 may either 41

- 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 46 an error. 47

(End of advice to implementors.)

10.5.3 1 MPI_APPNUM $\mathbf{2}$ There is a predefined attribute MPI_APPNUM of MPI_COMM_WORLD. In Fortran, the at-3 tribute is an integer value. In C, the attribute is a pointer to an integer value. If a process 4 was spawned with MPI_COMM_SPAWN_MULTIPLE, MPI_APPNUM is the command number 5that generated the current process. Numbering starts from zero. If a process was spawned 6 with MPI_COMM_SPAWN, it will have MPI_APPNUM equal to zero. 7 Additionally, if the process was not started by a spawn call, but by an implementation-8 specific startup mechanism that can handle multiple process specifications, MPI_APPNUM 9 should be set to the number of the corresponding process specification. In particular, if it 10 is started with 11 12mpiexec spec0 [: spec1 : spec2 : ...] 13 MPI_APPNUM should be set to the number of the corresponding specification. 14If an application was not spawned with MPI_COMM_SPAWN or 15MPI_COMM_SPAWN_MULTIPLE, and MPI_APPNUM does not make sense in the context of 16the implementation-specific startup mechanism, MPI_APPNUM is not set. 17 MPI implementations may optionally provide a mechanism to override the value of 18 MPI_APPNUM through the info argument. MPI reserves the following key for all SPAWN 19 calls. 2021appnum Value contains an integer that overrides the default value for MPI_APPNUM in the 22 child. 23 24 *Rationale.* When a single application is started, it is able to figure out how many pro-25cesses there are by looking at the size of MPI_COMM_WORLD. An application consisting 26of multiple SPMD sub-applications has no way to find out how many sub-applications 27there are and to which sub-application the process belongs. While there are ways to 28figure it out in special cases, there is no general mechanism. MPI_APPNUM provides 29such a general mechanism. (End of rationale.) 30 31 10.5.4 Releasing Connections 32 33 Before a client and server connect, they are independent MPI applications. An error in one 34does 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 3536 server to be able to disconnect, so that an error in one will not affect the other. Similarly, 37 it might be desirable for a parent and child to disconnect, so that errors in the child do not 38 affect the parent, or vice-versa. 39 • Two processes are **connected** if there is a communication path (direct or indirect) 40 between them. More precisely: 41 421. Two processes are connected if 43 (a) they both belong to the same communicator (inter- or intra-, including 44MPI_COMM_WORLD) or 45(b) they have previously belonged to a communicator that was freed with 46MPI_COMM_FREE instead of MPI_COMM_DISCONNECT or 4748

(c) they both belong to the group of the same window or filehandle.

Unofficial Draft for Comment Only

1	2. If A is connected to B and B to C, then A is connected to C.
2 3	• Two processes are disconnected (also independent) if they are not connected.
4 5 6 7	• By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.
8 9 10 11	• 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.
12	The following additional rules apply to MPI routines in other chapters:
13 14	• MPI_FINALIZE is collective over a set of connected processes.
15 16 17 18 19	• 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.
20 21 22 23	• If a process terminates without calling MPI_FINALIZE, independent processes are not affected but the effect on connected processes is not defined.
24 25 26	MPI_COMM_DISCONNECT(comm) INOUT comm communicator (handle)
27 28 29	C binding int MPI_Comm_disconnect(MPI_Comm *comm)
30 31 32 33 34	F08 binding MPI_Comm_disconnect(comm, ierror) TYPE(MPI_Comm), INTENT(INOUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35 36 37	F binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR
38 39 40	This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.
41 42 43 44	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.
$45 \\ 46$	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

waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.

1 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 2 3 communication paths between the two processes. Note that it may be necessary 4 to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) 56 *Rationale.* It would be nice to be able to use MPI_COMM_FREE instead, but that $\overline{7}$ function explicitly does not wait for pending communication to complete. (End of 8 rationale.) 9 10 11 10.5.5 Another Way to Establish MPI Communication 1213 14MPI_COMM_JOIN(fd, intercomm) 15IN fd socket file descriptor (integer) 16OUT intercomm new intercommunicator (handle) 1718 19 C binding 20int MPI_Comm_join(int fd, MPI_Comm *intercomm) 21F08 binding 22 MPI_Comm_join(fd, intercomm, ierror) 23INTEGER, INTENT(IN) :: fd 24 TYPE(MPI_Comm), INTENT(OUT) :: intercomm 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2627F binding 28 MPI_COMM_JOIN(FD, INTERCOMM, IERROR) 29 INTEGER FD, INTERCOMM, IERROR 30 3132 MPI_COMM_JOIN(fd, intercomm) 33 IN fd socket file descriptor 34 35 OUT intercomm new intercommunicator (handle) 36 37 C binding 38 int MPI_Comm_join(int fd, MPI_Comm *intercomm) 39 F08 binding 40 MPI_Comm_join(fd, intercomm, ierror) 41 INTEGER, INTENT(IN) :: fd 42TYPE(MPI_Comm), INTENT(OUT) :: intercomm 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 45F binding 46MPI_COMM_JOIN(FD, INTERCOMM, IERROR) 47INTEGER FD, INTERCOMM, IERROR 48

1	MPI_COMM_JOIN is intended for MPI implementations that exist in an environment
2	supporting the Berkeley Socket interface [45, 49]. Implementations that exist in an environ-
3	ment not supporting Berkeley Sockets should provide the entry point for MPI_COMM_JOIN
4	and should return MPI_COMM_NULL.

5This call creates an intercommunicator from the union of two MPI processes which are 6 connected by a socket. MPI_COMM_JOIN should normally succeed if the local and remote $\overline{7}$ 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 21byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must 22 not be enabled for the socket. The socket must be in a connected state. The socket must 23be quiescent when MPI_COMM_JOIN is called (see below). It is the responsibility of the 24application to create the socket using standard socket API calls. 25

MPI_COMM_JOIN must be called by the process at each end of the socket. It does not 26return until both processes have called MPI_COMM_JOIN. The two processes are referred 27to as the local and remote processes. 28

MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing 29 else. Upon return from MPI_COMM_JOIN, the file descriptor will be open and quiescent 30 (see below). 31

If MPI is unable to create an intercommunicator, but is able to leave the socket in its 32 original state, with no pending communication, it succeeds and sets intercomm to 33 MPI_COMM_NULL. 34

The socket must be quiescent before MPI_COMM_JOIN is called and after 35 MPI_COMM_JOIN returns. More specifically, on entry to MPI_COMM_JOIN, a read on the 36 socket will not read any data that was written to the socket before the remote process called 37 MPI_COMM_JOIN. On exit from MPI_COMM_JOIN, a read will not read any data that was 38 written to the socket before the remote process returned from MPI_COMM_JOIN. It is the 39 responsibility of the application to ensure the first condition, and the responsibility of the 40 MPI implementation to ensure the second. In a multithreaded application, the application 41 must ensure that one thread does not access the socket while another is calling 42MPI_COMM_JOIN, or call MPI_COMM_JOIN concurrently. 43

44Advice to implementors. MPI is free to use any available communication path(s)45for MPI messages in the new communicator; the socket is only used for the initial 46 handshaking. (End of advice to implementors.) 47

8 9

10 11

12

13 14

15

16

17

18

19 20

48

MPI_COMM_JOIN uses non-MPI communication to do its work. The interaction of	1
non-MPI communication with pending MPI communication is not defined. Therefore, the	2
result of calling MPI_COMM_JOIN on two connected processes (see Section 10.5.4 for the	3 4
definition of connected) is undefined. The returned communication may be used to establish MPI communication with addi	5
The returned communicator may be used to establish MPI communication with addi- tional processes, through the usual MPI communicator creation mechanisms.	6
tional processes, through the usual MFT communicator creation mechanisms.	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
	41
	42
	44
	45
	46
	47
	48

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.

Unofficial Draft for Comment Only

7 8 9

15 16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37 38

39 40

41

42

43 44

 $45 \\ 46$

47

1 MPI supports two fundamentally different *memory models*: separate and *unified*. The $\mathbf{2}$ separate model makes no assumption about memory consistency and is highly portable. 3 This model is similar to that of weakly coherent memory systems: the user must impose 4 correct ordering of memory accesses through synchronization calls. The unified model can $\mathbf{5}$ exploit cache-coherent hardware and hardware-accelerated, one-sided operations that are 6 commonly available in high-performance systems. The two different models are discussed 7in detail in Section 11.4. Both models support several synchronization calls to support 8 different synchronization styles.

9 The design of the RMA functions allows implementors to take advantage of fast or 10 asynchronous communication mechanisms provided by various platforms, such as coherent 11or noncoherent shared memory, DMA engines, hardware-supported put/get operations, and 12communication coprocessors. The most frequently used RMA communication mechanisms 13can be layered on top of message-passing. However, certain RMA functions might need 14support for asynchronous communication agents in software (handlers, threads, etc.) in a 15distributed memory environment.

16We shall denote by **origin** the process that performs the call, and by **target** the 17process in which the memory is accessed. Thus, in a put operation, source=origin and 18 destination=target; in a get operation, source=target and destination=origin.

1920

21

Initialization 11.2

22MPI provides the following window initialization functions: MPI_WIN_CREATE. 23

MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and

 24 MPI_WIN_CREATE_DYNAMIC, which are collective on an intracommunicator.

25MPI_WIN_CREATE allows each process to specify a "window" in its memory that is made 26accessible to accesses by remote processes. The call returns an opaque object that represents 27the group of processes that own and access the set of windows, and the attributes of each 28window, as specified by the initialization call. MPI_WIN_ALLOCATE differs from 29

MPI_WIN_CREATE in that the user does not pass allocated memory; 30

MPI_WIN_ALLOCATE returns a pointer to memory allocated by the MPI implementation. 31 MPI_WIN_ALLOCATE_SHARED differs from MPI_WIN_ALLOCATE in that the allocated 32 memory can be accessed from all processes in the window's group with direct load/store 33 instructions. Some restrictions may apply to the specified communicator. 34

MPI_WIN_CREATE_DYNAMIC creates a window that allows the user to dynamically control 35 which memory is exposed by the window. 36

Unofficial Draft for Comment Only

- 38
- 39
- 40 41
- 42

- 44
- 45
- 46
- 47
- 48

1

11.2.1	Window Creation		1
			2
	IN CREATE(base size		$^{3}_{4}$
	IN_CILATE(base, size,	. ,	5
IN	base	initial address of window (choice)	6
IN	size	size of window in bytes (non-negative integer)	7
IN	disp_unit	local unit size for displacements, in bytes (positive in- teger)	8 9
IN	info	info argument (handle)	10 11
IN	comm		12
OUT	win	window object returned by the call (handle)	13
			14
C bind	ling		15
int MP	I_Win_create(void *1	pase MPT Aint size int disp unit MPT Info info	16 17
	MPI_Comm com	m. MPT Win *win)	18
F08 bi	nding	:	19
		, disp_unit, info, comm, win, ierror)	20
TY	PE(*), DIMENSION()), ASYNCHRONOUS :: base	21
IN	TEGER(KIND=MPI_ADDRI	ESS_KIND), INTENT(IN) :: size	22
	TEGER, INTENT(IN) :	•	23
	PE(MPI_Info), INTEN		24
	PE(MPI_Comm), INTEN		25
	PE(MPI_Win), INTENT		26
IN	TEGER, OPTIONAL, INT		27
F bind	ing		28
	•	DISP UNIT INFO COMM WIN IFROR)	29
	ype> BASE(*)	· · · · · · · · · · · · · · · · · · ·	30
	TEGER(KIND=MPI_ADDR	ESS KIND) SIZE	31
		O COMM WIN TERBOR	32
		· · · · · · · · · · · · · · · · · · ·	33 34
		xecuted by an processes in the group of comm. It returns	35
	-	used by these processes to perform RMA operations. Each	36
-	-	existing memory that it exposes to runn accesses by the	37
		m. The window consists of size bytes, starting at address	38

base. In C, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 17.1.12). 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 47with a basic integer type. (End of rationale.) 48

Unofficial Draft for Comment Only

39

40

41

4243

	100	
1 2 3 4 5 6	sizeo latter	e to users. Common choices for disp_unit are 1 (no scaling), and (in C syntax) of (type), for a window that consists of an array of elements of type type. The choice will allow one to use array indices in RMA calls, and have those scaled etly to byte displacements, even in a heterogeneous environment. (<i>End of advice ers.</i>)
7 8		o argument provides optimization hints to the runtime about the expected usage he window. The following info keys are predefined:
9 10 11 12 13	nizati windo	if set to true, then the implementation may assume that passive target synchro- on (i.e., MPI_WIN_LOCK, MPI_WIN_LOCK_ALL) will not be used on the given w. This implies that this window is not used for 3-party communication, and can be implemented with no (less) asynchronous agent activity at this process.
14 15		ordering — controls the ordering of accumulate operations at the target. See on $11.7.2$ for details.
16 17 18 19 20 21 22	accum same_c calls t elimin	_ops — if set to same_op, the implementation will assume that all concurrent nulate calls to the same target address will use the same operation. If set to op_no_op, then the implementation will assume that all concurrent accumulate to the same target address will use the same operation or MPI_NO_OP. This can nate the need to protect access for certain operation types where the hardware marantee atomicity. The default is same_op_no_op.
23 24 25		- if set to true, then the implementation may assume that the argument size is cal on all processes, and that all processes have provided this info key with the value.
26 27 28 29	disp_u	unit — if set to true, then the implementation may assume that the argument unit is identical on all processes, and that all processes have provided this info ith the same value.
30 31 32 33	to que It is r	e to users. The info query mechanism described in Section 11.2.7 can be used ery the specified info arguments for windows that have been passed to a library. recommended that libraries check attached info keys for each passed window. of advice to users.)
34 35 36 37 38 39 40	windows, in put and acc should pose associated v	rious processes in the group of comm may specify completely different target a location, size, displacement units, and info arguments. As long as all the get, cumulate accesses to a particular process fit their specific target window this a no problem. The same area in memory may appear in multiple windows, each with a different window object. However, concurrent communications to distinct, a windows may lead to undefined results.
41 42 43 44 45 46 47 48	proces can be ification process imples	<i>nale.</i> The reason for specifying the memory that may be accessed from another as in an RMA operation is to permit the programmer to specify what memory e a target of RMA operations and for the implementation to enforce that spec- on. For example, with this definition, a server process can safely allow a client as to use RMA operations, knowing that (under the assumption that the MPI mentation does enforce the specified limits on the exposed memory) an error in ient cannot affect any memory other than what was explicitly exposed. (<i>End of tale.</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,	disp	_unit,	info,	comm,	baseptr,	win))

IN	J	size	size of window in bytes (non-negative integer)	28
IN	1	disp_unit	local unit size for displacements, in bytes (positive in-	29
II	N	disp_unit		30
			teger)	31
IN	١	info	info argument (handle)	32
IN	J	comm	intra-communicator (handle)	33
				34
0	UT	baseptr	initial address of window (choice)	35
0	UT	win	window object returned by the call (handle)	36
				37

C binding

F08 binding

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

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

38

39

40

41

42

43

44

45

46

47

1 2	TYPE(C_PTR), INTENT(OUT) :: baseptr TYPE(MPI_Win), INTENT(OUT) :: win
3	TYPE(MPI_Win), INTENT(OUT) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4	INTEGER, OFITONAL, INTENT(OUT) TETTOT
5	F binding
6	MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
7	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
8	INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
9	This is a collective call executed by all processes in the group of comm. On each
10	process, it allocates memory of at least size bytes, returns a pointer to it, and returns a
11	window object that can be used by all processes in comm to perform RMA operations. The
12	returned memory consists of size bytes local to each process, starting at address baseptr
13	and is associated with the window as if the user called MPI_WIN_CREATE on existing
14	memory. The size argument may be different at each process and size $= 0$ is valid; however, a
15	library might allocate and expose more memory in order to create a fast, globally symmetric
16	allocation. The discussion of and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in
17	Section 8.2 also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section 8.2
18	for an explanation of the type used for baseptr.
19	If the Fortran compiler provides TYPE (C_PTR), then the following generic interface must
20	be provided in the mpi module and should be provided in mpif.h through overloading,
21	i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
22	BASEPTR, but with a different specific procedure name:
23	
24	INTERFACE MPI_WIN_ALLOCATE
25	SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
26	WIN, IERROR)
27	
	IMPORT :: MPI_ADDRESS_KIND
28	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
28 29	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE
29	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
29 30	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR)
29 30 31 32 33	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
29 30 31 32 33 34	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND
29 30 31 32 33 34 35	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
29 30 31 32 33 34 35 36	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
29 30 31 32 33 34 35 36 37	<pre>INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &</pre>
29 30 31 32 33 34 35 36 37 38	<pre>INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &</pre>
29 30 31 32 33 34 35 36 37 38 39	<pre>INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &</pre>
29 30 31 32 33 34 35 36 37 38 39 40	<pre>INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &</pre>
29 30 31 32 33 34 35 36 37 38 39 40 41	<pre>INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &</pre>
29 30 31 32 33 34 35 36 37 38 39 40 41 42	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section 17.1.5.
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section 17.1.5. Rationale. By allocating (potentially aligned) memory instead of allowing the user
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section 17.1.5. <i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section 17.1.5. <i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section 17.1.5. <i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & WIN, IERROR) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR IMPORT :: MPI_ADDRESS_KIND INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE TYPE(C_PTR) :: BASEPTR END SUBROUTINE END INTERFACE The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section 17.1.5. <i>Rationale.</i> By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory

Unofficial Draft for Comment Only

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM.

11.2.3 Window That Allocates Shared Memory

MPI_WIN_ALLOCATE_SHARED(size, disp_unit, info, comm, baseptr, win)

				0
IN		size	size of local window in bytes (non-negative integer)	9
IN		disp_unit	local unit size for displacements, in bytes (positive in- teger)	10 11
IN		info	info argument (handle)	12
IN		comm	intra-communicator (handle)	13
				14
Οι	JT	baseptr	address of local allocated window segment (choice)	15
οι	іт	win	window object returned by the call (handle)	16
00	51	vviii	white work object retained by the can (nanale)	17

C binding

int	MPI_Win_allocate_s	shared(MPI_Aint	t size, int	disp_unit,	MPI_Info info,
	MPI_Comm	comm, void *b	aseptr, MPI	_Win *win)	

F08 binding

<pre>MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)</pre>
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
INTEGER, INTENT(IN) :: disp_unit
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(C_PTR), INTENT(OUT) :: baseptr
TYPE(MPI_Win), INTENT(OUT) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

```
MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

This is a collective call executed by all processes in the group of comm. On each process, it allocates memory of at least size bytes that is shared among all processes in comm, and returns a pointer to the locally allocated segment in **baseptr** that can be used for load/store accesses on the calling process. The locally allocated memory can be the target of load/store accesses by remote processes; the base pointers for other processes can be queried using the function MPI_WIN_SHARED_QUERY. The call also returns a window object that can be used by all processes in comm to perform RMA operations. The size argument may be different at each process and size = 0 is valid. It is the user's responsibility to ensure that the communicator comm represents a group of processes that can create a shared memory segment that can be accessed by all processes in the group. The discussions of rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in Section 8.2 also apply to MPI_WIN_ALLOCATE_SHARED; in particular, see the rationale in Section 8.2

Unofficial Draft for Comment Only

 $\mathbf{2}$

for an explanation of the type used for baseptr. The allocated memory is contiguous across process ranks unless the info key alloc_shared_noncontig is specified. Contiguous across process ranks means that the first address in the memory segment of process i is consecutive with the last address in the memory segment of process i - 1. This may enable the user to calculate remote address offsets with local information only.

If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
 be provided in the mpi module and should be provided in mpif.h through overloading,
 i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
 BASEPTR, but with a different specific procedure name:

```
    INTERFACE MPI_WIN_ALLOCATE_SHARED
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
```

```
BASEPTR, WIN, IERROR)
13
             IMPORT :: MPI_ADDRESS_KIND
14
             INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
15
             INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
16
         END SUBROUTINE
17
         SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
18
                                                   BASEPTR, WIN, IERROR)
19
             USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
20
             IMPORT :: MPI_ADDRESS_KIND
21
             INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
22
             INTEGER(KIND=MPI_ADDRESS_KIND) ::
                                                  SIZE
23
             TYPE(C_PTR) :: BASEPTR
24
         END SUBROUTINE
25
```

```
26 END INTERFACE
```

The base procedure name of this overloaded function is

MPI_WIN_ALLOCATE_SHARED_CPTR. The implied specific procedure names are described
 in Section 17.1.5.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE, MPI_WIN_ALLOCATE, and MPI_ALLOC_MEM. The additional info key alloc_shared_noncontig allows the library to optimize the layout of the shared memory segments in memory.

Advice to users. If the info key alloc_shared_noncontig is not set to true, the allocation strategy is to allocate contiguous memory across process ranks. This may limit the performance on some architectures because it does not allow the implementation to modify the data layout (e.g., padding to reduce access latency). (End of advice to users.)

Advice to implementors. If the user sets the info key alloc_shared_noncontig to true, the implementation can allocate the memory requested by each process in a location that is close to this process. This can be achieved by padding or allocating memory in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. (*End of advice to implementors.*)

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the *unified*

Unofficial Draft for Comment Only

10

27

28

35

36

37

38

39

0	`	4) by utilizing the window synchronization functions (see	1	
Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling			2 3	
MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows in the <i>separate memory model</i> .			3 4	
In the set	baraie memory moaei.		5	
			6	
MPI_WIN	I_SHARED_QUERY(wi	n, rank, size, disp_unit, baseptr)	7	
IN	win	shared memory window object (handle)	8	
IN	rank	rank in the group of window win (non-negative inte-	9	
		ger) or MPI_PROC_NULL	10	
OUT	size	size of the window segment (non-negative integer)	11 12	
OUT	disp_unit	local unit size for displacements, in bytes (positive in-	13	
001	disp_diff	teger)	14	
OUT	baseptr	address for load/store access to window segment	15	
001	basepti	(choice)	16	
			17	
C bindi	ng		18 19	
		PI_Win win, int rank, MPI_Aint *size,	20	
	int *disp_uni	t, void *baseptr)	21	
F08 bin	ding		22	
	0	rank, size, disp_unit, baseptr, ierror)	23	
	1 V	_C_BINDING, ONLY : C_PTR	24	
	E(MPI_Win), INTENT()		25	
	EGER, INTENT(IN) ::		26	
		SS_KIND), INTENT(OUT) :: size	27 28	
	EGER, INTENT(OUT) :	1	20	
	C(C_PTR), INTENT(OU	ENT(OUT) :: ierror	30	
			31	
F bindir	0		32	
		RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)	33	
	INTEGER WIN, RANK, DISP_UNIT, IERROR INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR			
	GER (RIND-HFI_ADDR)	LOD_KIND/ DIZE, DROEFIR	35	
This	function quaries the	process local address for remote memory segments created	36	

This function queries the process-local address for remote memory segments created with MPI_WIN_ALLOCATE_SHARED. This function can return different process-local addresses for the same physical memory on different processes. The returned memory can be used for load/store accesses subject to the constraints defined in Section 11.7. This function can only be called with windows of flavor MPI_WIN_FLAVOR_SHARED. If the passed window is not of flavor MPI_WIN_FLAVOR_SHARED, the error MPI_ERR_RMA_FLAVOR is raised. When rank is MPI_PROC_NULL, the pointer, disp_unit, and size returned are the pointer, disp_unit, and size of the memory segment belonging the lowest rank that specified size > 0. If all processes in the group attached to the window specified size = 0, then the call returns size = 0 and a baseptr as if MPI_ALLOC_MEM was called with size = 0.

If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading,

47 48

37

38

39

40

 41

42

43

44

45

```
1
     i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
\mathbf{2}
     BASEPTR, but with a different specific procedure name:
3
4
     INTERFACE MPI_WIN_SHARED_QUERY
          SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
5
6
                                              BASEPTR, IERROR)
              IMPORT :: MPI_ADDRESS_KIND
7
              INTEGER WIN, RANK, DISP_UNIT, IERROR
8
              INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
9
10
          END SUBROUTINE
11
          SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
                                                   BASEPTR, IERROR)
12
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
13
              IMPORT :: MPI_ADDRESS_KIND
14
              INTEGER :: WIN, RANK, DISP_UNIT, 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
21
     MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in
22
     Section 17.1.5.
23
^{24}
     11.2.4
            Window of Dynamically Attached Memory
25
26
     The MPI-2 RMA model requires the user to identify the local memory that may be a
27
     target of RMA calls at the time the window is created. This has advantages for both
28
     the programmer (only this memory can be updated by one-sided operations and provides
29
     greater safety) and the MPI implementation (special steps may be taken to make one-
30
     sided access to such memory more efficient). However, consider implementing a modifiable
^{31}
     linked list using RMA operations; as new items are added to the list, memory must be
32
     allocated. In a C or C++ program, this memory is typically allocated using malloc or
33
     new respectively. In MPI-2 RMA, the programmer must create a window with a predefined
34
     amount of memory and then implement routines for allocating memory from within the
35
     window's memory. In addition, there is no easy way to handle the situation where the
36
     predefined amount of memory turns out to be inadequate. To support this model, the
37
     routine MPI_WIN_CREATE_DYNAMIC creates a window that makes it possible to expose
38
     memory without remote synchronization. It must be used in combination with the local
39
     routines MPI_WIN_ATTACH and MPI_WIN_DETACH.
40
41
42
     MPI_WIN_CREATE_DYNAMIC(info, comm, win)
43
       IN
                                             info argument (handle)
                 info
44
       IN
                 comm
                                            intra-communicator (handle)
45
46
       OUT
                                             window object returned by the call (handle)
                 win
47
48
```

C binding int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win) F08 binding MPI_Win_create_dynamic(info, comm, win, ierror) TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Win), INTENT(OUT) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR) INTEGER INFO, COMM, WIN, IERROR This is a collective call executed by all processes in the group of comm. It returns

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 $\mathsf{MPI}_\mathsf{WIN}_\mathsf{CREATE}.$

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.

1

 $\mathbf{2}$

3

4

5

6

7

9 10

11

12 13

14

15

16

17

18

19

20

21

22

23

24

25

26 27

28

29

30

31 32

33

34

35

36

37

38

39 40

41

42

43

44

```
1
     MPI_WIN_ATTACH(win, base, size)
2
       IN
                 win
                                             window object (handle)
3
       IN
                 base
                                             initial address of memory to be attached
4
5
       IN
                 size
                                             size of memory to be attached in bytes
6
7
     C binding
8
     int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
9
     F08 binding
10
     MPI_Win_attach(win, base, size, ierror)
11
          TYPE(MPI_Win), INTENT(IN) :: win
12
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
13
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                                size
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     F binding
17
     MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
18
          INTEGER WIN, IERROR
19
          <type> BASE(*)
20
          INTEGER (KIND=MPI_ADDRESS_KIND) SIZE
21
          Attaches a local memory region beginning at base for remote access within the given
22
     window. The memory region specified must not contain any part that is already attached
23
     to the window win, that is, attaching overlapping memory concurrently within the same
24
     window is erroneous. The argument win must be a window that was created with
25
     MPI_WIN_CREATE_DYNAMIC. The local memory region attached to the window consists
26
     of size bytes, starting at address base. In C, base is the starting address of a memory region.
27
     In Fortran, one can pass the first element of a memory region or a whole array, which
28
     must be 'simply contiguous' (for 'simply contiguous,' see Section 17.1.12). Multiple (but
29
     non-overlapping) memory regions may be attached to the same window.
30
^{31}
           Rationale.
                        Requiring that memory be explicitly attached before it is exposed to
32
           one-sided access by other processes can simplify implementations and improve perfor-
33
           mance. The ability to make memory available for RMA operations without requiring a
34
           collective MPI_WIN_CREATE call is needed for some one-sided programming models.
35
           (End of rationale.)
36
37
           Advice to users.
                              Attaching memory to a window may require the use of scarce
           resources; thus, attaching large regions of memory is not recommended in portable
38
           programs. Attaching memory to a window may fail if sufficient resources are not
39
           available; this is similar to the behavior of MPI_ALLOC_MEM.
40
41
           The user is also responsible for ensuring that MPI_WIN_ATTACH at the target has
42
           returned before a process attempts to target that memory with an MPI RMA call.
43
           Performing an RMA operation to memory that has not been attached to a window
44
           created with MPI_WIN_CREATE_DYNAMIC is erroneous. (End of advice to users.)
45
46
                                     A high-quality implementation will attempt to make as
           Advice to implementors.
47
           much memory available for attaching as possible. Any limitations should be docu-
48
           mented by the implementor. (End of advice to implementors.)
```

Unofficial Draft for Comment Only

Attaching memory is a local operation as defined by MPI, which means that the call is not collective and completes without requiring any MPI routine to be called in any other process. Memory may be detached with the routine MPI_WIN_DETACH. After memory has been detached, it may not be the target of an MPI RMA operation on that window (unless the memory is re-attached with MPI_WIN_ATTACH).

MPI_WIN_DETACH(win, base) IN window object (handle) win IN initial address of memory to be detached base C binding int MPI_Win_detach(MPI_Win win, const void *base) F08 binding MPI_Win_detach(win, base, ierror) TYPE(MPI_Win), INTENT(IN) :: win TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_WIN_DETACH(WIN, BASE, IERROR)

INTEGER WIN, IERROR <type> BASE(*)

Detaches a previously attached memory region beginning at base. The arguments base and win must match the arguments passed to a previous call to MPI_WIN_ATTACH.

Advice to users. Detaching memory may permit the implementation to make more efficient use of special memory or provide memory that may be needed by a subsequent MPI_WIN_ATTACH. Users are encouraged to detach memory that is no longer needed. Memory should be detached before it is freed by the user. (*End of advice to users.*)

Memory becomes detached when the associated dynamic memory window is freed, see Section 11.2.5.

11.2.5 Window Destruction

```
MPI_WIN_FREE(win)
    INOUT win window object (handle)
C binding
    int MPI_Win_free(MPI_Win *win)
F08 binding
MPI_Win_free(win, ierror)
    TYPE(MPI_Win), INTENT(INOUT) :: win
```

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	INTEGER, OPTIONAL, INTENT(OUT) ::	ierror	
2			
3	F binding		
4	MPI_WIN_FREE(WIN, IERROR)		
5	INTEGER WIN, IERROR		
6	Frees the window object win and returns	a null handle (equal to MPI_WIN_NULL). This	
7	is a collective call executed by all processes		
8		by a process only after it has completed its	
9	involvement in RMA communications on win		
10		to match a previous call to MPI_WIN_POST	
11		a previous call to MPI_WIN_START or called	
12		l to MPI_WIN_LOCK. The memory associated	
13	-	CREATE may be freed after the call returns. If	
14		OCATE, MPI_WIN_FREE will free the window	
15	memory that was allocated in MPI_WIN_AL		
16	MPI_WIN_ALLOCATE_SHARED, MPI_WIN_	FREE will free the window memory that was	
17	allocated in MPI_WIN_ALLOCATE_SHARED).	
18	Freeing a window that was created with	h a call to MPI_WIN_CREATE_DYNAMIC de-	
19	taches all associated memory; i.e., it has the	ne same effect as if all attached memory was	
20	detached by calls to MPI_WIN_DETACH.		
21			
22	·	FREE requires a barrier synchronization: no	
23	process can return from free until all		
24	-	occess will attempt to access a remote window	
25	(e.g., with lock/unlock) after it was freed. The only exception to this rule is when the		
26 27		hen creating the window. In that case, an MPI	
28	advice to implementors.)	dow without barrier synchronization. (End of	
29	uavice to implementors.)		
30	11.2.6 Window Attributes		
31	11.2.0 Window Attributes		
32	The following attributes are cached with a w	vindow when the window is created.	
33			
34		vindow base address.	
35		vindow size, in bytes.	
36		lisplacement unit associated with the window.	
37		now the window was created. nemory model for window.	
38		-	
39	In C, calls to $MPI_Win_get_attr(win, MF)$	e , , ,	
40	MPI_Win_get_attr(win, MPI_WIN_SIZE, &siz		
41	MPI_Win_get_attr(win, MPI_WIN_DISP_UNI	-,.	
42	MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag), and		
43	MPI_Win_get_attr(win, MPI_WIN_MODEL, &memory_model, &flag) will return in base a		
44		will return in size, disp_unit, create_kind, and	
45		ment unit of the window, the kind of routine	
46	,	model, respectively. A detailed listing of the	
47	type of the pointer in the attribute value ar		
48	MPI_WIN_SET_ATTR is shown in Table 11.1		

Attribute	C Type
MPI_WIN_BASE	void *
MPI_WIN_SIZE	MPI_Aint *
MPI_WIN_DISP_UNIT	int *
MPI_WIN_CREATE_FLAVOR	int *
MPI_WIN_MODEL	int *

Table 11.1: C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR.

In Fortran, calls to MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, ierror),
MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror),
MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror),
MPI_WIN_GET_ATTR(win, MPI_WIN_CREATE_FLAVOR, create_kind, flag, ierror), and
MPI_WIN_GET_ATTR(win, MPI_WIN_MODEL, memory_model, flag, ierror) will return in
base, size, disp_unit, create_kind, and memory_model the (integer representation of) the
base address, the size, the displacement unit of the window win, the kind of routine used to
create the window, and the memory model, respectively.

The values of create_kind are

MPI_WIN_FLAVOR_CREATE	Window was created with MPI_WIN_CREATE.	22
MPI_WIN_FLAVOR_ALLOCATE	Window was created with	23
	MPI_WIN_ALLOCATE.	24
MPI_WIN_FLAVOR_DYNAMIC	Window was created with	25
	MPI_WIN_CREATE_DYNAMIC.	26
MPI_WIN_FLAVOR_SHARED	Window was created with	27
	MPI_WIN_ALLOCATE_SHARED.	28

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)			40
IN	win	window object (handle)	41 42
OUT	group	group of processes which share access to the window (handle)	43 44
			45
C binding			46

int MPI_Win_get_group(MPI_Win win, MPI_Group *group)

 31

 $47 \\ 48$

1	F08 bind	ing		
2		get_group(win, group, ier	ror)	
3		(MPI_Win), INTENT(IN) ::		
4		(MPI_Group), INTENT(OUT)		
5		GER, OPTIONAL, INTENT(OUT		
6			,	
7	F binding			
8		GET_GROUP(WIN, GROUP, IER	ROR)	
9	INTEC	GER WIN, GROUP, IERROR		
10	MPL	WIN GET GROUP returns a o	luplicate of the group of the communicator used to	
11			The group is returned in group.	
12				
13	1127 W	/indow Info		
14				
15	-	· · · · · · · · · · · · · · · · · · ·	allow a user to provide information to direct opti-	
16			n implementation to deliver increased performance	
17	-		y. An implementation is free to ignore all hints;	
18			any info hints they provide that are used by the	
19	-		by a call to MPI_WIN_GET_INFO) and that place	
20			lication. Hints are specified on a per window basis,	
21			WIN_SET_INFO, via the opaque info object. When	
22			valid hints is passed to MPI_WIN_SET_INFO there	
23	will be no	effect on previously set or de	fault hints that the info does not specify.	
24	Advi	ce to implementors. It may	happen that a program is coded with hints for one	
25		*	ther system that does not support these hints. In	
26	ě		simply be ignored. Needless to say, no hint can be	
27	-		t used by a specific implementation, a default value	
28 29			oes not specify a value for the hint. (End of advice	
30		nplementors.)		
31				
32				
33		_SET_INFO(win, info)		
34		. ,		
35	INOUT	win	window object (handle)	
36	IN	info	info object (handle)	
37				
38	C bindin	g		
39	int MPI_W	Vin_set_info(MPI_Win win,	MPI_Info info)	
40	FOR hind	•		
41	F08 bind	8	r)	
42	<pre>MPI_Win_set_info(win, info, ierror) TYPE(MPI_Win), INTENT(IN) :: win</pre>			
43	TYPE(MPI_WIN), INTENT(IN) :: WIN TYPE(MPI_Info), INTENT(IN) :: info			
44	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
45			, 101101	
46	F binding			
47		SET_INFO(WIN, INFO, IERRO	R)	
48	INTEC	GER WIN, INFO, IERROR		

MPI_WIN_SET_INFO updates the hints of the window associated with win using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI_WIN_SET_INFO. The call is collective on the group of win. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Advice to users. Some info items that an implementation can use when it creates a window cannot easily be changed once the window has been created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI_WIN_SET_INFO. MPI_WIN_GET_INFO can be used to determine whether info changes were ignored by the implementation. (*End of advice to users.*)

MPI_WIN_GET_INFO(win, info_used)

IN	win	window object (handle)
OUT	info_used	new info object (handle)

C binding

int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)

```
F08 binding
```

```
MPI_Win_get_info(win, info_used, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Info), INTENT(OUT) :: info_used
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

```
MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
INTEGER WIN, INFO_USED, IERROR
```

MPI_WIN_GET_INFO returns a new info object containing the hints of the window associated with win. The current setting of all hints related to this window is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

11.3 Communication Calls

MPI supports the following RMA communication calls: MPI_PUT and MPI_RPUT transfer 44 data from the caller memory (origin) to the target memory; MPI_GET and MPI_RGET 45 transfer data from the target memory to the caller memory; MPI_ACCUMULATE and 46 MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these locations values sent from the caller memory; MPI_GET_ACCUMULATE, 48

Unofficial Draft for Comment Only

 $\overline{7}$

1 MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP perform atomic read-modify-write $\mathbf{2}$ and return the data before the accumulate operation; and MPI_COMPARE_AND_SWAP per-3 forms a remote atomic compare and swap operation. These operations are *nonblocking*: the 4 call initiates the transfer, but the transfer may continue after the call returns. The transfer $\mathbf{5}$ is completed, at the origin or both the origin and the target, when a subsequent synchro-6 *nization* call is issued by the caller on the involved window object. These synchronization $\overline{7}$ calls are described in Section 11.5. Transfers can also be completed with calls to flush rou-8 tines; see Section 11.5.4 for details. For the MPI_RPUT, MPI_RGET, MPI_RACCUMULATE, 9 and MPI_RGET_ACCUMULATE calls, the transfer can be locally completed by using the 10 MPI test or wait operations described in Section 3.7.3.

¹¹ The local communication buffer of an RMA call should not be updated, and the local ¹² communication buffer of a get call should not be accessed after the RMA call until the ¹³ operation completes at the origin.

14The resulting data values, or outcome, of concurrent conflicting accesses to the same 15memory locations is undefined; if a location is updated by a put or accumulate operation, 16then the outcome of loads or other RMA operations is undefined until the updating operation 17has completed at the target. There is one exception to this rule; namely, the same location 18 can be updated by several concurrent accumulate calls, the outcome being as if these updates 19occurred in some order. In addition, the outcome of concurrent load/store and RMA updates 20to the same memory location is undefined. These restrictions are described in more detail 21in 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.*)

MPI_PROC_NULL is a valid target rank in all MPI RMA communication calls. The effect is the same as for MPI_PROC_NULL in MPI point-to-point communication. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch.

11.3.1 Put

The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target process. The obvious difference is that all arguments are provided by one call — the call executed by the origin process.

43

28

29

30

31 32

37

- 44
- 45
- 46
- 47 48

MPI_P	UT(origin_addr, origin_count, target_datatype, wir	origin_datatype, target_rank, target_disp, target_count,	1 2
IN	origin_addr	initial address of origin buffer (choice)	3
IN	origin_count	number of entries in origin buffer (non-negative inte- ger)	4 5 6
IN	origin_datatype	datatype of each entry in origin buffer (handle)	7
IN	target_rank	rank of target (non-negative integer)	8
IN	target_disp	displacement from start of window to target buffer (non-negative integer)	9 10 11
IN	target_count	number of entries in target buffer (non-negative integer)	12 13
IN	target_datatype	datatype of each entry in target buffer (handle)	14
IN	win	window object used for communication (handle)	15 16
F08 b	PI_Put(const void *origin MPI_Datatype ori MPI_Aint target_ MPI_Datatype tar inding ut(origin_addr, origin_co	n_addr, int origin_count, gin_datatype, int target_rank, disp, int target_count, get_datatype, MPI_Win win) ount, origin_datatype, target_rank,	18 19 20 21 22 23 24 25
IN TY IN TY	<pre>'PE(*), DIMENSION(), II 'TEGER, INTENT(IN) :: of 'PE(MPI_Datatype), INTEN'</pre>		26 27 28 29 30 31 32
<t IN</t 	T(ORIGIN_ADDR, ORIGIN_CO TARGET_DISP, TAR ype> ORIGIN_ADDR(*) TEGER(KIND=MPI_ADDRESS_H	GIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	32 33 34 35 36 37 38 39 40
	-	ve entries of the type specified by the origin_datatype, he origin node, to the target node specified by the win,	40 41 42

starting at address origin_addr on the origin node, to the target node specified by the win, target_rank pair. The data are written in the target buffer at address target_addr = window_base+target_disp×disp_unit, where window_base and disp_unit are the base address and window displacement unit specified at window initialization, by the target process.

The target buffer is specified by the arguments target_count and target_datatype.

The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin_addr, origin_count, origin_datatype, target_rank, tag,

Unofficial Draft for Comment Only

43

44

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.

⁵ The communication must satisfy the same constraints as for a similar message-passing ⁶ communication. The target_datatype may not specify overlapping entries in the target ⁷ buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, ⁸ the target buffer must fit in the target window or in attached memory in a dynamic window.

⁹ The target_datatype argument is a handle to a datatype object defined at the origin ¹⁰ process. However, this object is interpreted at the target process: the outcome is as if ¹¹ the target datatype object was defined at the target process by the same sequence of calls ¹² used to define it at the origin process. The target datatype must contain only relative ¹³ displacements, not absolute addresses. The same holds for get and accumulate operations.

14

27

28

29

30

 31

32

33

34

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.*)

Similar to MPI_PUT, except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The origin_datatype may not specify overlapping entries in the origin buffer. The target buffer must be contained within the target window or within attached memory in a dynamic window, and the copied data must fit, without truncation, in the origin buffer.

Unofficial Draft for Comment Only

5

6

7

8

9 10

11

12

13

14 15

16

17

18

19 20 21

22

23

 24

25

26

27

28

29

30

31

32

33

34

35 36

37

38

39

40

41

42

```
1
     11.3.3 Examples for Communication Calls
\mathbf{2}
     These examples show the use of the MPI_GET function. As all MPI RMA communication
3
     functions are nonblocking, they must be completed. In the following, this is accomplished
4
     with the routine MPI_WIN_FENCE, introduced in Section 11.5.
5
6
     Example 11.1 We show how to implement the generic indirect assignment A = B(map),
7
     where A, B, and map have the same distribution, and map is a permutation. To simplify, we
8
     assume a block distribution with equal size blocks.
9
10
     SUBROUTINE MAPVALS(A, B, map, m, comm, p)
11
     USE MPI
12
     INTEGER m, map(m), comm, p
13
     REAL A(m), B(m)
14
15
     INTEGER otype(p), oindex(m),
                                      & ! used to construct origin datatypes
16
                                  & ! used to construct target datatypes
           ttype(p), tindex(m),
17
           count(p), total(p),
                                       &
18
           disp_int, win, ierr
19
     INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
20
21
     ! This part does the work that depends on the locations of B.
22
     ! Can be reused while this does not change
23
^{24}
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
25
     disp_int = realextent
26
     size = m * realextent
27
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                                  &
28
                           comm, win, ierr)
29
30
     ! This part does the work that depends on the value of map and
31
     ! the locations of the arrays.
32
     ! Can be reused while these do not change
33
34
     ! Compute number of entries to be received from each process
35
36
     DO i=1,p
37
       count(i) = 0
38
     END DO
39
     DO i=1,m
40
       j = map(i)/m+1
41
       count(j) = count(j)+1
42
     END DO
43
44
     total(1) = 0
45
     DO i=2,p
46
       total(i) = total(i-1) + count(i-1)
47
     END DO
48
```

```
1
DO i=1,p
                                                                                      \mathbf{2}
  count(i) = 0
                                                                                      3
END DO
                                                                                      4
! compute origin and target indices of entries.
                                                                                      5
                                                                                      6
! entry i at current process is received from location
                                                                                      7
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
                                                                                      8
! j = 1...p and k = 1...m
                                                                                      9
                                                                                      10
DO i=1,m
                                                                                      11
  j = map(i)/m+1
  k = MOD(map(i), m) + 1
                                                                                      12
  count(j) = count(j)+1
                                                                                      13
                                                                                      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, &
                                        oindex(total(i)+1:total(i)+count(i)), &
                                                                                      21
                                        MPI_REAL, otype(i), ierr)
                                                                                      22
                                                                                      23
  CALL MPI_TYPE_COMMIT(otype(i), ierr)
                                                                                      24
  CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                                                                      25
                                        tindex(total(i)+1:total(i)+count(i)), &
                                                                                      26
                                        MPI_REAL, ttype(i), ierr)
  CALL MPI_TYPE_COMMIT(ttype(i), ierr)
                                                                                      27
END DO
                                                                                      28
                                                                                      29
                                                                                      30
! this part does the assignment itself
                                                                                      31
CALL MPI_WIN_FENCE(0, win, ierr)
                                                                                      32
disp_aint = 0
                                                                                      33
DO i=1,p
                                                                                      34
  CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
END DO
                                                                                      35
CALL MPI_WIN_FENCE(0, win, ierr)
                                                                                      36
                                                                                      37
                                                                                      38
CALL MPI_WIN_FREE(win, ierr)
                                                                                      39
DO i=1,p
  CALL MPI_TYPE_FREE(otype(i), ierr)
                                                                                      40
                                                                                      41
  CALL MPI_TYPE_FREE(ttype(i), ierr)
                                                                                      42
END DO
RETURN
                                                                                      43
                                                                                      44
END
                                                                                      45
                                                                                      46
Example 11.2
                                                                                      47
                                                                                      48
```

1 A simpler version can be written that does not require that a datatype be built for the $\mathbf{2}$ target buffer. But, one then needs a separate get call for each entry, as illustrated below. 3 This code is much simpler, but usually much less efficient, for large arrays. 4 SUBROUTINE MAPVALS(A, B, map, m, comm, p) 56 USE MPI 7INTEGER m, map(m), comm, p REAL A(m), B(m)8 9 INTEGER disp_int, win, ierr 10 INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint 11CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr) 12disp_int = realextent 1314size = m * realextent CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL, & 1516comm, win, ierr) 1718CALL MPI_WIN_FENCE(0, win, ierr) 19DO i=1,m j = map(i)/m20disp_aint = MOD(map(i),m) 21CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL, win, ierr) 2223END DO 24 CALL MPI_WIN_FENCE(0, win, ierr) CALL MPI_WIN_FREE(win, ierr) 2526RETURN END 27282911.3.4 Accumulate Functions 30 It is often useful in a put operation to combine the data moved to the target process with the

³¹ It is often useful in a put operation to combine the data moved to the target process with the ³² data that resides at that process, rather than replacing it. This will allow, for example, the ³³ accumulation of a sum by having all involved processes add their contributions to the sum ³⁴ variable in the memory of one process. The accumulate functions have slightly different ³⁵ semantics with respect to overlapping data accesses than the put and get functions; see ³⁶ Section 11.7 for details.

- 37 38
- 39 40
- 41
- 42 43

- 45
- 46
- 46 47
- . 48

			2
			3 4
MPI_AC	. –	rigin_count, origin_datatype, target_rank, target_disp, t_datatype, op, win)	5
IN	origin_addr	initial address of buffer (choice)	6 7
IN	origin_count	number of entries in buffer (non-negative integer)	8
IN	origin_datatype	datatype of each entry (handle)	9
IN	target_rank	rank of target (non-negative integer)	10 11
IN	target_disp	displacement from start of window to beginning of tar- get buffer (non-negative integer)	11 12 13
IN	target_count	number of entries in target buffer (non-negative integer)	14 15
IN	target_datatype	datatype of each entry in target buffer (handle)	16 17
IN	ор	reduce operation (handle)	18
IN	win	window object (handle)	19 20
F08 bii	MPI_Aint target, MPI_Datatype ta	igin_datatype, int target_rank, _disp, int target_count, rget_datatype, MPI_Op op, MPI_Win win)	23 24 25 26 27
MPI_Acc TYF INT TYF INT TYF TYF	cumulate(origin_addr, o target_disp, ta PE(*), DIMENSION(), I TEGER, INTENT(IN) :: o PE(MPI_Datatype), INTEN	:: win	28 29 30 31 32 33 34 35 36
F bind MPI_ACC <ty INT INT</ty 	ing CUMULATE(ORIGIN_ADDR, C TARGET_DISP, TA 7Pe> ORIGIN_ADDR(*) TEGER(KIND=MPI_ADDRESS_ TEGER ORIGIN_COUNT, ORI TARGET_DATATYPE	DRIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, RGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR) KIND) TARGET_DISP GGIN_DATATYPE,TARGET_RANK, TARGET_COUNT, E, OP, WIN, IERROR	37 38 39 40 41 42 43 44
		e origin buffer (as defined by origin_addr, origin_count, and ified by arguments target_count and target_datatype, at	45 46

Unofficial Draft for Comment Only

offset target_disp, in the target window specified by target_rank and win, using the operation

1

1	op. This is like MPI_PUT except that data is combined into the target area instead of
2	overwriting it.
3	Any of the predefined operations for MPI_REDUCE can be used. User-defined functions
4	cannot be used. For example, if op is MPI_SUM, each element of the origin buffer is added
5	to the corresponding element in the target, replacing the former value in the target.
6	Each datatype argument must be a predefined datatype or a derived datatype, where
7	all basic components are of the same predefined datatype. Both datatype arguments must
8	be constructed from the same predefined datatype. The operation op applies to elements of
9	that predefined type. The parameter target_datatype must not specify overlapping entries,
10 11	and the target buffer must fit in the target window.
11	A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative
12	function $f(a,b) = b$; i.e., the current value in the target memory is replaced by the value
13	supplied by the origin.
15	MPI_REPLACE can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE, MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but not
16	in collective reduction operations such as MPI_REDUCE.
17	In conective reduction operations such as With_REDUCE.
18	Advice to users. MPI_PUT is a special case of MPI_ACCUMULATE, with the op-
19	eration MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have
20	different constraints on concurrent updates. (End of advice to users.)
21	
22	Example 11.3 We want to compute $B(j) = \sum_{map(i)=j} A(i)$. The arrays A, B, and map
23	are distributed in the same manner. We write the simple version.
24	are asserbated in the same mainter. We write the simple version,
25	SUBROUTINE SUM(A, B, map, m, comm, p)
26	USE MPI
26 27	USE MPI INTEGER m, map(m), comm, p, win, ierr, disp_int
26 27 28	USE MPI INTEGER m, map(m), comm, p, win, ierr, disp_int REAL A(m), B(m)
26 27 28 29	USE MPI INTEGER m, map(m), comm, p, win, ierr, disp_int
26 27 28 29 30	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
26 27 28 29 30 31	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)
26 27 28 29 30	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
26 27 28 29 30 31 32	<pre>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</pre>
26 27 28 29 30 31 32 33	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34	<pre>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</pre>
26 27 28 29 30 31 32 33 34 35	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36 37	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	<pre>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, &</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>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, &</pre>

This code is identical to the code in Example 11.2, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, the code computes $B = A(map^{-1})$, which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 11.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 11.7 for details). The predefined operation MPI_REPLACE provides fetch-and-set behavior.

			10
MPI GE	T ACCUMULATE(origin a	addr, origin_count, origin_datatype, result_addr,	16
	(•	t_datatype, target_rank, target_disp, target_count,	17
	target_datatype, o		18
IN	origin_addr	initial address of buffer (choice)	19 20
	-		20 21
IN	origin_count	number of entries in origin buffer (non-negative inte-	21
		ger)	23
IN	origin_datatype	datatype of each entry in origin buffer (handle)	24
OUT	result_addr	initial address of result buffer (choice)	25
IN	result_count	number of entries in result buffer (non-negative inte-	26
		$\operatorname{ger})$	27
IN	result_datatype	datatype of each entry in result buffer (handle)	28
IN	target_rank	rank of target (non-negative integer)	29 30
IN	target_disp	displacement from start of window to beginning of tar-	31
	50.800-0.0P	get buffer (non-negative integer)	32
IN	target_count	number of entries in target buffer (non-negative inte-	33
IIN III		ger)	34
IN	target_datatype	datatype of each entry in target buffer (handle)	35
	0 11	·- · · · · · · · · · · · · · · · · · ·	36
IN	ор	reduce operation (handle)	37
IN	win	window object (handle)	38 39
			40
C bind	ing		41
	/		

<pre>int MPI_Get_accumulate(const void *origin_addr, int origin_count,</pre>	42
MPI_Datatype origin_datatype, void *result_addr,	43
<pre>int result_count, MPI_Datatype result_datatype,</pre>	44
<pre>int target_rank, MPI_Aint target_disp, int target_count,</pre>	45
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	46

F08 binding

```
1
     MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
\mathbf{2}
                    result_count, result_datatype, target_rank, target_disp,
3
                    target_count, target_datatype, op, win, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
5
6
         INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
7
                    target_count
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
8
9
                    result_datatype
10
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
11
         TYPE(MPI_Op), INTENT(IN) :: op
12
         TYPE(MPI_Win), INTENT(IN) :: win
13
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
14
     F binding
15
     MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
16
                    RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
17
                    TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
18
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
19
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
20
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
21
                    TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
22
23
         Accumulate origin_count elements of type origin_datatype from the origin buffer (
^{24}
     origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank
25
     and win, using the operation op and return in the result buffer result_addr the content
26
     of the target buffer before the accumulation, specified by target_disp, target_count, and
27
     target_datatype. The data transferred from origin to target must fit, without truncation,
28
     in the target buffer. Likewise, the data copied from target to origin must fit, without
29
     truncation, in the result buffer.
30
         The origin and result buffers (origin_addr and result_addr) must be disjoint. Each
^{31}
```

datatype argument must be a predefined datatype or a derived datatype where all basic components are of the same predefined datatype. All datatype arguments must be constructed from the same predefined datatype. The operation op applies to elements of that predefined type. target_datatype must not specify overlapping entries, and the target buffer must fit in the target window or in attached memory in a dynamic window. The operation is executed atomically for each basic datatype; see Section 11.7 for details.

37 Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or 38 MPI_REPLACE can be specified as op. User-defined functions cannot be used. A new 39 predefined operation, MPI_NO_OP, is defined. It corresponds to the associative function 40f(a,b) = a; i.e., the current value in the target memory is returned in the result buffer at 41 the origin and no operation is performed on the target buffer. When MPI_NO_OP is specified 42as the operation, the origin_addr, origin_count, and origin_datatype arguments are ignored. 43MPI_NO_OP can be used only in MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, 44and MPI_FETCH_AND_OP. MPI_NO_OP cannot be used in MPI_ACCUMULATE, 45MPI_RACCUMULATE, or collective reduction operations, such as MPI_REDUCE and others. 46

Advice to users. MPI_GET is similar to MPI_GET_ACCUMULATE, with the opera tion MPI_NO_OP. Note, however, that MPI_GET and MPI_GET_ACCUMULATE have

different constraints on concurrent updates. (End of advice to users.)

Fetch and Op Function

The generic functionality of MPI_GET_ACCUMULATE might limit the performance of fetchand-increment or fetch-and-add calls that might be supported by special hardware operations. MPI_FETCH_AND_OP thus allows for a fast implementation of a commonly used subset of the functionality of MPI_GET_ACCUMULATE.

MPI_FETCH_AND_OP(origin_addr, result_addr, datatype, target_rank, target_disp, op, win) IN origin_addr initial address of buffer (choice) result_addr OUT initial address of result buffer (choice) datatype datatype of the entry in origin, result, and target buf-IN fers (handle) rank of target (non-negative integer) IN target_rank IN target_disp displacement from start of window to beginning of target buffer (non-negative integer) IN reduce operation (handle) ор IN win window object (handle)

C binding int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,

MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Op op, MPI_Win win) F08 binding

MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank, 30 target_disp, op, win, ierror) 31TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 32 33 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr TYPE(MPI_Datatype), INTENT(IN) :: datatype 34 INTEGER, INTENT(IN) :: target_rank 35 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 36 TYPE(MPI_Op), INTENT(IN) :: op 37 TYPE(MPI_Win), INTENT(IN) :: win 38 INTEGER, OPTIONAL, INTENT(OUT) :: 39 ierror 40 41

F binding

MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK, 42TARGET_DISP, OP, WIN, IERROR) 43 <type> ORIGIN_ADDR(*), RESULT_ADDR(*) 44INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 45INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR 46

47Accumulate one element of type datatype from the origin buffer (origin_addr) to the 48 buffer at offset target_disp, in the target window specified by target_rank and win, using

Unofficial Draft for Comment Only

1

 $\mathbf{2}$ 3

4

5

6

7

8 9 10

11 12

13

1415

16

17

18

19

20

21

22

2324

25

26

27

28

1 the operation op and return in the result buffer result_addr the content of the target buffer $\mathbf{2}$ before the accumulation. 3 The origin and result buffers (origin_addr and result_addr) must be disjoint. Any of the 4 predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE, can be 5specified as op; user-defined functions cannot be used. The datatype argument must be a 6 predefined datatype. The operation is executed atomically. 7 8 Compare and Swap Function 9 Another useful operation is an atomic compare and swap where the value at the origin is 10 compared to the value at the target, which is atomically replaced by a third value only if 11 the values at origin and target are equal. 1213 14MPI_COMPARE_AND_SWAP(origin_addr, compare_addr, result_addr, datatype, target_rank, 15target_disp, win) 16origin_addr IN initial address of buffer (choice) 1718 IN compare_addr initial address of compare buffer (choice) 19OUT result_addr initial address of result buffer (choice) 20datatype of the element in all buffers (handle) IN datatype 2122IN target_rank rank of target (non-negative integer) 23IN target_disp displacement from start of window to beginning of tar-24get buffer (non-negative integer) 25IN win window object (handle) 2627C binding 28int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr, 29void *result_addr, MPI_Datatype datatype, int target_rank, 30 MPI_Aint target_disp, MPI_Win win) 31 32 F08 binding 33 MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, 34 target_rank, target_disp, win, ierror) 35 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 36 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: compare_addr 37 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr 38 TYPE(MPI_Datatype), INTENT(IN) :: datatype 39 INTEGER, INTENT(IN) :: target_rank 40 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 41 TYPE(MPI_Win), INTENT(IN) :: win 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 F binding 44MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE, 4546 TARGET_RANK, TARGET_DISP, WIN, IERROR) 47<type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*) INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP 48

Unofficial Draft for Comment Only

(origin_addr and result_addr) must be disjoint.

INTEGER DATATYPE, TARGET_RANK, WIN, IERROR

The end of the epoch, or explicit bulk synchronization using

Request-based RMA Communication Operations 11.3.5 Request-based RMA communication operations allow the user to associate a request handle with the RMA operations and test or wait for the completion of these requests using the

This function compares one element of type datatype in the compare buffer

compare_addr with the buffer at offset target_disp in the target window specified by

target_rank and win and replaces the value at the target with the value in the origin buffer

origin_addr if the compare buffer and the target buffer are identical. The original value at

the target is returned in the buffer result_addr. The parameter datatype must belong to

one of the following categories of predefined datatypes: C integer, Fortran integer, Logical,

Multi-language types, or Byte as specified in Section 5.9.2. The origin and result buffers

functions described in Section 3.7.3. Request-based RMA operations are only valid within

a passive target epoch (see Section 11.5). Upon returning from a completion call in which an RMA operation completes, the MPI_ERROR field in the associated status object is set appropriately (see Section 3.2.5). All

other fields of status and the results of status query functions (e.g., MPI_GET_COUNT) are undefined. It is valid to mix different request types (e.g., any combination of RMA requests, collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or MPI_CANCEL for a request associated with an RMA operation. RMA requests are not persistent.

MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, or MPI_WIN_FLUSH_LOCAL_ALL, also indicates completion of the RMA operations. However, users must still wait or test on the request handle to allow the MPI implementation to clean up any resources associated with these requests; in such cases the wait operation will complete locally.

1

 $\mathbf{2}$

3

4

5

6

7

8

9

10 11

1213

14

15

16

17

18

19

20

21

22

23

 24

25

26

27

28

29

30

1 2	MPI_RPU	T(origin_addr, origin_count, or target_datatype, win, re	igin_datatype, target_rank, target_disp, target_count, quest)
$\frac{3}{4}$	IN	origin_addr	initial address of origin buffer (choice)
5	IN	origin_count	number of entries in origin buffer (non-negative integer)
7	IN	origin_datatype	datatype of each entry in origin buffer (handle)
8 9	IN	target_rank	rank of target (non-negative integer)
9 10 11	IN	target_disp	displacement from start of window to target buffer (non-negative integer)
12 13	IN	target_count	number of entries in target buffer (non-negative integer)
14 15	IN	target_datatype	datatype of each entry in target buffer (handle)
16	IN	win	window object used for communication (handle)
17	OUT	request	RMA request (handle)
18			
19 20 21 22 23 24 25	C bindin int MPI_F	Rput(const void *origin_a MPI_Datatype origin MPI_Aint target_dis	_datatype, int target_rank, p, int target_count, _datatype, MPI_Win win,
26	F08 bind	0	
27	MPI_Rput		nt, origin_datatype, target_rank,
28 29		ierror)	_count, target_datatype, win, request,
30	TYPE		NT(IN), ASYNCHRONOUS :: origin_addr
31	INTEGER, INTENT(IN) :: origin_count, target_rank, target_count		
32		• -	N) :: origin_datatype, target_datatype D), INTENT(IN) :: target_disp
33 34		(MPI_Win), INTENT(IN) ::	0 1
35		(MPI_Request), INTENT(OUT	
36	INTEC	GER, OPTIONAL, INTENT(OUT	I) :: ierror
37	F binding	g	
38 39	MPI_RPUT	-	NT, ORIGIN_DATATYPE, TARGET_RANK,
40		TARGET_DISP, TARGET_ IERROR)	_COUNT, TARGET_DATATYPE, WIN, REQUEST,
41	<tvpe< th=""><th>e> ORIGIN_ADDR(*)</th><th></th></tvpe<>	e> ORIGIN_ADDR(*)	
42	• 1	GER(KIND=MPI_ADDRESS_KIND)) TARGET_DISP
43 44	INTEC		DATATYPE, TARGET_RANK, TARGET_COUNT,
45		TARGET_DATATYPE, WI	IN, REQUEST, IERROR
46			(Section 11.3.1), except that it allocates a commu-
47	nication re	equest object and associates	it with the request handle (the argument request).
48			

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)	10
IN	origin_count	number of entries in origin buffer (non-negative integer)	11 12
IN	origin_datatype	datatype of each entry in origin buffer (handle)	13 14
IN	target_rank	rank of target (non-negative integer)	15
IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)	16 17
IN	target_count	number of entries in target buffer (non-negative inte- ger)	18 19 20
IN	target_datatype	datatype of each entry in target buffer (handle)	21
IN	win	window object used for communication (handle)	22
OUT	request	RMA request (handle)	23 24

C binding

<pre>int MPI_Rget(void *origin_addr, int origin_count,</pre>		
MPI_Datatype origin_datatype, int target_rank,	28	
MPI_Aint target_disp, int target_count,	29	
MPI_Datatype target_datatype, MPI_Win win,	30	
MPI_Request *request)	31	
E08 binding	32	
F08 binding	33	
<pre>MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,</pre>	34	

F binding
44
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
45
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
46
IERROR)
47
<type> ORIGIN_ADDR(*)
48

```
1
          INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
\mathbf{2}
          INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
3
                     TARGET_DATATYPE, WIN, REQUEST, IERROR
4
          MPI_RGET is similar to MPI_GET (Section 11.3.2), except that it allocates a commu-
5
     nication request object and associates it with the request handle (the argument request)
6
     that can be used to wait or test for completion. The completion of an MPI_RGET operation
7
     indicates that the data is available in the origin buffer. If origin_addr points to memory
8
     attached to a window, then the data becomes available in the private copy of this window.
9
10
11
     MPI_RACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
12
                     target_count, target_datatype, op, win, request)
13
       IN
                 origin_addr
                                              initial address of buffer (choice)
14
       IN
                 origin_count
                                              number of entries in buffer (non-negative integer)
15
16
       IN
                 origin_datatype
                                              datatype of each entry in origin buffer (handle)
17
       IN
                 target_rank
                                              rank of target (non-negative integer)
18
19
       IN
                 target_disp
                                              displacement from start of window to beginning of tar-
                                              get buffer (non-negative integer)
20
21
       IN
                                              number of entries in target buffer (non-negative inte-
                 target_count
22
                                              ger)
23
       IN
                 target_datatype
                                              datatype of each entry in target buffer (handle)
24
       IN
                                              reduce operation (handle)
25
                 ор
26
                                              window object (handle)
       IN
                 win
27
       OUT
                 request
                                              RMA request (handle)
28
29
     C binding
30
     int MPI_Raccumulate(const void *origin_addr, int origin_count,
^{31}
                     MPI_Datatype origin_datatype, int target_rank,
32
                     MPI_Aint target_disp, int target_count,
33
                     MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
34
                     MPI_Request *request)
35
36
     F08 binding
37
     MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
38
                     target_disp, target_count, target_datatype, op, win, request,
39
                     ierror)
40
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
41
          INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
42
          TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
43
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
44
          TYPE(MPI_Op), INTENT(IN) :: op
45
          TYPE(MPI_Win), INTENT(IN) :: win
46
          TYPE(MPI_Request), INTENT(OUT) ::
                                                  request
47
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
48
```

F binding				
MPI_RACC		IGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	2	
	TARGET_DISP, TARGE IERROR)	T_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	3 4	
<t td="" vn<=""><td>e> ORIGIN_ADDR(*)</td><td></td><td>5</td></t>	e> ORIGIN_ADDR(*)		5	
• 1	GER(KIND=MPI_ADDRESS_KIN	ND) TARGET DISP	6	
		N_DATATYPE, TARGET_RANK, TARGET_COUNT,	7	
	TARGET_DATATYPE, (DP, WIN, REQUEST, IERROR	8	
MPI	RACCUMULATE is similar t	to MPI_ACCUMULATE (Section 11.3.4), except that	9	
		object and associates it with the request handle (the	10 11	
argument	request) that can be used t	o wait or test for completion. The completion of an	12	
	-	eates that the origin buffer is free to be updated. It	13	
does not i	indicate that the operation h	has completed at the target window.	14	
			15	
MPI_RGE	T_ACCUMULATE(origin_add	dr, origin_count, origin_datatype, result_addr,	16	
		tatype, target_rank, target_disp, target_count,	17 18	
	target_datatype, op, w	in, request)	18	
IN	origin_addr	initial address of buffer (choice)	20	
IN	origin_count	number of entries in origin buffer (non-negative inte-	21	
		ger)	22	
IN	origin_datatype	datatype of each entry in origin buffer (handle)	23 24	
OUT	result_addr	initial address of result buffer (choice)	24 25	
IN	result_count	number of entries in result buffer (non-negative inte-	26	
		ger)	27	
IN	result_datatype	datatype of each entry in result buffer (handle)	28 29	
IN	target_rank	rank of target (non-negative integer)	30	
IN	target_disp	displacement from start of window to beginning of tar-	31	
		get buffer (non-negative integer)	32	
IN	target_count	number of entries in target buffer (non-negative inte-	33	
		ger)	$\frac{34}{35}$	
IN	target_datatype	datatype of each entry in target buffer (handle)	36	
IN	ор	reduce operation (handle)	37	
IN	win	window object (handle)	38	
OUT	request	RMA request (handle)	$\frac{39}{40}$	
	,	· · · /	40 41	
C bindir	ıg		42	
int MPI_Rget_accumulate(const void *origin_addr, int origin_count, 43				
	MPI_Datatype origin_datatype, void *result_addr, 44			

MPI_Datatype origin_datatype, void *result_addr, Int origin_count, 43
MPI_Datatype origin_datatype, void *result_addr, 44
int result_count, MPI_Datatype result_datatype, 45
int target_rank, MPI_Aint target_disp, int target_count, 46
MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, 47
MPI_Request *request) 48

Unofficial Draft for Comment Only

```
1
     F08 binding
\mathbf{2}
     MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
3
                  result_addr, result_count, result_datatype, target_rank,
4
                  target_disp, target_count, target_datatype, op, win, request,
5
                   ierror)
6
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
8
         INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
9
                   target_count
10
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
11
                   result_datatype
12
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
13
         TYPE(MPI_Op), INTENT(IN) :: op
14
         TYPE(MPI_Win), INTENT(IN) :: win
15
         TYPE(MPI_Request), INTENT(OUT) ::
                                             request
16
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                              ierror
17
     F binding
18
     MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
19
                  RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
20
                  TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
21
                   IERROR)
22
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
23
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
24
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
25
                   TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
26
         IERROR
27
28
         MPI_RGET_ACCUMULATE is similar to MPI_GET_ACCUMULATE (Section 11.3.4),
```

²⁹ except that it allocates a communication request object and associates it with the request ³⁰ handle (the argument request) that can be used to wait or test for completion. The com-³¹ pletion of an MPI_RGET_ACCUMULATE operation indicates that the data is available in ³² the result buffer and the origin buffer is free to be updated. It does not indicate that the ³³ operation has been completed at the target window.

11.4 Memory Model

35

36

37 The memory semantics of RMA are best understood by using the concept of *public* and 38 private window copies. We assume that systems have a public memory region that is 39 addressable by all processes (e.g., the shared memory in shared memory machines or the 40 exposed main memory in distributed memory machines). In addition, most machines have 41 fast private buffers (e.g., transparent caches or explicit communication buffers) local to 42each process where copies of data elements from the main memory can be stored for faster 43 access. Such buffers are either coherent, i.e., all updates to main memory are reflected in 44 all private copies consistently, or non-coherent, i.e., conflicting accesses to main memory 45need to be synchronized and updated in all private copies explicitly. Coherent systems 46 allow direct updates to remote memory without any participation of the remote side. Non-47 coherent systems, however, need to call RMA functions in order to reflect updates to the 48

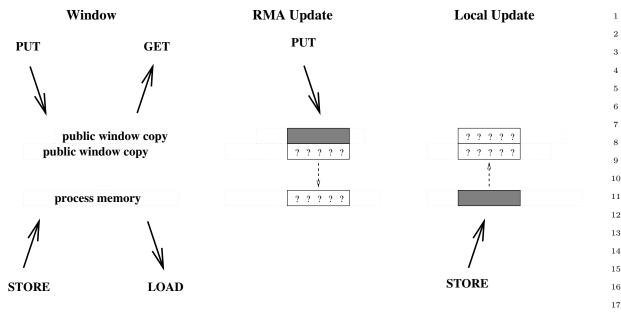


Figure 11.1: Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows.

public window in their private memory. Thus, in coherent memory, the public and the private window are identical while they remain logically separate in the non-coherent case. MPI thus differentiates between two **memory models** called **RMA unified**, if public and private window are logically identical, and **RMA separate**, otherwise.

In the RMA separate model, there is only one instance of each variable in process memory, but a distinct *public* copy of the variable for each window that contains it. A load accesses the instance in process memory (this includes MPI sends). A local store accesses and updates the instance in process memory (this includes MPI receives), but the update may affect other public copies of the same locations. A get on a window accesses the public copy of that window. A put or accumulate on a window accesses and updates the public copy of that window, but the update may affect the private copy of the same locations in process memory, and public copies of other overlapping windows. This is illustrated in Figure 11.1.

In the RMA unified model, public and private copies are identical and updates via put or accumulate calls are eventually observed by load operations without additional RMA calls. A store access to a window is eventually visible to remote get or accumulate calls without additional RMA calls. These stronger semantics of the RMA unified model allow the user to omit some synchronization calls and potentially improve performance.

Advice to users. If accesses in the RMA unified model are not synchronized (with locks or flushes, see Section 11.5.3), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (*End of advice to users.*)

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.

Unofficial Draft for Comment Only

18

19 20

21

22

23

 24

25

26

27

28

29

30

 31

32

33

34

35

36

37

38 39

40

41

42

43

44

 $45 \\ 46$

47

11.5 Synchronization Calls

RMA communications fall in two categories:

• active target communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.

• **passive target communication**, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI_PUT, MPI_GET or MPI_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations 27only within an **exposure epoch**. Such an epoch is started and completed by RMA syn-28chronization calls executed by the target process. Distinct exposure epochs at a process on 29 the same window must be disjoint, but such an exposure epoch may overlap with exposure 30 epochs on other windows or with access epochs for the same or other win arguments. There 31 is a one-to-one matching between access epochs at origin processes and exposure epochs 32 on target processes: RMA operations issued by an origin process for a target window will 33 access that target window during the same exposure epoch if and only if they were issued 34 during the same access epoch. 35

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

38 39

40

41

42

43

44

45

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

Unofficial Draft for Comment Only

1 2 3

4

5

6

7

8

9 10

11

12

13

14

15

16

during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.

2. The four functions MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST, and MPI_WIN_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly RMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.

These calls are used for active target communication. An access epoch is started at the origin process by a call to MPI_WIN_START and is terminated by a call to MPI_WIN_COMPLETE. The start call has a group argument that specifies the group of target processes for that epoch. An exposure epoch is started at the target process by a call to MPI_WIN_POST and is completed by a call to MPI_WIN_WAIT. The post call has a group argument that specifies the set of origin processes for that epoch.

3. Finally, shared lock access is provided by the functions MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, MPI_WIN_UNLOCK, and MPI_WIN_UNLOCK_ALL. MPI_WIN_LOCK and MPI_WIN_UNLOCK also provide exclusive lock capability. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a "billboard" model, where processes can, at random times, access or update different parts of the billboard.

These four calls provide passive target communication. An access epoch is started by a call to MPI_WIN_LOCK or MPI_WIN_LOCK_ALL and terminated by a call to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL, respectively.

Figure 11.2 illustrates the general synchronization pattern for active target communication. The synchronization between **post** and **start** ensures that the put call of the origin process does not start until the target process exposes the window (with the **post** call); the target process will expose the window only after preceding local accesses to the window have completed. The synchronization between **complete** and **wait** ensures that the put call of the origin process completes before the window is unexposed (with the **wait** call). The target process will execute following local accesses to the target window only after the **wait** returned.

Figure 11.2 shows operations occurring in the natural temporal order implied by the 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

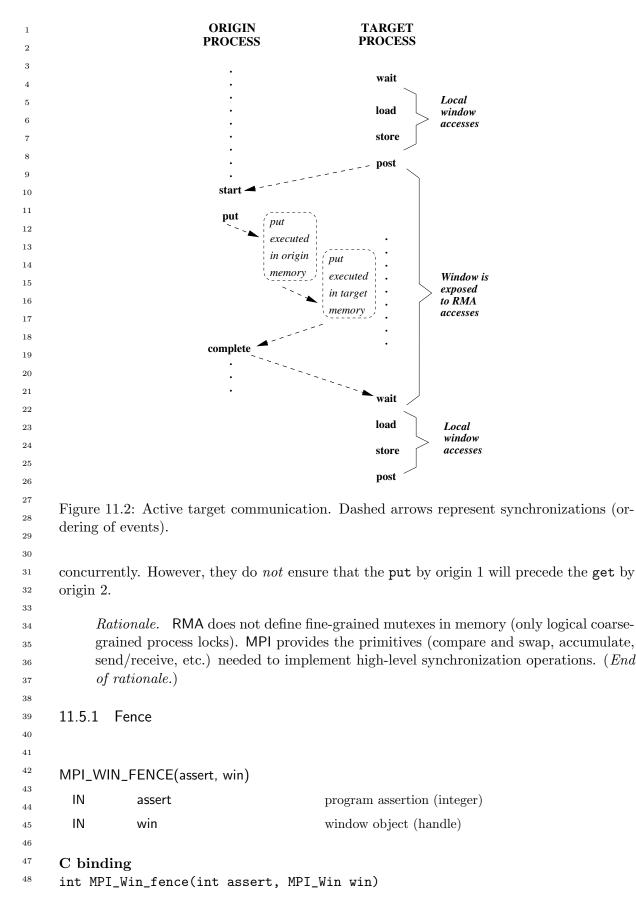
Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

 24

 31



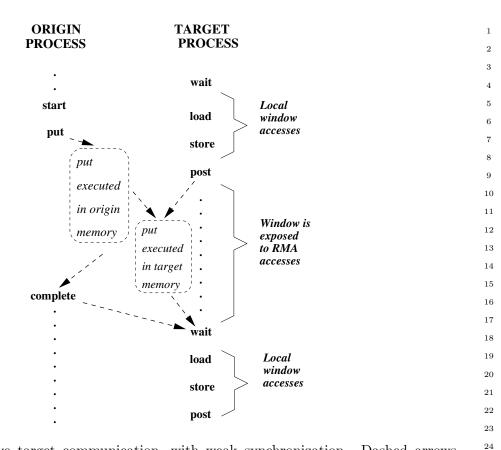


Figure 11.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

F08 binding

```
MPI_Win_fence(assert, win, ierror)
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

```
MPI_WIN_FENCE(ASSERT, WIN, IERROR)
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 48

Unofficial Draft for Comment Only

25

26 27

28

29

30

31

32 33

34

35

36

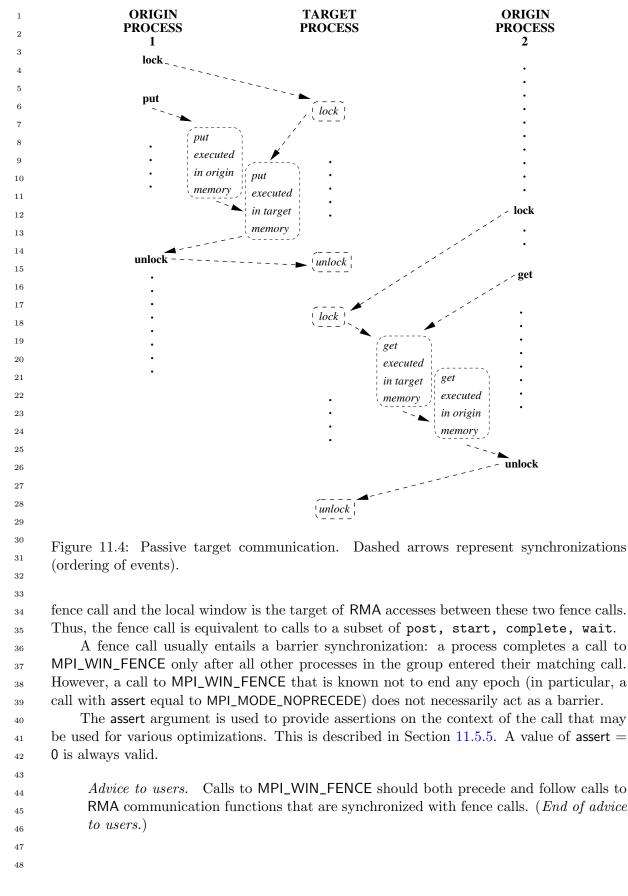
37

38

39

40

41



11.5.2 General Active Target Synchronization

MPI_WIN_START(group, assert, win)

IN	group	group of target processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)

C binding

int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)

F08 binding

```
MPI_Win_start(group, assert, win, ierror)
    TYPE(MPI_Group), INTENT(IN) ::
                                    group
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                       ierror
```

F binding

MPI_WIN_START(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR

Starts an RMA access epoch for win. RMA calls issued on win during this epoch must access only windows at processes in group. Each process in group must issue a matching call to MPI_WIN_POST. RMA accesses to each target window will be delayed, if necessary, until the target process executed the matching call to MPI_WIN_POST. MPI_WIN_START is allowed to block until the corresponding MPI_WIN_POST calls are executed, but is not required to.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.5.5. A value of assert =0 is always valid.

MPI_WIN_COMPLETE(win) win

```
IN
```

window object (handle)

C binding

int MPI_Win_complete(MPI_Win win)

F08 binding

MPI_Win_complete(win, ierror) TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

```
MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR
```

9 10

11

12

13

1415

1617

18

19

20

21

22 23

 24

25

26

2728

29

30

31

32 33

34 35

36 37

38

39

40

41

42

43 44

45

46

1 Completes an RMA access epoch on win started by a call to MPI_WIN_START. All $\mathbf{2}$ RMA communication calls issued on win during this epoch will have completed at the origin 3 when the call returns. 4 MPI_WIN_COMPLETE enforces completion of preceding RMA calls at the origin, but $\mathbf{5}$ not at the target. A put or accumulate call may not have completed at the target when it 6 has completed at the origin. $\overline{7}$ Consider the sequence of calls in the example below. 8 Example 11.4 9 10 MPI_Win_start(group, flag, win); 11 MPI_Put(..., win); 12MPI_Win_complete(win); 13 14The call to MPI_WIN_COMPLETE does not return until the put call has completed 15at the origin; and the target window will be accessed by the put operation only after the 16call to MPI_WIN_START has matched a call to MPI_WIN_POST by the target process. 17This still leaves much choice to implementors. The call to MPI_WIN_START can block 18until the matching call to MPI_WIN_POST occurs at all target processes. One can also 19have implementations where the call to MPI_WIN_START is nonblocking, but the call to 20MPI_PUT blocks until the matching call to MPI_WIN_POST occurs; or implementations 21where the first two calls are nonblocking, but the call to MPI_WIN_COMPLETE blocks 22until the call to MPI_WIN_POST occurred; or even implementations where all three calls 23can complete before any target process has called MPI_WIN_POST — the data put must 24 be buffered, in this last case, so as to allow the put to complete at the origin ahead of its 25completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence 26above must complete, without further dependencies. 2728MPI_WIN_POST(group, assert, win) 2930 IN group of origin processes (handle) group 31 IN assert program assertion (integer) 32 IN win window object (handle) 33 3435 C binding 36 int MPI_Win_post(MPI_Group group, int assert, MPI_Win win) 37 F08 binding 38 MPI_Win_post(group, assert, win, ierror) 39 TYPE(MPI_Group), INTENT(IN) :: group 40 INTEGER, INTENT(IN) :: assert 41 TYPE(MPI_Win), INTENT(IN) :: win 42INTEGER, OPTIONAL, INTENT(OUT) :: ierror 43 44F binding

```
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
INTEGER GROUP, ASSERT, WIN, IERROR
```

45

46

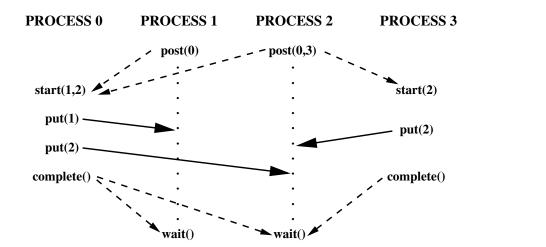


Figure 11.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

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.

MPI_WIN_WAIT(win)

IN win window object (handle)

C binding

int MPI_Win_wait(MPI_Win win)

F08 binding

```
MPI_Win_wait(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

MPI_WIN_WAIT(WIN, IERROR) INTEGER WIN, IERROR

Completes an RMA exposure epoch started by a call to MPI_WIN_POST on win. This call matches calls to MPI_WIN_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. The call to MPI_WIN_WAIT will block until all matching calls to MPI_WIN_COMPLETE have occurred. This guarantees that all these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

Figure 11.5 illustrates the use of these four functions. Process 0 puts data in the 43 windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each 44 start call lists the ranks of the processes whose windows will be accessed; each post call lists 45 the ranks of the processes that access the local window. The figure illustrates a possible 46 timing for the events, assuming strong synchronization; in a weak synchronization, the start, 47 put or complete calls may occur ahead of the matching post calls. 48

Unofficial Draft for Comment Only

```
1
     MPI_WIN_TEST(win, flag)
2
       IN
                                              window object (handle)
                 win
3
       OUT
                 flag
                                              success flag (logical)
4
5
6
     C binding
     int MPI_Win_test(MPI_Win win, int *flag)
7
8
     F08 binding
9
     MPI_Win_test(win, flag, ierror)
10
          TYPE(MPI_Win), INTENT(IN) :: win
11
          LOGICAL, INTENT(OUT) :: flag
12
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
13
14
     F binding
15
     MPI_WIN_TEST(WIN, FLAG, IERROR)
16
          INTEGER WIN, IERROR
17
          LOGICAL FLAG
18
          This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if all accesses
19
     to the local window by the group to which it was exposed by the corresponding
20
     MPI_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE
21
     calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned
22
     immediately. The effect of return of MPI_WIN_TEST with flag = true is the same as the
23
     effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible
24
     effect.
25
          MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once
26
     the call has returned flag = true, it must not be invoked anew, until the window is posted
27
     anew.
28
          Assume that window win is associated with a "hidden" communicator wincomm, used
29
     for communication by the processes of win. The rules for matching of post and start calls
30
     and for matching complete and wait calls can be derived from the rules for matching sends
^{31}
     and receives, by considering the following (partial) model implementation.
32
33
     MPI_WIN_POST(group,0,win) initiates a nonblocking send with tag tag0 to each process
34
           in group, using wincomm. There is no need to wait for the completion of these sends.
35
36
     MPI_WIN_START(group,0,win) initiates a nonblocking receive with tag tag0 from each
37
           process in group, using wincomm. An RMA access to a window in target process i is
38
           delayed until the receive from i is completed.
39
     MPI_WIN_COMPLETE(win) initiates a nonblocking send with tag tag1 to each process
40
           in the group of the preceding start call. No need to wait for the completion of these
41
           sends.
42
43
     MPI_WIN_WAIT(win) initiates a nonblocking receive with tag tag1 from each process in
44
           the group of the preceding post call. Wait for the completion of all receives.
45
46
          No races can occur in a correct program: each of the sends matches a unique receive.
47
     and vice versa.
48
```

Rationale. The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph $G = \langle V, E \rangle$, where $V = \{0, \ldots, n-1\}$ and $ij \in E$ if origin process i accesses the window at target process j. Then each process i issues a call to MPI_WIN_POST($ingroup_i, \ldots$), followed by a call to MPI_WIN_START($outgroup_i, \ldots$), where $outgroup_i = \{j : ij \in E\}$ and $ingroup_i = \{i \in E\}$. A call is a poop and can be clipped if the group argument is empty.

 $\{j : ji \in E\}$. 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 MPI_LOCK_SHARED (state)
IN	rank	rank of locked window (non-negative integer)
IN	assert	program assertion (integer)
IN	win	window object (handle)

C binding

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

```
F08 binding
```

```
MPI_Win_lock(lock_type, rank, assert, win, ierror)
    INTEGER, INTENT(IN) :: lock_type, rank, assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR) INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR

Starts an RMA access epoch. The window at the process with rank rank can be accessed by RMA operations on win during that epoch. Multiple RMA access epochs (with calls

Unofficial Draft for Comment Only

```
1
     to MPI_WIN_LOCK) can occur simultaneously; however, each access epoch must target a
\mathbf{2}
     different process.
3
4
     MPI_WIN_LOCK_ALL(assert, win)
5
6
       IN
                 assert
                                             program assertion (integer)
7
       IN
                 win
                                             window object (handle)
8
9
     C binding
10
     int MPI_Win_lock_all(int assert, MPI_Win win)
11
12
     F08 binding
13
     MPI_Win_lock_all(assert, win, ierror)
14
          INTEGER, INTENT(IN) :: assert
15
          TYPE(MPI_Win), INTENT(IN) :: win
16
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
17
     F binding
18
     MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
19
          INTEGER ASSERT, WIN, IERROR
20
21
          Starts an RMA access epoch to all processes in win, with a lock type of
22
     MPI_LOCK_SHARED. During the epoch, the calling process can access the window memory on
23
     all processes in win by using RMA operations. A window locked with MPI_WIN_LOCK_ALL
24
     must be unlocked with MPI_WIN_UNLOCK_ALL. This routine is not collective — the ALL
25
     refers to a lock on all members of the group of the window.
26
27
                             There may be additional overheads associated with using
           Advice to users.
28
           MPI_WIN_LOCK and MPI_WIN_LOCK_ALL concurrently on the same window. These
29
           overheads could be avoided by specifying the assertion MPI_MODE_NOCHECK when
30
           possible (see Section 11.5.5). (End of advice to users.)
^{31}
32
33
     MPI_WIN_UNLOCK(rank, win)
34
35
       IN
                 rank
                                             rank of window (non-negative integer)
36
       IN
                 win
                                             window object (handle)
37
38
     C binding
39
     int MPI_Win_unlock(int rank, MPI_Win win)
40
41
     F08 binding
42
     MPI_Win_unlock(rank, win, ierror)
43
          INTEGER, INTENT(IN) :: rank
44
          TYPE(MPI_Win), INTENT(IN) :: win
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     F binding
47
     MPI_WIN_UNLOCK(RANK, WIN, IERROR)
48
```

INTEGER RANK, WIN, IERROR

Completes an RMA access epoch started by a call to MPI_WIN_LOCK on window win. RMA operations issued during this period will have completed both at the origin and at the target when the call returns.

MPI_WIN_UNLOCK_ALL(win)

IN win window object (handle)
C binding
int MPI_Win_unlock_all(MPI_Win win)
F08 binding
MPI_Win_unlock_all(win, ierror)
 TYPE(MPI_Win), INTENT(IN) :: win
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

MPI_WIN_UNLOCK_ALL(WIN, IERROR) INTEGER WIN, IERROR

Completes a shared RMA access epoch started by a call to MPI_WIN_LOCK_ALL on 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.)

Unofficial Draft for Comment Only

 31

1	Imp	lementors may rest	rict the use of RMA communication that is synchronized by	
2	lock call	s to windows in me	emory allocated by MPI_ALLOC_MEM (Section 8.2),	
3	MPI_WIN_ALLOCATE (Section 11.2.2), MPI_WIN_ALLOCATE_SHARED (Section 11.2.3),			
4	or attached with MPI_WIN_ATTACH (Section 11.2.4). Locks can be used portably only in			
5		such memory.		
6		U		
7	Ra	tionale. The im	plementation of passive target communication when memory	
8	is	not shared may red	uire an asynchronous software agent. Such an agent can be	
9	im	plemented more eas	ily, and can achieve better performance, if restricted to specially	
10	all	ocated memory. It	can be avoided altogether if shared memory is used. It seems	
11		÷	rictions that allows one to use shared memory for third party	
12		-	red memory machines.	
13				
14	(E	nd of rationale.)		
15	Con	sider the sequence	of calls in the example below.	
16	001	sider the sequence	si cans in the example below.	
10	Examp	le 11.5		
18				
			XCLUSIVE, rank, assert, win);	
19		(, rank,,	-	
20	MPI_Win	_unlock(rank, wi	n);	
21	The	call to MPL WIN I	JNLOCK will not return until the put transfer has completed at	
22			. This still leaves much freedom to implementors. The call to	
23	-	-	until an exclusive lock on the window is acquired; or, the first	
24		•	÷ , , ,	
25	two calls may not block, while MPI_WIN_UNLOCK blocks until a lock is acquired — the undete of the target window is then pertoand until the call to MPI_WIN_UNLOCK acquired			
26	update of the target window is then postponed until the call to MPI_WIN_UNLOCK occurs.			
27	However, if the call to MPI_WIN_LOCK is used to lock a local window, then the call must block until the lock is acquired, since the lock may protect local load/store accesses to the			
28		issued after the lock		
29	window	issued after the loci	. can returns.	
30				
31	11.5.4	Flush and Sync		
32	All flush	and sync functions	can be called only within passive target epochs.	
33		v		
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	C bindi	0		
41	int MPI_Win_flush(int rank, MPI_Win win)			
42	F08 bin	ding		
43		0	ierrer	
44		_flush(rank, win	-	
45		EGER, INTENT(IN)		
46		E(MPI_Win), INTE		
47	TNT	EGER, OPTIONAL,	INTENT(OUT) :: ierror	
48	F bindi	ng		

MPI_WIN_FLUSH(RANK, WIN, IERROR) INTEGER RANK, WIN, IERROR				
MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling process to the target rank on the specified window. The operations are completed both at the origin and at the target.				
MPI_WIN_FLUSH_ALL(win)				
IN	win	window object (handle)	9 10	
			11	
	inding MPI_Win_flush_all(MPI_Win	win)	12 13	
F08 binding				
	.Win_flush_all(win, ierror)	15	
_	TYPE(MPI_Win), INTENT(IN)		16	
	INTEGER, OPTIONAL, INTENT	(OUT) :: ierror	17 18	
\mathbf{F} bi	nding		19	
	WIN_FLUSH_ALL(WIN, IERROR)	20	
	INTEGER WIN, IERROR			
All RMA operations issued by the calling process to any target on the specified w prior to this call and in the specified window will have completed both at the origin			22 23 24	
the t	arget when this call returns.		25	
			26	
MPI_WIN_FLUSH_LOCAL(rank, win)			27	
IN	rank	rank of target window (non-negative integer)	28	
IN	win	window object (handle)	29 30	
	•••••		31	
Сb	inding		32	
int MPI_Win_flush_local(int rank, MPI_Win win)			33	
F08 binding MPI_Win_flush_local(rank, win, ierror)			34	
			35	
··· - <u>-</u>		ank	36	
	TYPE(MPI_Win), INTENT(IN)	:: win	37 38	
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
F bi	nding		40	
	PI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR) INTEGER RANK, WIN, IERROR			
-				
	Locally completes at the origin all outstanding RMA operations initiated by the calling rocess to the target process specified by rank on the specified window. For example, after			
			45	
	routine completes, the user ma	ay reuse any buffers provided to put, get, or accumulate	46	
	routine completes, the user ma ations.	ay reuse any bullers provided to put, get, or accumulate	46 47	

```
1
     MPI_WIN_FLUSH_LOCAL_ALL(win)
\mathbf{2}
       IN
                                             window object (handle)
                 win
3
4
     C binding
5
     int MPI_Win_flush_local_all(MPI_Win win)
6
7
     F08 binding
8
     MPI_Win_flush_local_all(win, ierror)
9
          TYPE(MPI_Win), INTENT(IN) :: win
10
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
11
     F binding
12
     MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
13
          INTEGER WIN, IERROR
14
15
          All RMA operations issued to any target prior to this call in this window will have
16
     completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.
17
18
     MPI_WIN_SYNC(win)
19
20
       IN
                                             window object (handle)
                 win
21
22
     C binding
23
     int MPI_Win_sync(MPI_Win win)
^{24}
25
     F08 binding
26
     MPI_Win_sync(win, ierror)
          TYPE(MPI_Win), INTENT(IN) :: win
27
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
29
     F binding
30
     MPI_WIN_SYNC(WIN, IERROR)
^{31}
          INTEGER WIN, IERROR
32
33
          The call MPI_WIN_SYNC synchronizes the private and public window copies of win.
34
     For the purposes of synchronizing the private and public window, MPI_WIN_SYNC has the
35
     effect of ending and reopening an access and exposure epoch on the window (note that it
     does not actually end an epoch or complete any pending MPI RMA operations).
36
37
38
     11.5.5 Assertions
39
     The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE,
40
     MPI_WIN_LOCK, and MPI_WIN_LOCK_ALL is used to provide assertions on the context of
41
     the call that may be used to optimize performance. The assert argument does not change
42
     program semantics if it provides correct information on the program — it is erroneous to
43
     provide incorrect information. Users may always provide assert = 0 to indicate a general
44
     case where no guarantees are made.
45
46
           Advice to users. Many implementations may not take advantage of the information
47
           in assert; some of the information is relevant only for noncoherent shared memory ma-
48
```

chines. Users should consult their implementation's manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations whenever available. (*End of advice to users.*)

Advice to implementors. Implementations can always ignore the assert argument. Implementors should document which assert values are significant on their implementation. (*End of advice to implementors.*)

assert is the bit-vector OR of zero or more of the following integer constants: MPI_MODE_NOCHECK, MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE, and MPI_MODE_NOSUCCEED. The significant options are listed below for each call.

Advice to users. C/C++ users can use bit vector or (|) to combine these constants; Fortran 90 users can use the bit-vector IOR intrinsic. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (*End of advice to users.*)

MPI_WIN_START:

MPI_MODE_NOCHECK — the matching calls to MPI_WIN_POST have already completed on all target processes when the call to MPI_WIN_START is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit in the code. (However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.)

MPI_WIN_POST:

- MPI_MODE_NOCHECK the matching calls to MPI_WIN_START have not yet occurred on any origin processes when the call to MPI_WIN_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
- 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.

1 2

3

4

5

6

 $\overline{7}$

8 9

10

11

12

13 14

15

16

17

18 19

20 21

22

23

 24

25

26

27

28

29 30

 31

32

33

34

35

36

37 38

39

40

41

42 43

44

45

46

47 48

499

Unofficial Draft for Comment Only

1 2 3	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.
4 5 6 7	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.
8	MPI_WIN_LOCK, MPI_WIN_LOCK_ALL:
9 10 11 12 13	MPI_MODE_NOCHECK — no other process holds, or will attempt to acquire, a con- flicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.
14 15 16 17	Advice to users. Note that the nostore and noprecede flags provide information on what happened <i>before</i> the call; the noput and nosucceed flags provide information on what will happen <i>after</i> the call. (<i>End of advice to users.</i>)
18 19	11.5.6 Miscellaneous Clarifications
 Once an RMA routine completes, it is safe to free any opaque objects passed as an to that routine. For example, the datatype argument of a MPI_PUT call can be 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 communication. 	
26 27	11.6 Error Handling
28 29	11.6.1 Error Handlers
30 31 32 33 34 35 36	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).
37	11.6.2 Error Classes
 38 39 40 41 42 	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.
42 43	11.7 Semantics and Correctness

CHAPTER 11. ONE-SIDED COMMUNICATIONS

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

Unofficial Draft for Comment Only

MPI_ERR_WIN	invalid win argument	1
MPI_ERR_BASE	invalid base argument	2
MPI_ERR_SIZE	invalid size argument	3
MPI_ERR_DISP	invalid disp argument	4
MPI_ERR_LOCKTYPE	invalid locktype argument	5
MPI_ERR_ASSERT	invalid assert argument	6
MPI_ERR_RMA_CONFLICT	conflicting accesses to window	7
MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls	8
MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case	9
	of a window created with	10
	MPI_WIN_CREATE_DYNAMIC, target memory is not	11
	attached)	12
MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource	13
	exhaustion)	14
MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the	15
	group of the specified communicator cannot expose	16
	shared memory)	17
MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called	18
	function	19
		20
		21

Table 11.2: Error classes in one-sided communication routines

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

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

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.

10 The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public 11 copy to private copy (6) is the same call that completes the put or accumulate operation in 12the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then 13 the update of the public window copy is complete as soon as the updating process executed 14MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. In the RMA separate memory model, the 15update of a private copy in the process memory may be delayed until the target process 16executes a synchronization call on that window (6). Thus, updates to process memory can 17always be delayed in the RMA separate memory model until the process executes a suitable 18 synchronization call, while they must complete in the RMA unified model without additional 19synchronization calls. If fence or post-start-complete-wait synchronization is used, updates 20to a public window copy can be delayed in both memory models until the window owner 21executes a synchronization call. When passive target synchronization is used, it is necessary 22to update the public window copy even if the window owner does not execute any related 23synchronization call. 24

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI_WIN_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI_WIN_FENCE(0, win2) makes these updates visible in the public copy of win2.

The behavior of some MPI RMA operations may be *undefined* in certain situations. For 31 example, the result of several origin processes performing concurrent MPI_PUT operations 32 to the same target location is undefined. In addition, the result of a single origin process 33 performing multiple MPI_PUT operations to the same target location within the same 34access epoch is also undefined. The result at the target may have all of the data from one 35 of the MPI_PUT operations (the "last" one, in some sense), bytes from some of each of the 36 operations, or something else. In MPI-2, such operations were erroneous. That meant that 37 an MPI implementation was permitted to signal an MPI exception. Thus, user programs or 38 tools that used MPI RMA could not portably permit such operations, even if the application 39 code could function correctly with such an undefined result. In MPI-3, these operations are 40not erroneous, but do not have a defined behavior. 41

Rationale. As discussed in [6], requiring operations such as overlapping puts to
 be erroneous makes it difficult to use MPI RMA to implement programming models—
 such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further,
 while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any
 implementation that enforces this rule, as it would require significant overhead. Thus,
 relaxing this condition does not impact existing implementations or applications. (End
 of rationale.)

1

2

3

4

5

6

7

8

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.
- 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 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

43

44

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 16of a remote read (but not update) is valid (not erroneous) but the precise result 17 will depend on the behavior of the implementation. Store updates will appear in 18 memory, but there are no atomicity or ordering guarantees if more than one byte is 19 updated. Updates are stable in the sense that once data appears in memory, the data 20remains until replaced by another update. This permits updates to memory with 21store operations without requiring an RMA epoch. Users are cautioned that remote 22 accesses to a window that is updated by the local process has defined behavior only 23if the other rules given here and elsewhere in this chapter are followed. 24
 - 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.
 - U5. A put or accumulate must not access a target window once a store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, a store operation to a location in a window must not start once a put or accumulate update to the same location in that target window has started and until the put or accumulate update update completes at the target.
- 39 In the unified memory model, in the case where the window is Advice to users. 40 in shared memory, MPI_WIN_SYNC can be used to order store operations and make 41 store updates to the window visible to other processes and threads. Use of this 42routine is necessary to ensure portable behavior when point-to-point, collective, or shared memory synchronization is used in place of an RMA synchronization routine. 43 44 MPI_WIN_SYNC should be called by the writer before the non-RMA synchroniza-45tion operation and by the reader after the non-RMA synchronization, as shown in 46 Example 11.21. (End of advice to users.)
- ⁴⁸ A program that violates these rules has undefined behavior.

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10 11

12

13

14 15

25

26

27

28

29

30

31 32

33

34

35

36

37

38

Advice to users. A user can write correct programs by following the following rules:

- fence: During each period between fence calls, each window is either updated by put or accumulate calls, or updated by stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.
- **post-start-complete-wait:** A window should not be updated with store operations while posted if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window is posted.

With the post-start synchronization, the target process can tell the origin process that its window is now ready for RMA access; with the complete-wait synchronization, the origin process can tell the target process that it has finished its RMA accesses to the window.

- **lock:** Updates to the window are protected by exclusive locks if they may conflict. Nonconflicting accesses (such as read-only accesses or accumulate accesses) are protected by shared locks, both for load/store accesses and for RMA accesses.
- changing window or synchronization mode: One can change synchronization 22 mode, or change the window used to access a location that belongs to two over-23lapping 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.

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8 9

10

11 12

13

14

15

16

17

18

19

20

21

 24

25

26

27

28

2930

 31

32

33

34

35

36 37

38 39

40

41

42

```
1
     Process A:
                                    Process B:
\mathbf{2}
                                    window location X
3
4
                                    MPI_Win_lock(EXCLUSIVE, B)
                                    store X /* local update to private copy of B */
5
6
                                    MPI_Win_unlock(B)
7
                                    /* now visible in public window copy */
8
9
     MPI_Barrier
                                    MPI_Barrier
10
11
     MPI_Win_lock(EXCLUSIVE, B)
12
     MPI_Get(X) /* ok, read from public window */
13
     MPI_Win_unlock(B)
14
15
     Example 11.7 In the RMA unified model, although the public and private copies of the
16
     windows are synchronized, caution must be used when combining load/stores and multi-
17
     process synchronization. Although the following example appears correct, the compiler or
18
     hardware may delay the store to X after the barrier, possibly resulting in the MPI_GET
19
     returning an incorrect value of X.
20
21
     Process A:
                               Process B:
22
                               window location X
23
^{24}
                               store X /* update to private & public copy of B */
25
     MPI_Barrier
                               MPI_Barrier
26
     MPI_Win_lock_all
27
     MPI_Get(X) /* ok, read from window */
28
     MPI_Win_flush_local(B)
29
     /* read value in X */
30
     MPI_Win_unlock_all
31
32
     MPI_BARRIER provides process synchronization, but not memory synchronization. The
33
     example could potentially be made safe through the use of compiler- and hardware-specific
34
     notations to ensure the store to X occurs before process B enters the MPI_BARRIER. The
35
     use of one-sided synchronization calls, as shown in Example 11.6, also ensures the correct
36
     result.
37
38
     Example 11.8 The following example demonstrates the reading of a memory location
39
     updated by a remote process (Rule 6) in the RMA separate memory model. Although the
40
     MPI_WIN_UNLOCK on process A and the MPI_BARRIER ensure that the public copy on
41
     process B reflects the updated value of X, the call to MPI_WIN_LOCK by process B is
42
     necessary to synchronize the private copy with the public copy.
43
     Process A:
                                    Process B:
44
                                    window location X
45
46
     MPI_Win_lock(EXCLUSIVE, B)
47
     MPI_Put(X) /* update to public window */
48
```

MPI_Win_unlock(B)		1
		2
MPI_Barrier	MPI_Barrier	3
		4
	MPI_Win_lock(EXCLUSIVE, B)	5
	<pre>/* now visible in private copy of B */</pre>	6
	load X	7
	MPI_Win_unlock(B)	8
		q

Note that in this example, the barrier is not critical to the semantic correctness. The use of exclusive locks guarantees a remote process will not modify the public copy after MPI_WIN_LOCK synchronizes the private and public copies. A polling implementation looking for changes in X on process B would be semantically correct. The barrier is required to ensure that process A performs the put operation before process B performs the load of X.

Example 11.9 Similar to Example 11.7, the following example is unsafe even in the unified model, because the load of X can not be guaranteed to occur after the MPI_BARRIER. While Process B does not need to explicitly synchronize the public and private copies through MPI_WIN_LOCK as the MPI_PUT will update both the public and private copies of the window, the scheduling of the load could result in old values of X being returned. Compiler and hardware specific notations could ensure the load occurs after the data is updated, or explicit one-sided synchronization calls can be used to ensure the proper result.

Process A:	Process B: window location X
<pre>MPI_Win_lock_all MPI_Put(X) /* update to MPI_Win_flush(B)</pre>	o window */
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.

Process A:	Process B:	40
	window location X	41
		42
	store X /* update to private copy of B */	43
	MPI_Win_lock(SHARED, B)	44
MPI_Barrier	MPI_Barrier	45
		46
MPI_Win_lock(SHARED, B)		47
MPI_Get(X) /* X may be the	• X before the store */	48

Unofficial Draft for Comment Only

1 MPI_Win_unlock(B) $\mathbf{2}$ MPI_Win_unlock(B) 3 /* update on X now visible in public window */ 4 The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would 5guarantee process A would see the updated value of X, as the public copy of the window 6 would be explicitly synchronized with the private copy. 7 8 **Example 11.11** Similar to the previous example, Rule 5 can have unexpected implications 9 for general active target synchronization with the RMA separate memory model. It is not 10 guaranteed that process B reads the value of X as per the local update by process A, because 11 neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in 12the public window copy. 13 14Process A: Process B: 15window location X 16window location Y 1718 store Y 19 MPI_Win_post(A, B) /* Y visible in public window */ 20MPI_Win_start(A) MPI_Win_start(A) 2122store X /* update to private window */ 23 24 MPI_Win_complete MPI_Win_complete 25MPI_Win_wait 26/* update on X may not yet visible in public window */ 2728MPI_Barrier MPI_Barrier 29 30 MPI_Win_lock(EXCLUSIVE, A) 31 MPI_Get(X) /* may return an obsolete value */ 32 MPI_Get(Y) 33 MPI_Win_unlock(A) 34 35 To allow process B to read the value of X stored by A the local store must be replaced by 36 a local MPI_PUT that updates the public window copy. Note that by this replacement X 37 may become visible in the private copy of process A only after the MPI_WIN_WAIT call in 38 process A. The update to Y made before the MPI_WIN_POST call is visible in the public 39 window after the MPI_WIN_POST call and therefore process B will read the proper value 40of Y. The MPI_GET(Y) call could be moved to the epoch started by the MPI_WIN_START 41 operation, and process B would still get the value stored by process A. 42**Example 11.12** The following example demonstrates the interaction of general active 43 target synchronization with local read operations with the RMA separate memory model. 44Rules 5 and 6 do not guarantee that the private copy of X at process B has been updated 45before the load takes place. 4647

Process A:	Process B: window location X	
<pre>MPI_Win_lock(EXCLUSIVE, B) MPI_Put(X) /* update to pu MPI_Win_unlock(B)</pre>	blic window */	
MPI_Barrier	MPI_Barrier	
	MPI_Win_post(B) MPI_Win_start(B)	1
	<pre>load X /* access to private window */ /* may return an obsolete value */</pre>	:
	MPI_Win_complete MPI_Win_wait	:

To ensure that the value put by process A is read, the local load must be replaced with a local MPI_GET operation, or must be placed after the call to MPI_WIN_WAIT.

11.7.1 Atomicity

The outcome of concurrent accumulate operations to the same location with the same predefined datatype is as if the accumulates were done at that location in some serial order. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 11.2.1. Concurrent accumulate operations with different origin and target pairs are not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to an accumulate operation cannot be accessed by a load or an RMA call other than accumulate until the accumulate operation has completed (at the target). Different interleavings can lead to different results only to the extent that computer arithmetics are not truly associative or commutative. The outcome of accumulate operations with overlapping types of different sizes or target displacements is undefined.

11.7.2 Ordering

Accumulate calls enable element-wise atomic read and write to remote memory locations. 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.

The default strict ordering may incur a significant performance penalty. MPI specifies the info key accumulate_ordering to allow relaxation of the ordering semantics when specified 48

Unofficial Draft for Comment Only

 24

 $45 \\ 46$

1 to any window creation function. The values for this key are as follows. If set to none, $\mathbf{2}$ then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA 3 in MPI-2 but is not the default in MPI-3. The key can be set to a comma-separated list 4 of required access orderings at the target. Allowed values in the comma-separated list $\mathbf{5}$ are rar, war, raw, and waw for read-after-read, write-after-read, read-after-write, and write-6 after-write ordering, respectively. These indicate whether operations of the specified type $\overline{7}$ complete in the order they were issued. For example, raw means that any writes must 8 complete at the target before subsequent reads. These ordering requirements apply only to 9 operations issued by the same origin process and targeting the same target process. The 10 default value for accumulate_ordering is rar,raw,war,waw, which implies that writes complete at 11the target in the order in which they were issued, reads complete at the target before any 12writes that are issued after the reads, and writes complete at the target before any reads 13that are issued after the writes. Any subset of these four orderings can be specified. For 14example, if only read-after-read and write-after-write ordering is required, then the value 15of the accumulate_ordering key could be set to rar, waw. The order of values is not significant. 16Note that the above ordering semantics apply only to accumulate operations, not put 17

and get. Put and get within an epoch are unordered.

18 19

20

11.7.3 Progress

One-sided communication has the same progress requirements as point-to-point communi-21cation: once a communication is enabled it is guaranteed to complete. RMA calls must have 22local semantics, except when required for synchronization with other RMA calls. 23

There is some fuzziness in the definition of the time when a RMA communication 24 becomes enabled. This fuzziness provides to the implementor more flexibility than with 25point-to-point communication. Access to a target window becomes enabled once the corre-26sponding synchronization (such as MPI_WIN_FENCE or MPI_WIN_POST) has executed. On 27the origin process, an RMA communication may become enabled as soon as the correspond-28ing put, get or accumulate call has executed, or as late as when the ensuing synchronization 29 call is issued. Once the communication is enabled both at the origin and at the target, the 30 communication must complete. 31

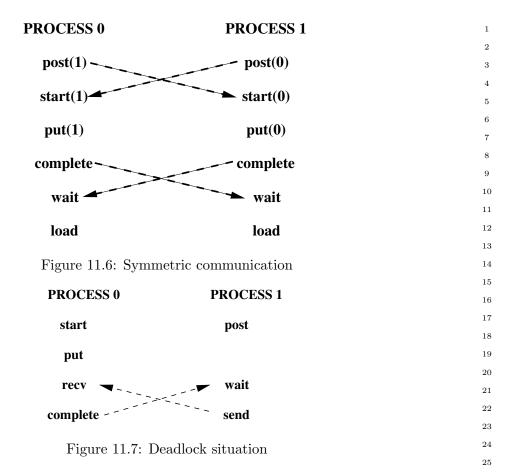
Consider the code fragment in Example 11.4. Some of the calls may block if the target 32 window is not posted. However, if the target window is posted, then the code fragment 33 must complete. The data transfer may start as soon as the put call occurs, but may be 34delayed until the ensuing complete call occurs. 35

Consider the code fragment in Example 11.5. Some of the calls may block if another 36 process holds a conflicting lock. However, if no conflicting lock is held, then the code 37 fragment must complete. 38

Consider the code illustrated in Figure 11.6. Each process updates the window of 39 the other process using a put operation, then accesses its own window. The post calls are 40 nonblocking, and should complete. Once the post calls occur, RMA access to the windows is 41 enabled, so that each process should complete the sequence of calls start-put-complete. Once 42these are done, the wait calls should complete at both processes. Thus, this communication 43 should not deadlock, irrespective of the amount of data transferred. 44

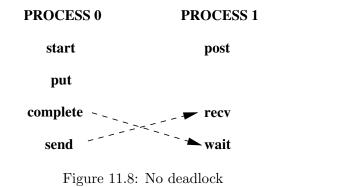
Assume, in the last example, that the order of the post and start calls is reversed at 45each process. Then, the code may deadlock, as each process may block on the start call, 46 waiting for the matching post to occur. Similarly, the program will deadlock if the order of 47the complete and wait calls is reversed at each process. 48

Unofficial Draft for Comment Only



The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice versa. Consider the code illustrated in Figure 11.7. This code will deadlock: the wait of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until process 1 calls send. Consider, on the other hand, the code illustrated in Figure 11.8. This code will not deadlock. Once process 1 calls post, then the sequence start, put, complete 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



 $46 \\ 47$

for send-receive communication and applies to RMA communication as well. Thus, in the example in Figure 11.8, the put and complete calls of process 0 should complete while process 1 is blocked on the receive call. This may require the involvement of process 1, e.g., to transfer the data put, while it is blocked on the receive call.

A similar issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call. Suppose that in the last example the send-receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not specify whether deadlock is avoided. Suppose that the blocking receive of process 1 is replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious. (*End of rationale*.)

11.7.4 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (End of advice to users.)

A coherence problem exists between variables kept in registers and the memory values of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI_WIN_UNIFIED.

The problem is illustrated by the following code:

33	Source of Process 1	Source of Process 2	Executed in Process 2
34	bbbb = 777	buff = 999	reg_A:=999
35	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
36	call MPI_PUT(bbbb		stop appl.thread
37	into buff of process 2)		buff:=777 in PUT handler
38			continue appl.thread
39	call MPI_WIN_FENCE	call MPI_WIN_FENCE	
40		ccc = buff	ccc:=reg_A
41			

In this example, variable **buff** is allocated in the register **reg_A** and therefore **ccc** will have the old value of **buff** and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 17.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran
 compilers will avoid this problem, without disabling compiler optimizations. However, in
 order to avoid register coherence problems in a completely portable manner, users should

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19 20

21 22

23

24

31 32

42

restrict their use of RMA windows to variables stored in modules or COMMON blocks. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10-17.1.20. Sections 17.1.17 to 17.1.17discuss several solutions for the problem in this example.

11.8 Examples

Example 11.13 The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array A, which contains the origin and target buffers of the put calls.

```
. . .
while (!converged(A)) {
 update(A);
 MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
 for(i=0; i < toneighbors; i++)</pre>
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                          todisp[i], 1, totype[i], win);
 MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
}
```

The same code could be written with get rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

Example 11.14 Same generic example, with more computation/communication overlap. We assume that the update phase is broken 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.

```
. . .
while (!converged(A)) {
  update_boundary(A);
 MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
  for(i=0; i < fromneighbors; i++)</pre>
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                     fromdisp[i], 1, fromtype[i], win);
 update_core(A);
 MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
}
```

The get communication can be concurrent with the core update, since they do not access the 43 44same 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 45with 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.

Unofficial Draft for Comment Only

1

2

3

4

56

7 8 9

10

11

1213

14

15

16

17

18

19

20

2122

23

 24

25

2627

28

29

30

3132

33

34

35

36

37

38

39

40

41

42

46

47

```
1
     Example 11.15 Same code as in Example 11.13, rewritten using post-start-complete-wait.
\mathbf{2}
3
     . . .
     while (!converged(A)) {
4
       update(A);
5
       MPI_Win_post(fromgroup, 0, win);
6
       MPI_Win_start(togroup, 0, win);
7
       for(i=0; i < toneighbors; i++)</pre>
8
         MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
9
                  todisp[i], 1, totype[i], win);
10
       MPI_Win_complete(win);
11
       MPI_Win_wait(win);
12
     }
13
14
15
     Example 11.16 Same example, with split phases, as in Example 11.14.
16
17
     . . .
     while (!converged(A)) {
18
       update_boundary(A);
19
       MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
20
       MPI_Win_start(fromgroup, 0, win);
21
       for(i=0; i < fromneighbors; i++)</pre>
22
         MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
23
                  fromdisp[i], 1, fromtype[i], win);
24
       update_core(A);
25
       MPI_Win_complete(win);
26
       MPI_Win_wait(win);
27
     }
28
29
30
     Example 11.17 A checkerboard, or double buffer communication pattern, that allows
^{31}
     more computation/communication overlap. Array A0 is updated using values of array A1,
32
     and vice versa. We assume that communication is symmetric: if process A gets data from
33
     process B, then process B gets data from process A. Window wini consists of array Ai.
34
35
     . . .
36
     if (!converged(A0,A1))
37
       MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
38
     MPI_Barrier(comm0);
     /* the barrier is needed because the start call inside the
39
40
     loop uses the nocheck option */
41
     while (!converged(A0, A1)) {
42
       /* communication on AO and computation on A1 */
       update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
43
       MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
44
       for(i=0; i < fromneighbors; i++)</pre>
45
         MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
46
47
                      fromdisp0[i], 1, fromtype0[i], win0);
48
       update1(A1); /* local update of A1 that is
```

Unofficial Draft for Comment Only

}

```
concurrent with communication that updates A0 */
MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
MPI_Win_complete(win0);
MPI_Win_wait(win0);
```

A process posts the local window associated with win0 before it completes RMA accesses to the remote windows associated with win1. When the wait(win1) call returns, then all neighbors of the calling process have posted the windows associated with win0. Conversely, when the wait(win0) call returns, then all neighbors of the calling process have posted the windows associated with win1. Therefore, the nocheck option can be used with the calls to MPI_WIN_START.

Put calls can be used, instead of get calls, if the area of array 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.

Process A:	Process B:	43
MPI_Win_lock_all	MPI_Win_lock_all	44
window location X		45
X=2		46
MPI_Win_sync		47
MPI_Barrier	MPI_Barrier	48

Unofficial Draft for Comment Only

 $\mathbf{2}$

```
1
\mathbf{2}
     MPI_Accumulate(X, MPI_SUM, -1)
                                                   MPI_Accumulate(X, MPI_SUM, -1)
3
4
     stack variable z
                                                    stack variable z
5
     do
                                                    do
6
       z = MPI_Get_accumulate(X,
                                                      z = MPI_Get_accumulate(X,
7
            MPI_NO_OP, 0)
                                                           MPI_NO_OP, 0)
8
       MPI_Win_flush(A)
                                                      MPI_Win_flush(A)
9
     while(z!=0)
                                                    while(z!=0)
10
^{11}
     MPI_Win_unlock_all
                                                    MPI_Win_unlock_all
12
```

12

14

15

16

17

18

19

44

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.

```
20
     Process A:
                                              Process B:
21
     window location X
                                              window location Y
22
     window location T
23
^{24}
     MPI_Win_lock_all
                                              MPI_Win_lock_all
25
     X=1
                                              Y=1
26
     MPI_Win_sync
                                              MPI_Win_sync
27
     MPI_Barrier
                                              MPI_Barrier
28
     MPI_Accumulate(T, MPI_REPLACE, 1)
                                              MPI_Accumulate(T, MPI_REPLACE, 0)
^{29}
     stack variables t,y
                                              stack variable t,x
30
     t=1
                                              t=0
^{31}
     y=MPI_Get_accumulate(Y,
                                              x=MPI_Get_accumulate(X,
32
        MPI_NO_OP, 0)
                                                 MPI_NO_OP, 0)
33
     while(y==1 && t==1) do
                                              while(x==1 && t==0) do
34
       y=MPI_Get_accumulate(Y,
                                                x=MPI_Get_accumulate(X,
35
          MPI_NO_OP, 0)
                                                   MPI_NO_OP, 0)
36
       t=MPI_Get_accumulate(T,
                                                t=MPI_Get_accumulate(T,
37
          MPI_NO_OP, 0)
                                                   MPI_NO_OP, 0)
38
       MPI_Win_flush_all
                                                MPI_Win_flush(A)
39
     done
                                              done
40
     // critical region
                                              // critical region
41
     MPI_Accumulate(X, MPI_REPLACE, 0)
                                              MPI_Accumulate(Y, MPI_REPLACE, 0)
42
     MPI_Win_unlock_all
                                              MPI_Win_unlock_all
43
```

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

Unofficial Draft for Comment Only

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: MPI_Win_lock_all	Process B: MPI_Win_lock_all	4
atomic location A		e
A=0		7
MPI_Win_sync		٤
MPI_Barrier	MPI_Barrier	ę
stack variable r=1	stack variable r=1	1
while(r != 0) do	while(r != 0) do	1
<pre>r = MPI_Compare_and_swap(A, 0, 1)</pre>	<pre>r = MPI_Compare_and_swap(A, 0, 1)</pre>	1
MPI_Win_flush(A)	MPI_Win_flush(A)	1
done	done	1
// critical region	<pre>// critical region</pre>	1
<pre>r = MPI_Compare_and_swap(A, 1, 0)</pre>	<pre>r = MPI_Compare_and_swap(A, 1, 0)</pre>	1
MPI_Win_unlock_all	MPI_Win_unlock_all	1

Example 11.21 The following example demonstrates the proper synchronization in the unified memory model when a data transfer is implemented with load and store in the 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 A	Process B	34
		35
MPI_WIN_LOCK_ALL(MPI_WIN_LOCK_ALL(36
MPI_MODE_NOCHECK,win)	MPI_MODE_NOCHECK,win)	37
		38
DO	DO	39
X=		40
		41
MPI_F_SYNC_REG(X)		42
MPI_WIN_SYNC(win)		43
MPI_SEND	MPI_RECV	44
	MPI_WIN_SYNC(win)	45
	MPI_F_SYNC_REG(X)	46
		47
	print X	48

print X

 24

```
1
\mathbf{2}
                                             MPI_F_SYNC_REG(X)
3
       MPI_RECV
                                             MPI_SEND
4
       MPI_F_SYNC_REG(X)
\mathbf{5}
     END DO
                                           END DO
6
7
     MPI_WIN_UNLOCK_ALL(win)
                                           MPI_WIN_UNLOCK_ALL(win)
8
9
     Example 11.22 The following example shows how request-based operations can be used
10
     to overlap communication with computation. Each process fetches, processes, and writes
11
     the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used to
12
     allow up to M communication operations to overlap with computation.
13
14
     int
                   i, j;
15
     MPI_Win
                   win;
16
     MPI_Request put_req[M] = { MPI_REQUEST_NULL };
17
     MPI_Request get_req;
18
     double
                   *baseptr;
19
     double
                   data[M][N];
20
21
     MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
22
       MPI_COMM_WORLD, &baseptr, &win);
23
^{24}
     MPI_Win_lock_all(0, win);
25
26
     for (i = 0; i < NSTEPS; i++) {</pre>
27
      if (i<M)
28
         j=i;
29
      else
30
         MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
^{31}
32
      MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
33
                 &get_req);
34
      MPI_Wait(&get_req,MPI_STATUS_IGNORE);
35
      compute(i, data[j], ...);
36
      MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
37
                 &put_req[j]);
38
     }
39
40
     MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
41
     MPI_Win_unlock_all(win);
42
43
44
     Example 11.23 The following example constructs a distributed shared linked list using
45
     dynamic windows. Initially process 0 creates the head of the list, attaches it to the window,
46
     and broadcasts the pointer to all processes. All processes then concurrently append N new
```

CHAPTER 11. ONE-SIDED COMMUNICATIONS

elements to the list. When a process attempts to attach its element to the tail of the

list it may discover that its tail pointer is stale and it must chase ahead to the new tail

47

before the element can be attached. This example requires some modification to work in an environment where the layout of the structures is different on different processes.

```
4
#define NUM_ELEMS 10
                                                                                    5
                                                                                    6
#define LLIST_ELEM_NEXT_RANK ( offsetof(llist_elem_t, next) + \
                                                                                    7
                                 offsetof(llist_ptr_t, rank) )
                                                                                    8
#define LLIST_ELEM_NEXT_DISP ( offsetof(llist_elem_t, next) + \
                                                                                    9
                                 offsetof(llist_ptr_t, disp) )
                                                                                    10
                                                                                    11
/* Linked list pointer */
                                                                                    12
typedef struct {
                                                                                    13
 MPI_Aint disp;
                                                                                    14
  int
           rank;
                                                                                    15
} llist_ptr_t;
                                                                                    16
                                                                                    17
/* Linked list element */
                                                                                    18
typedef struct {
                                                                                    19
  llist_ptr_t next;
                                                                                    20
  int value;
                                                                                    21
} llist_elem_t;
                                                                                    22
                                                                                    23
const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM, -1 };
                                                                                    24
                                                                                    25
/* List of locally allocated list elements. */
                                                                                    26
static llist_elem_t **my_elems = NULL;
                                                                                    27
static int my_elems_size = 0;
                                                                                    28
static int my_elems_count = 0;
                                                                                    29
                                                                                    30
/* Allocate a new shared linked list element */
                                                                                    31
MPI_Aint alloc_elem(int value, MPI_Win win) {
                                                                                    32
  MPI_Aint disp;
                                                                                    33
  llist_elem_t *elem_ptr;
                                                                                    34
                                                                                    35
  /* Allocate the new element and register it with the window */
                                                                                    36
  MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
                                                                                    37
  elem_ptr->value = value;
                                                                                    38
  elem_ptr->next = nil;
                                                                                    39
  MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
                                                                                    40
                                                                                    41
                                                                                    42
  /* Add the element to the list of local elements so we can free
     it later. */
                                                                                    43
  if (my_elems_size == my_elems_count) {
                                                                                    44
    my_elems_size += 100;
                                                                                    45
    my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
                                                                                    46
  }
                                                                                    47
  my_elems[my_elems_count] = elem_ptr;
                                                                                    48
```

Unofficial Draft for Comment Only

1

2

```
1
       my_elems_count++;
2
3
       MPI_Get_address(elem_ptr, &disp);
4
       return disp;
5
     }
6
7
     int main(int argc, char *argv[]) {
8
       int
                      procid, nproc, i;
9
       MPI_Win
                      llist_win;
10
       llist_ptr_t
                     head_ptr, tail_ptr;
11
12
       MPI_Init(&argc, &argv);
13
14
       MPI_Comm_rank(MPI_COMM_WORLD, &procid);
15
       MPI_Comm_size(MPI_COMM_WORLD, &nproc);
16
17
       MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
18
19
       /* Process 0 creates the head node */
20
       if (procid == 0)
21
         head_ptr.disp = alloc_elem(-1, llist_win);
22
       /* Broadcast the head pointer to everyone */
23
^{24}
       head_ptr.rank = 0;
25
       MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
26
       tail_ptr = head_ptr;
27
28
       /* Lock the window for shared access to all targets */
29
       MPI_Win_lock_all(0, llist_win);
30
^{31}
       /* All processes concurrently append NUM_ELEMS elements to the list */
32
       for (i = 0; i < NUM_ELEMS; i++) {</pre>
33
         llist_ptr_t new_elem_ptr;
34
         int success;
35
36
         /* Create a new list element and attach it to the window */
37
         new_elem_ptr.rank = procid;
38
         new_elem_ptr.disp = alloc_elem(procid, llist_win);
39
40
         /* Append the new node to the list. This might take multiple
41
            attempts if others have already appended and our tail pointer
42
            is stale. */
43
         do {
44
           llist_ptr_t next_tail_ptr = nil;
45
46
           MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
47
                (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
48
                MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
```

. . .

```
1
        llist_win);
                                                                                   \mathbf{2}
                                                                                   3
    MPI_Win_flush(tail_ptr.rank, llist_win);
    success = (next_tail_ptr.rank == nil.rank);
                                                                                   4
                                                                                   5
    if (success) {
                                                                                   6
                                                                                   7
      MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
          MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
                                                                                   8
          MPI_AINT, MPI_REPLACE, llist_win);
                                                                                   9
                                                                                   10
                                                                                   11
      MPI_Win_flush(tail_ptr.rank, llist_win);
      tail_ptr = new_elem_ptr;
                                                                                   12
                                                                                   13
    } else {
                                                                                   14
                                                                                   15
      /* Tail pointer is stale, fetch the displacement. May take
                                                                                   16
         multiple tries if it is being updated. */
                                                                                   17
      do {
        MPI_Get_accumulate(NULL, 0, MPI_AINT, &next_tail_ptr.disp,
                                                                                   18
                                                                                   19
             1, MPI_AINT, tail_ptr.rank,
                                                                                   20
             MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
                                                                                   21
             1, MPI_AINT, MPI_NO_OP, llist_win);
                                                                                   22
        MPI_Win_flush(tail_ptr.rank, llist_win);
                                                                                   23
                                                                                   24
      } while (next_tail_ptr.disp == nil.disp);
                                                                                   25
      tail_ptr = next_tail_ptr;
                                                                                   26
    }
  } while (!success);
                                                                                   27
}
                                                                                   28
                                                                                   29
                                                                                   30
MPI_Win_unlock_all(llist_win);
MPI_Barrier(MPI_COMM_WORLD);
                                                                                   31
                                                                                   32
                                                                                   33
/* Free all the elements in the list */
                                                                                   34
for ( ; my_elems_count > 0; my_elems_count--) {
  MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
                                                                                   35
  MPI_Free_mem(my_elems[my_elems_count-1]);
                                                                                   36
                                                                                   37
}
                                                                                   38
MPI_Win_free(&llist_win);
                                                                                   39
                                                                                   40
                                                                                   41
                                                                                   42
                                                                                   43
                                                                                   44
                                                                                   45
```

1			
2			
3			
4			
5			
6			
7			
8			
9			
10 11			
11			
12			
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			
38 39			
40			
40 41			
41			
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	the MPI i plication. formed by	mplementation, and the op For a generalized request, the application; therefore,	ation associated with the request is performed by eration completes without intervention by the ap- the operation associated with the request is per- the application must notify MPI through a call to he operation completes. MPI maintains the "comple-
6			Any other request state has to be maintained by the
7	user.		
8 9	A nev	v generalized request is start	ed with
10			
11	MPI_GRE	QUEST_START(query_fn, fre	e_fn, cancel_fn, extra_state, request)
12 13	IN	query_fn	callback function invoked when request status is queried (function)
14 15 16	IN	free_fn	callback function invoked when request is freed (function)
17 18	IN	cancel_fn	callback function invoked when request is cancelled (function)
19	IN	extra_state	extra state
20	OUT	request	generalized request (handle)
21 22			
23	C bindin	•	
24	int MPI_(<pre>est_query_function *query_fn,</pre>
25		MPI_Grequest_free_f	L_function *iree_in,
26 27		MPI_Request *reques	
28	F08 bind	ing -	
29			e_fn, cancel_fn, extra_state, request,
30	-	ierror)	
31 32	PROCEDURE(MPI_Grequest_query_function) :: query_fn		
33	PROCEDURE(MPI_Grequest_free_function) :: free_fn PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn		
34		-	D), INTENT(IN) :: extra_state
35		(MPI_Request), INTENT(OU	
36		GER, OPTIONAL, INTENT(OU	1
37	F binding	a.	
38			E_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
39 40		IERROR)	
41	EXTER	RNAL QUERY_FN, FREE_FN,	CANCEL_FN
42	INTEC	GER(KIND=MPI_ADDRESS_KIN	D) EXTRA_STATE
43	INTEC	GER REQUEST, IERROR	
44			
45	Advi	<i>ice to users.</i> Note that a	generalized request is of the same type as regular
46	requ	ests, in C and Fortran. (End	d of advice to users.)
47		11	
48	The c	can starts a generalized reque	est and returns a handle to it in request.

The syntax and meaning of the callback functions are listed below. All callback functions are passed the extra_state argument that was associated with the request by the starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined state for the request.

In C, the query function is
<pre>typedef int MPI_Grequest_query_function(void *extra_state,</pre>
MPI_Status *status);
in Fortran with the mpi_f08 module
ABSTRACT INTERFACE
SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
TYPE(MPI_Status) :: status
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
INTEGER :: ierror
in Fortran with the mpi module and mpif.h
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), IERROR

INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

The query_fn function computes the status that should be returned for the generalized request. The status also includes information about successful/unsuccessful cancellation of the request (result to be returned by MPI_TEST_CANCELLED).

The query_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that 23completed the generalized request associated with this callback. The callback function is 24 also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when 25the call occurs. In both cases, the callback is passed a reference to the corresponding 26status variable passed by the user to the MPI call; the status set by the callback function 27is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or 28MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI 29will pass a valid status object to query_fn, and this status will be ignored upon return of the 30 callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE 31 is called on the request; it may be invoked several times for the same generalized request, 32 e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also 33 that a call to MPI_{WAIT|TEST}{SOME|ALL} may cause multiple invocations of query_fn 34 callback functions, one for each generalized request that is completed by the MPI call. The 35order of these invocations is not specified by MPI. 36

In C, the free function is 37 typedef int MPI_Grequest_free_function(void *extra_state); 38 39 in Fortran with the mpi_f08 module 40 ABSTRACT INTERFACE 41 SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) 42INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 43 INTEGER :: ierror 44in Fortran with the mpi module and mpif.h 45SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR) 46INTEGER IERROR 47INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

1

2

3

19

20

21

22

1 The free_fn function is invoked to clean up user-allocated resources when the generalized $\mathbf{2}$ request is freed. 3 The free_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that 4 completed the generalized request associated with this callback. free_fn is invoked after $\mathbf{5}$ the call to query_fn for the same request. However, if the MPI call completed multiple 6 generalized requests, the order in which free_fn callback functions are invoked is not specified 7by MPI. 8 The free_fn callback is also invoked for generalized requests that are freed by a call 9 to MPI_REQUEST_FREE (no call to MPI_{WAIT|TEST}{ANY|SOME|ALL} will occur for 10 such a request). In this case, the callback function will be called either in the MPI call 11MPI_REQUEST_FREE(request), or in the MPI call MPI_GREQUEST_COMPLETE(request), 12whichever happens last, i.e., in this case the actual freeing code is executed as soon as both 13calls MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request 14is not deallocated until after free_fn completes. Note that free_fn will be invoked only once 15per request by a correct program. 16Advice to users. Calling MPI_REQUEST_FREE(request) will cause the request handle 17 to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer 18 valid. However, user copies of this handle are valid until after free_fn completes since 19 MPI does not deallocate the object until then. Since free_fn is not called until after 20MPI_GREQUEST_COMPLETE, the user copy of the handle can be used to make this 21call. Users should note that MPI will deallocate the object after free_fn executes. At 22 this point, user copies of the request handle no longer point to a valid request. MPI will 23not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to avoid 24accessing this stale handle. This is a special case in which MPI defers deallocating the 25object until a later time that is known by the user. (End of advice to users.) 2627In C, the cancel function is 28typedef int MPI_Grequest_cancel_function(void *extra_state, int complete); 2930 in Fortran with the mpi_f08 module 31ABSTRACT INTERFACE 32 SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror) 33 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 34 LOGICAL :: complete 35INTEGER :: ierror 36 in Fortran with the mpi module and mpif.h 37 SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR) 38 INTEGER IERROR 39 INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE 40 LOGICAL COMPLETE 41 42The cancel_fn function is invoked to start the cancelation of a generalized request. 43It is called by MPI_CANCEL(request). MPI passes complete=true to the callback function 44if MPI_GREQUEST_COMPLETE was already called on the request, and 45complete=false otherwise. 46All callback functions return an error code. The code is passed back and dealt with as 47appropriate for the error code by the MPI function that invoked the callback function. For 48

example, if error codes are returned then the error code returned by the callback function

will be returned by the MPI function that invoked the callback function. In the case of an MPI_{WAIT|TEST}{ANY} call that invokes both query_fn and free_fn, the MPI call will 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_GREG	QUEST_COMPLETE(request)	
INOUT	request	generalized request (handle)
C binding int MPI_G	g request_complete(MPI_Requ	uest request)
TYPE(ing lest_complete(request, ien MPI_Request), INTENT(IN) ER, OPTIONAL, INTENT(OUT)	:: request
-	g EST_COMPLETE(REQUEST, IEF ER REQUEST, IERROR	RROR)
The ca	all informs MPI that the operat	tions represented by the generalized request

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, 37 new nonblocking operations should be defined so that the general semantic rules about MPI 38 calls such as MPI_TEST, MPI_REQUEST_FREE, or MPI_CANCEL still hold. For example, 39 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 41 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 43 other processes (the operation has acquired "local" semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

A call to MPI_GREQUEST_COMPLETE may unblock a Advice to implementors.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5

6 $\overline{7}$

8 9

10

11

12

13

14

33

34

35

36

40

42

44

45

4647

1

 $\mathbf{2}$

3 4

5 6

7

8

9

blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (*End of advice to implementors.*)

```
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.

```
10
     typedef struct {
11
        MPI_Comm comm;
12
        int tag;
13
        int root;
14
        int valin;
15
        int *valout;
16
        MPI_Request request;
17
        } ARGS;
18
19
20
     int myreduce(MPI_Comm comm, int tag, int root,
21
                   int valin, int *valout, MPI_Request *request)
22
     {
23
        ARGS *args;
24
        pthread_t thread;
25
26
        /* start request */
27
        MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);
28
29
        args = (ARGS*)malloc(sizeof(ARGS));
30
        args->comm = comm;
^{31}
        args->tag = tag;
32
        args->root = root;
33
        args->valin = valin;
34
        args->valout = valout;
35
        args->request = *request;
36
37
        /* spawn thread to handle request */
38
        /* The availability of the pthread_create call is system dependent */
39
        pthread_create(&thread, NULL, reduce_thread, args);
40
41
        return MPI_SUCCESS;
42
     }
43
44
     /* thread code */
45
     void* reduce_thread(void *ptr)
46
     {
47
        int lchild, rchild, parent, lval, rval, val;
48
```

```
1
   MPI_Request req[2];
                                                                                    2
   ARGS *args;
                                                                                    3
   args = (ARGS*)ptr;
                                                                                    4
                                                                                    5
                                                                                    6
   /* compute left and right child and parent in tree; set
                                                                                    7
      to MPI_PROC_NULL if does not exist */
                                                                                    8
   /* code not shown */
                                                                                    9
   . . .
                                                                                    10
                                                                                    11
   MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
   MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
                                                                                    12
   MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
                                                                                    13
                                                                                    14
   val = lval + args->valin + rval;
                                                                                    15
   MPI_Send(&val, 1, MPI_INT, parent, args->tag, args->comm);
   if (parent == MPI_PROC_NULL) *(args->valout) = val;
                                                                                    16
                                                                                    17
   MPI_Grequest_complete((args->request));
                                                                                    18
   free(ptr);
                                                                                    19
   return(NULL);
}
                                                                                    20
                                                                                    21
int query_fn(void *extra_state, MPI_Status *status)
                                                                                    22
                                                                                    23
{
                                                                                    24
   /* always send just one int */
                                                                                    25
   MPI_Status_set_elements(status, MPI_INT, 1);
                                                                                    26
   /* can never cancel so always true */
   MPI_Status_set_cancelled(status, 0);
                                                                                    27
   /* choose not to return a value for this */
                                                                                    28
                                                                                    29
   status->MPI_SOURCE = MPI_UNDEFINED;
                                                                                    30
   /* tag has no meaning for this generalized request */
                                                                                    31
   status->MPI_TAG = MPI_UNDEFINED;
                                                                                    32
   /* this generalized request never fails */
                                                                                    33
   return MPI_SUCCESS;
                                                                                    34
}
                                                                                    35
                                                                                    36
                                                                                    37
int free_fn(void *extra_state)
                                                                                    38
{
                                                                                    39
   /* this generalized request does not need to do any freeing */
   /* as a result it never fails here */
                                                                                    40
                                                                                    41
   return MPI_SUCCESS;
                                                                                    42
}
                                                                                    43
                                                                                    44
int cancel_fn(void *extra_state, int complete)
                                                                                    45
                                                                                    46
{
                                                                                    47
   /* This generalized request does not support cancelling.
                                                                                    48
      Abort if not already done. If done then treat as if cancel failed.*/
```

12.3 Associating Information with Status

¹¹ MPI supports several different types of requests besides those for point-to-point operations. ¹² These range from MPI calls for I/O to generalized requests. It is desirable to allow these ¹³ calls to use the same request mechanism, which allows one to wait or test on different ¹⁴ types of requests. However, MPI_{TEST|WAIT}{ANY|SOME|ALL} returns a status with ¹⁶ information about the request. With the generalization of requests, one needs to define ¹⁷ 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:

```
29
30
```

9

10

MPI_STATUS_SET_ELEMENTS(status, datatype, count)

```
31
       INOUT
                 status
                                             status with which to associate count (Status)
32
       IN
                 datatype
                                             datatype associated with count (handle)
33
34
       IN
                 count
                                             number of elements to associate with status (non-negative
35
                                             integer)
36
37
     C binding
38
     int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
39
                    int count)
40
     F08 binding
41
     MPI_Status_set_elements(status, datatype, count, ierror)
42
          TYPE(MPI_Status), INTENT(INOUT) :: status
43
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
          INTEGER, INTENT(IN) :: count
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     F binding
48
```

12.3.	ASSOCI	ATING INFORMATION	WITH STATUS	531		
		ET_ELEMENTS(STATUS, DAT STATUS(MPI_STATUS_SIZE)	CATYPE, COUNT, IERROR) , DATATYPE, COUNT, IERROR	1 2 3		
				4		
MPI_	STATUS_	SET_ELEMENTS_X(status	, datatype, count)	5		
INC	DUT sta	tus	status with which to associate count (Status)	7		
IN	dat	atype	datatype associated with count (handle)	8		
IN	ςοι	unt	number of elements to associate with status (inte	ger) 9 10		
C bi	nding			11		
		is_set_elements_x(MPI_S MPI_Count count)	Status *status, MPI_Datatype datatype,	12 13 14		
F08	binding			15		
	MPI_Status_set_elements_x(status, datatype, count, ierror)					
		Status), INTENT(INOUT)		17		
		_Datatype), INTENT(IN)		18 19		
		<pre>XIND=MPI_COUNT_KIND), 1 OPTIONAL, INTENT(OUT)</pre>		20		
		UTIONAL, INILAT(UUI)		21		
	nding			22		
		STATUS(MPI_STATUS_SIZE)	DATATYPE, COUNT, IERROR)	23		
		(IND=MPI_COUNT_KIND) CC		24 25		
1	These fun	ctions modify the opaque	part of status so that a call to	26		
		MENTS or MPI_GET_ELE ompatible value.	MENTS_X will return count. MPI_GET_COU	28		
	Dational	The number of elemen	to is not instead of the count because the for	29 mer 30		
			ts is set instead of the count because the for of datatypes. (<i>End of rationale.</i>)	mer 30 31		
	can dear			32		
	-		UNT(status, datatype, count),	33		
		MENTS(status, datatype, c		34		
		· · · · · ·	e, count) must use a datatype argument that			
			argument that was used in the call to STATUS_SET_ELEMENTS_X.	36 37		
			5777105_3E1_EEEMENT5_7.	38		
	Rationale	e. The requirement of m	atching type signatures for these calls is sim-			
			count is set by a receive operation: in that c	<i>,</i>		
			_GET_ELEMENTS, and MPI_GET_ELEMENT			
		a datatype with the same s <i>ationale.</i>)	signature as the datatype used in the receive of			
	(Ena Of 1	<i>uuuuu</i> .)		43 44		
				44 45		
				46		
				47		
				48		

```
1
      MPI_STATUS_SET_CANCELLED(status, flag)
2
       INOUT
                                              status with which to associate cancel flag (Status)
                 status
3
       IN
                 flag
                                              if true, indicates request was cancelled (logical)
4
5
6
      C binding
\overline{7}
      int MPI_Status_set_cancelled(MPI_Status *status, int flag)
8
     F08 binding
9
     MPI_Status_set_cancelled(status, flag, ierror)
10
          TYPE(MPI_Status), INTENT(INOUT) ::
                                                    status
11
          LOGICAL, INTENT(IN) :: flag
12
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
13
14
      F binding
15
     MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
16
          INTEGER STATUS(MPI_STATUS_SIZE), IERROR
17
          LOGICAL FLAG
18
          If flag is set to true then a subsequent call to MPI_TEST_CANCELLED(status, flag) will
19
      also return flag = true, otherwise it will return false.
20
21
                              Users are advised not to reuse the status fields for values other
           Advice to users.
22
           than those for which they were intended. Doing so may lead to unexpected results
23
           when using the status object. For example, calling MPI_GET_ELEMENTS may cause
24
           an error if the value is out of range or it may be impossible to detect such an error.
25
           The extra_state argument provided with a generalized request can be used to return
26
           information that does not logically belong in status. Furthermore, modifying the
27
           values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable
```

28 29 30

31 32

12.4 MPI and Threads

This section specifies the interaction between MPI calls and threads. The section lists 33 minimal requirements for thread compliant MPI implementations and defines functions 34 that can be used for initializing the thread environment. MPI may be implemented in 35 environments where threads are not supported or perform poorly. Therefore, MPI imple-36 mentations are not required to be thread compliant as defined in this section. Regard-37 less of whether or not the MPI implementation is thread compliant, MPI_INITIALIZED, 38 MPI_FINALIZED, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN, MPI_GET_VERSION 39 and MPI_GET_LIBRARY_VERSION must always be thread-safe. When a thread is exe-40 cuting one of these routines, if another concurrently running thread also makes an MPI call, 41 the outcome will be as if the calls executed in some order. 42

results and is strongly discouraged. (End of advice to users.)

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.

- 46
- 47
- 48

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.)

Advice to users. It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. (End of advice to users.)

The two main requirements for a thread-compliant implementation are listed below.

- 1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
- 2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

Example 12.2 Process 0 consists of two threads. The first thread executes a blocking 31 send call MPI_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes 32 33 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 34always succeed. According to the first requirement, the execution will correspond to some 35 interleaving of the two calls. According to the second requirement, a call can only block 36 the calling thread and cannot prevent progress of the other thread. If the send call went 37 ahead of the receive call, then the sending thread may block, but this will not prevent 38the 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. 42The 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 45a time, e.g., by protecting MPI code with one process-global lock. However, blocked 46operations cannot hold the lock, as this would prevent progress of other threads in 47the process. The lock is held only for the duration of an atomic, locally-completing 48

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

5

6 $\overline{7}$

8

9

10

11

1213

14

15

16

17 18

19

20

21

22 23

 24

25

26

27

28

2930

39

40 41

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.

11 12 13

14

20

21

22

23

24

25

26

27

1

2

3

4

5 6

7

8

9

10

Rationale. This constraint simplifies implementation. (End of rationale.)

¹⁵ Multiple threads completing the same request. A program in which two threads block, wait-¹⁶ ing on the same request, is erroneous. Similarly, the same request cannot appear in the ¹⁷ array of requests of two concurrent MPI_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a ¹⁸ request can only be completed once. Any combination of wait or test that violates this rule ¹⁹ is erroneous.

Rationale. This restriction is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is possible to post a second wait on the same handle. With threads, an

MPI_WAIT{ANY|SOME|ALL} may be blocked without having nullified its request(s) so it becomes the user's responsibility to avoid using the same request in an MPI_WAIT on another thread. This constraint also simplifies implementation, as only one thread will be blocked on any communication or I/O event. (*End of rationale.*)

28 29

Probe A receive call that uses source and tag values returned by a preceding call to MPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only if there was no other matching receive after the probe and before that receive. In a multithreaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process. Alternatively, MPI_MPROBE or MPI_IMPROBE can be used.

37

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.*)

47 48

42

43

44

45

Rationale. As specified in MPI_FILE_OPEN and MPI_WIN_CREATE, a file handle and a window handle inherit only the group of processes of the underlying communicator, but not the communicator itself. Accesses to communicators, window handles and file handles cannot affect one another. (*End of rationale.*)

Advice to implementors. If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (*End of advice to implementors.*)

Exception handlers An exception handler does not necessarily execute in the context of the thread that made the exception-raising MPI call; the exception handler may be executed by a thread that is distinct from the thread that will return the error code.

Rationale. The MPI implementation may be multithreaded, so that part of the communication protocol may execute on a thread that is distinct from the thread that made the MPI call. The design allows the exception handler to be executed on the thread where the exception occurred. (*End of rationale.*)

Interaction with signals and cancellations The outcome is undefined if a thread that executes an MPI call is cancelled (by another thread), or if a thread catches a signal while executing an MPI call. However, a thread of an MPI process may terminate, and may catch signals or be cancelled by another thread when not executing MPI calls.

Rationale. Few C library functions are signal safe, and many have cancellation points — points at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancel-safe" or "async-signal-safe"). (*End of rationale.*)

Advice to users. Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to sigwait for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

Advice to implementors. The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (End of advice to implementors.)

12.4.3 Initialization

The following function may be used to initialize MPI, and to initialize the MPI thread environment, instead of MPI_INIT.

MPI_INIT_THREAD(required, provided)

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)

C binding

 $\mathbf{2}$

 $\overline{7}$

 24

 31

```
1
     int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
\mathbf{2}
     F08 binding
3
     MPI_Init_thread(required, provided, ierror)
4
          INTEGER, INTENT(IN) :: required
5
          INTEGER, INTENT(OUT) :: provided
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     F binding
9
     MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
10
          INTEGER REQUIRED, PROVIDED, IERROR
11
12
           Advice to users. In C, the passing of argc and argv is optional, as with MPI_INIT as
13
           discussed in Section 8.7. In C, null pointers may be passed in their place. (End of
14
           advice to users.)
15
          This call initializes MPI in the same way that a call to MPI_INIT would. In addition,
16
     it initializes the thread environment. The argument required is used to specify the desired
17
     level of thread support. The possible values are listed in increasing order of thread support.
18
19
     MPI_THREAD_SINGLE Only one thread will execute.
20
     MPI_THREAD_FUNNELED The process may be multi-threaded, but the application must
21
           ensure that only the main thread makes MPI calls (for the definition of main thread,
22
           see MPI_IS_THREAD_MAIN on page ??).
23
^{24}
     MPI_THREAD_SERIALIZED The process may be multi-threaded, and multiple threads may
25
           make MPI calls, but only one at a time: MPI calls are not made concurrently from
26
           two distinct threads (all MPI calls are "serialized").
27
     MPI_THREAD_MULTIPLE Multiple threads may call MPI, with no restrictions.
28
29
     These values are monotonic; i.e., MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED <
30
     MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE.
^{31}
          Different processes in MPI_COMM_WORLD may require different levels of thread sup-
32
     port.
33
          The call returns in provided information about the actual level of thread support that
34
     will be provided by MPI. It can be one of the four values listed above.
35
          The level(s) of thread support that can be provided by MPI_INIT_THREAD will depend
36
     on the implementation, and may depend on information provided by the user before the
37
     program started to execute (e.g., with arguments to mpiexec). If possible, the call will
38
     return provided = required. Failing this, the call will return the least supported level such
39
     that provided > required (thus providing a stronger level of support than required by the
40
     user). Finally, if the user requirement cannot be satisfied, then the call will return in
41
     provided the highest supported level.
42
          A thread compliant MPI implementation will be able to return provided
43
     = MPI_THREAD_MULTIPLE. Such an implementation may always return provided
44
     = MPI_THREAD_MULTIPLE, irrespective of the value of required.
45
          An MPI library that is not thread compliant must always return
46
     provided=MPI_THREAD_SINGLE, even if MPI_INIT_THREAD is called on a multithreaded
47
     process. The library should also return correct values for the MPI calls that can be executed
48
     before initialization, even if multiple threads have been spawned.
```

Such code is erroneous, but if the MPI initialization is performed by a Rationale. library, the error cannot be detected until MPI_INIT_THREAD is called. The requirements in the previous paragraph ensure that the error can be properly detected. (End of rationale.)

A call to MPI_INIT has the same effect as a call to MPI_INIT_THREAD with a required = MPI_THREAD_SINGLE.

Vendors may provide (implementation dependent) means to specify the level(s) of thread support available when the MPI program is started, e.g., with arguments to mpiexec. This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for example, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is available. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required; a call to MPI_INIT will also initialize the MPI thread support level to MPI_THREAD_MULTIPLE. Suppose, instead, that an MPI program has been started 14so that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD will return provided = required; alternatively, a call to MPI_INIT will initialize the MPI thread 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.

48

1

 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

11

12

13

15

16

17 18

19

20

21

22

23

 24

25

26

2728

29

30

 31

32 33

34

3536

37

38

39

40

41

42

```
1
     MPI_QUERY_THREAD(provided)
\mathbf{2}
       OUT
                 provided
                                              provided level of thread support (integer)
3
4
      C binding
5
      int MPI_Query_thread(int *provided)
6
\overline{7}
     F08 binding
8
     MPI_Query_thread(provided, ierror)
9
          INTEGER, INTENT(OUT) :: provided
10
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
11
     F binding
12
     MPI_QUERY_THREAD(PROVIDED, IERROR)
13
          INTEGER PROVIDED, IERROR
14
15
          The call returns in provided the current level of thread support, which will be the value
16
     returned in provided by MPI_INIT_THREAD, if MPI was initialized by a call to
17
     MPI_INIT_THREAD().
18
19
      MPI_IS_THREAD_MAIN(flag)
20
21
       OUT
                 flag
                                              true if calling thread is main thread, false otherwise
22
                                               (logical)
23
^{24}
      C binding
25
      int MPI_Is_thread_main(int *flag)
26
27
     F08 binding
28
     MPI_Is_thread_main(flag, ierror)
          LOGICAL, INTENT(OUT) :: flag
29
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
      F binding
32
     MPI_IS_THREAD_MAIN(FLAG, IERROR)
33
          LOGICAL FLAG
34
          INTEGER IERROR
35
36
          This function can be called by a thread to determine if it is the main thread (the thread
37
      that called MPI_INIT or MPI_INIT_THREAD).
38
          All routines listed in this section must be supported by all MPI implementations.
39
           Rationale.
                         MPI libraries are required to provide these calls even if they do not
40
           support threads, so that portable code that contains invocations to these functions
41
42
           can link correctly. MPI_INIT continues to be supported so as to provide compatibility
           with current MPI codes. (End of rationale.)
43
44
           Advice to users. It is possible to spawn threads before MPI is initialized, but no MPI
45
           call other than MPI_GET_VERSION, MPI_INITIALIZED, or MPI_FINALIZED should
46
           be executed by these threads, until MPI_INIT_THREAD is invoked by one thread
47
48
```

(which, thereby, becomes the main thread). In particular, it is possible to enter the MPI execution with a multi-threaded process.

The level of thread support provided is a global property of the MPI process that can be specified only once, when MPI is initialized on that process (or before). Portable third party libraries have to be written so as to accommodate any provided level of thread support. Otherwise, their usage will be restricted to specific level(s) of thread support. If such a library can run only with specific level(s) of thread support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be used to check whether the user initialized MPI to the correct level of thread support and, if not, raise an exception. (*End of advice to users.*)

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\overline{7}$

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, 51, 57], and disk-directed I/O [43]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

13.1.1 Definitions

- file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.
- **displacement** A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement."
- etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

 24

filetype A *filetype* is the basis for partitioning a file among processes and defines a template
 for accessing the file. A filetype is either a single etype or a derived MPI datatype
 constructed from multiple instances of the same etype. In addition, the extent of any
 hole in the filetype must be a multiple of the etype's extent. The displacements in the
 typemap of the filetype are not required to be distinct, but they must be non-negative
 and monotonically nondecreasing.

view A view defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 13.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).

etype
filetype holes
tiling a file with the filetype: displacement accessible data
Figure 13.1: Etypes and filetypes
A group of processes can use complementary views to achieve a global data distribution such as a scatter/gather pattern (see Figure 13.2).
etype
process 0 filetype
process 1 filetype
process 2 filetype
tiling a file with the filetypes:
displacement
Figure 13.2: Partitioning a file among parallel processes

offset An offset is a position in the file relative to the current view, expressed as a count of
 etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0
 is the location of the first etype visible in the view (after skipping the displacement and
 any initial holes in the view). For example, an offset of 2 for process 1 in Figure 13.2 is
 the position of the eighth etype in the file after the displacement. An "explicit offset"
 is an offset that is used as an argument in explicit data access routines.

Unofficial Draft for Comment Only

- file size and end of file The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file.
- file pointer A *file pointer* is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.
- file handle A *file handle* is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE. All operations on an open file reference the file through the file handle.

13.2 File Manipulation

13.2.1 Opening a File

MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN	comm	communicator (handle)	21
			22
IN	filename	name of file to open (string)	23
IN	amode	file access mode (integer)	24
IN	info	info object (handle)	25
OUT	fh	new file handle (handle)	26
			27

C binding

F08 binding

```
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
F binding
MDI_FILE_OPEN(COMM__ELLENAME__AMODE__INEO__ELL_EDDOD)
```

MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
 CHARACTER*(*) FILENAME
 INTEGER COMM, AMODE, INFO, FH, IERROR

MPI_FILE_OPEN opens the file identified by the file name filename on all processes in the comm communicator group. MPI_FILE_OPEN is a collective routine: all processes must provide the same value for amode, and all processes must provide filenames that reference the

Unofficial Draft for Comment Only

1 same file. (Values for info may vary.) comm must be an intracommunicator; it is erroneous to $\mathbf{2}$ pass an intercommunicator to MPI_FILE_OPEN. Errors in MPI_FILE_OPEN are raised using 3 the default file error handler (see Section 13.7). A process can open a file independently of 4 other processes by using the MPI_COMM_SELF communicator. The file handle returned, fh, 5can be subsequently used to access the file until the file is closed using MPI_FILE_CLOSE. 6 Before calling MPI_FINALIZE, the user is required to close (via MPI_FILE_CLOSE) all files $\overline{7}$ that were opened with MPI_FILE_OPEN. Note that the communicator comm is unaffected 8 by MPI_FILE_OPEN and continues to be usable in all MPI routines (e.g., MPI_SEND). 9 Furthermore, the use of **comm** will not interfere with I/O behavior.

¹⁰ The format for specifying the file name in the filename argument is implementation ¹¹ 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.

12

13

14

15

16

17 18

19

20

21

22

23

24 25

26

27

28

29

32

33 34

35

36 37

38

39 40

41 42

43

44 45

Advice to users. C users can use bit vector OR (|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Fortran 77 users can use (nonportably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition.). (End of advice to users.)

Advice to implementors. The values of these constants must be defined such that the bitwise OR and the sum of any distinct set of these constants is equivalent. (*End of advice to implementors.*)

The modes MPI_MODE_RDONLY, MPI_MODE_RDWR, MPI_MODE_WRONLY, MPI_MODE_CREATE, and MPI_MODE_EXCL have identical semantics to their POSIX counterparts [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.

Unofficial Draft for Comment Only

 24

```
1
     13.2.2 Closing a File
\mathbf{2}
3
4
     MPI_FILE_CLOSE(fh)
5
       INOUT
                 fh
                                              file handle (handle)
6
7
     C binding
8
     int MPI_File_close(MPI_File *fh)
9
10
     F08 binding
11
     MPI_File_close(fh, ierror)
12
          TYPE(MPI_File), INTENT(INOUT) :: fh
13
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                   ierror
14
     F binding
15
16
     MPI_FILE_CLOSE(FH, IERROR)
17
          INTEGER FH, IERROR
18
          MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
19
     MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
20
     opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
21
     MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.
22
23
           Advice to users. If the file is deleted on close, and there are other processes currently
^{24}
           accessing the file, the status of the file and the behavior of future accesses by these
25
           processes are implementation dependent. (End of advice to users.)
26
27
          The user is responsible for ensuring that all outstanding nonblocking requests and
28
     split collective operations associated with fh made by a process have completed before that
29
     process calls MPI_FILE_CLOSE.
30
          The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
^{31}
     MPI_FILE_NULL.
32
33
     13.2.3 Deleting a File
34
35
36
     MPI_FILE_DELETE(filename, info)
37
                 filename
       IN
                                              name of file to delete (string)
38
39
       IN
                 info
                                              info object (handle)
40
41
     C binding
42
     int MPI_File_delete(const char *filename, MPI_Info info)
43
     F08 binding
44
     MPI_File_delete(filename, info, ierror)
45
46
          CHARACTER(LEN=*), INTENT(IN) :: filename
47
          TYPE(MPI_Info), INTENT(IN) :: info
48
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

```
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
CHARACTER*(*) FILENAME
INTEGER INFO, IERROR
```

MPI_FILE_DELETE deletes the file identified by the file name filename. If the file does not exist, MPI_FILE_DELETE raises an error in the class MPI_ERR_NO_SUCH_FILE.

The info argument can be used to provide information regarding file system specifics (see Section 13.2.8). The constant MPI_INFO_NULL refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class MPI_ERR_FILE_IN_USE or MPI_ERR_ACCESS will be raised. Errors are raised using the default error handler (see Section 13.7).

13.2.4 Resizing a File

MPI_FILE_SET_SIZE(fh, size)

INOUT	fh	file handle (handle)
IN	size	size to truncate or expand file (integer)

C binding

int MPI_File_set_size(MPI_File fh, MPI_Offset size)

F08 binding

```
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
```

F binding

```
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

Unofficial Draft for Comment Only

 $\mathbf{2}$

```
1
      pointer. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is
\mathbf{2}
      erroneous to call this routine.
3
4
           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
5
           to seeking beyond the current end of file. (End of advice to users.)
6
7
          All nonblocking requests and split collective operations on fh must be completed before
8
      calling MPI_FILE_SET_SIZE. Otherwise, calling MPI_FILE_SET_SIZE is erroneous. As far
9
      as consistency semantics are concerned, MPI_FILE_SET_SIZE is a write operation that
10
      conflicts with operations that access bytes at displacements between the old and new file
11
     sizes (see Section 13.6.1).
12
13
      13.2.5 Preallocating Space for a File
14
15
16
17
     MPI_FILE_PREALLOCATE(fh, size)
18
        INOUT
                  fh
                                               file handle (handle)
19
        IN
                                               size to preallocate file (integer)
                  size
20
21
22
     C binding
23
     int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
^{24}
      F08 binding
25
     MPI_File_preallocate(fh, size, ierror)
26
          TYPE(MPI_File), INTENT(IN) ::
                                               fh
27
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::
                                                                 size
28
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                    ierror
29
30
      F binding
^{31}
     MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
32
          INTEGER FH, IERROR
33
          INTEGER(KIND=MPI_OFFSET_KIND) SIZE
34
          MPI_FILE_PREALLOCATE ensures that storage space is allocated for the first size bytes
35
      of the file associated with fh. MPI_FILE_PREALLOCATE is collective; all processes in the
36
      group must pass identical values for size. Regions of the file that have previously been
37
      written are unaffected. For newly allocated regions of the file, MPI_FILE_PREALLOCATE
38
      has the same effect as writing undefined data. If size is larger than the current file size, the
39
      file size increases to size. If size is less than or equal to the current file size, the file size is
40
      unchanged.
41
          The treatment of file pointers, pending nonblocking accesses, and file consistency is the
42
      same as with MPI_FILE_SET_SIZE. If MPI_MODE_SEQUENTIAL mode was specified when
43
      the file was opened, it is erroneous to call this routine.
44
45
           Advice to users. In some implementations, file preallocation may be expensive. (End
46
```

freeing group.

13.2.6 Querying the Size of a File 1 $\mathbf{2}$ 3 4 MPI_FILE_GET_SIZE(fh, size) 5 IN fh file handle (handle) 6 OUT size of the file in bytes (integer) size 7 8 9 C binding 10 int MPI_File_get_size(MPI_File fh, MPI_Offset *size) 11 F08 binding 12MPI_File_get_size(fh, size, ierror) 13 TYPE(MPI_File), INTENT(IN) :: fh 14INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617 F binding 18 MPI_FILE_GET_SIZE(FH, SIZE, IERROR) 19 INTEGER FH, IERROR 20INTEGER(KIND=MPI_OFFSET_KIND) SIZE 21MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with 22 the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a 23data access operation (see Section 13.6.1). 242513.2.7 Querying File Parameters 262728 MPI_FILE_GET_GROUP(fh, group) 29 30 IN fh file handle (handle) 31OUT group which opened the file (handle) group 32 33 C binding 34 int MPI_File_get_group(MPI_File fh, MPI_Group *group) 35 36 F08 binding 37 MPI_File_get_group(fh, group, ierror) 38 TYPE(MPI_File), INTENT(IN) :: fh 39 TYPE(MPI_Group), INTENT(OUT) :: group 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42F binding MPI_FILE_GET_GROUP(FH, GROUP, IERROR) 43 44INTEGER FH, GROUP, IERROR 45MPI_FILE_GET_GROUP returns a duplicate of the group of the communicator used to 46open the file associated with fh. The group is returned in group. The user is responsible for 47

```
1
     MPI_FILE_GET_AMODE(fh, amode)
\mathbf{2}
       IN
                fh
                                            file handle (handle)
3
       OUT
                amode
                                            file access mode used to open the file (integer)
4
5
6
     C binding
7
     int MPI_File_get_amode(MPI_File fh, int *amode)
8
     F08 binding
9
     MPI_File_get_amode(fh, amode, ierror)
10
         TYPE(MPI_File), INTENT(IN) ::
                                            fh
11
         INTEGER, INTENT(OUT) :: amode
12
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
13
14
     F binding
15
     MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
16
         INTEGER FH, AMODE, IERROR
17
         MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with
18
     fh.
19
20
     Example 13.1 In Fortran 77, decoding an amode bit vector will require a routine such as
21
     the following:
22
23
            SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
^{24}
     !
25
     !
         TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
26
     !
         IF SET, RETURN 1 IN BIT_FOUND, O OTHERWISE
27
     !
28
            INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
29
            BIT_FOUND = 0
30
            CP_AMODE = AMODE
31
      100 CONTINUE
32
            LBIT = 0
33
            HIFOUND = 0
34
            DO 20 L = MAX_BIT, 0, -1
35
               MATCHER = 2**L
36
               IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
37
                  HIFOUND = 1
38
                  LBIT = MATCHER
39
                   CP_AMODE = CP_AMODE - MATCHER
40
               END IF
^{41}
       20 CONTINUE
42
            IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
43
            IF (BIT_FOUND .EQ. O .AND. HIFOUND .EQ. 1 .AND. &
44
                CP_AMODE .GT. 0) GO TO 100
45
            END
46
47
```

This routine could be called successively to decode amode, one bit at a time. For example, the following code fragment would check for MPI_MODE_RDONLY.

CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
IF (BIT_FOUND .EQ. 1) THEN
PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
ELSE
PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
END IF

13.2.8 File Info

Hints specified via info (see Chapter 9) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_FILE_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per file basis, in MPI_FILE_OPEN, MPI_FILE_DELETE, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

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

MPI_FILE_SET_INFO(fh, info)			
INOUT	fh	file handle (handle)	
IN	info	info object (handle)	

C binding

int MPI_File_set_info(MPI_File fh, MPI_Info info)

```
F08 binding
```

```
MPI_File_set_info(fh, info, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

MPI_FILE_SET_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR

MPI_FILE_SET_INFO updates the hints of the file associated with fh using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not 48

 24

1 specified by info. It also has no effect on previously set or defaulted hints that are specified $\mathbf{2}$ by info, but are ignored by the MPI implementation in this call to MPI_FILE_SET_INFO. 3 MPI_FILE_SET_INFO is a collective routine. The info object may be different on each 4 process, but any info entries that an implementation requires to be the same on all processes $\mathbf{5}$ must appear with the same value in each process's info object. 6 Advice to users. Many info items that an implementation can use when it creates or 7opens a file cannot easily be changed once the file has been created or opened. Thus, 8 an implementation may ignore hints issued in this call that it would have accepted in 9 an open call. An implementation may also be unable to update certain info hints in a 10 call to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO. MPI_FILE_GET_INFO can be 11 used to determine whether info changes were ignored by the implementation. (End of 12advice to users.) 13 141516MPI_FILE_GET_INFO(fh, info_used) 17IN fh file handle (handle) 18 19OUT info_used new info object (handle) 2021C binding 22int MPI_File_get_info(MPI_File fh, MPI_Info *info_used) 23F08 binding 24 MPI_File_get_info(fh, info_used, ierror) 2526TYPE(MPI_File), INTENT(IN) :: fh TYPE(MPI_Info), INTENT(OUT) :: 27info_used INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2829F binding 30 MPI_FILE_GET_INFO(FH, INFO_USED, IERROR) 31 INTEGER FH, INFO_USED, IERROR 32 33 MPI_FILE_GET_INFO returns a new info object containing the hints of the file associ-34ated with fh. The current setting of all hints related to this file is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementa-35 36 tion and have default values specified; any user-supplied hints that were not ignored by the 37 implementation; and any additional hints that were set by the implementation. If no such 38hints exist, a handle to a newly created info object is returned that contains no key/value 39 pairs. The user is responsible for freeing info_used via MPI_INFO_FREE. 40 41 Reserved File Hints 42Some potentially useful hints (info key values) are outlined below. The following key values 43 are reserved. An implementation is not required to interpret these key values, but if it does 44 interpret the key value, it must provide the functionality described. (For more details on 45"info," see Chapter 9.) 46

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.
- collective_buffering (boolean) [SAME]: This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Valid values for this key are true and false. Collective buffering parameters are further directed via additional hints: cb_block_size, cb_buffer_size, and cb_nodes.
- cb_block_size (integer) [SAME]: This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (cyclic) pattern.
- cb_buffer_size (integer) [SAME]: This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of cb_block_size.
- cb_nodes (integer) [SAME]: This hint specifies the number of target nodes to be used for collective buffering.
- chunked (comma separated list of integers) [SAME]: This hint specifies that the file consists of a multidimentional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).
- chunked_item (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.
- chunked_size (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.
- filename (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI_FILE_GET_INFO. This key is ignored when passed to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, MPI_FILE_SET_INFO, and MPI_FILE_DELETE.
- file_perm (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI_FILE_OPEN with an amode that includes MPI_MODE_CREATE. The set of valid values for this key is implementation dependent.

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 2 3	I/O d	. –	strings) [SAME]: This hint specifies the list of store the file. This hint is most relevant when the			
4 5 6 7	typica	- /	specifies the number of parallel processes that will ams that access this file. This hint is most relevant			
8 9 10		es (integer) [SAME]: This m. This hint is most relevant	hint specifies the number of I/O devices in the when the file is created.			
11 12			hint specifies the number of I/O devices that the s relevant only when the file is created.			
13 14 15 16 17 18	for the device	is file. The striping unit is the before progressing to the next	int specifies the suggested striping unit to be used an amount of consecutive data assigned to one I/O at device, when striping across a number of devices. is relevant only when the file is created.			
19 20 21	13.3 Fil	e Views				
22 23	MPI_FILE_	SET_VIEW(fh, disp, etype, file	etype, datarep, info)			
24	INOUT	fh	file handle (handle)			
25	IN	disp	displacement (integer)			
26	IN	etype	elementary datatype (handle)			
27 28	IN	filetype	filetype (handle)			
29	IN	datarep	data representation (string)			
30	IN	info	info object (handle)			
31 32	IIN	IIIIO	nno object (nandie)			
33	C binding					
34 35		ile_set_view(MPI_File fh,	, MPI_Offset disp, MPI_Datatype etype, e, const char *datarep, MPI_Info info)			
36	F08 bindi	ng				
37 38	MPI_File_a	<pre>set_view(fh, disp, etype,</pre>	, filetype, datarep, info, ierror)			
39		TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp				
40		ER(KIND=MPI_OFFSET_KIND), MPI_Datatype), INTENT(IN)	-			
41		CTER(LEN=*), INTENT(IN) :	VI VI			
42		MPI_Info), INTENT(IN) ::	-			
43 44	INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror			
45	F binding					
46	-		, FILETYPE, DATAREP, INFO, IERROR)			
47		ER FH, ETYPE, FILETYPE, I				
48	CHARA	CTER*(*) DATAREP				

INTEGER(KIND=MPI_OFFSET_KIND) DISP

The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file. The start of the view is set to disp; the type of data is set to etype; the distribution of data to processes is set to filetype; and the representation of data in the file is set to datarep. In addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file pointer to zero. MPI_FILE_SET_VIEW is collective; the values for datarep and the extents of etype in the file data representation must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be committed.

The etype always specifies the data layout in the file. If etype is a portable datatype (see Section 2.4), the extent of etype is computed by scaling any displacements in the datatype to match the file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section 13.5.1 for further details.

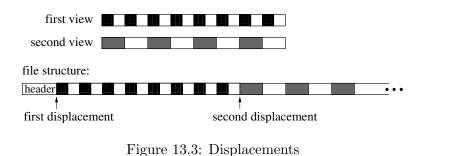
If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 13.3). Separate views, each using a different displacement and filetype, can be used to access each segment.



(End of advice to users.)

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

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $46 \\ 47$

,

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.
```

```
30
31
```

32

29

MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)

33	IN	fh	file handle (handle)
34	OUT	disp	displacement (integer)
35 36	OUT	etype	elementary datatype (handle)
37	OUT	filetype	filetype (handle)
38 39	OUT	datarep	data representation (string)
40 41 42 43	C bindin int MPI_	0	n, MPI_Offset *disp, MPI_Datatype *etype ype, char *datarep)
44	F08 bind	ling	
45	MPI_File	_get_view(fh, disp, etype	e, filetype, datarep, ierror)
46	TYPE	(MPI_File), INTENT(IN) ::	: fh
47	INTE	GER(KIND=MPI_OFFSET_KIND)), INTENT(OUT) :: disp
48	TYPE	(MPI_Datatype), INTENT(OU	JT) :: etype, filetype

5 6

7

8 9

10

11

12

```
CHARACTER(LEN=*), INTENT(OUT) :: datarep
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
INTEGER FH, ETYPE, FILETYPE, IERROR
CHARACTER*(*) DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

MPI_FILE_GET_VIEW returns the process's view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI_FILE_GET_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

13.4 Data Access

13.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 13.1.

positioning	synchronism	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

POSIX read()/fread() and write()/fwrite() are blocking, noncollective operations and use individual file pointers. The MPI equivalents are MPI_FILE_READ and

Unofficial Draft for Comment Only

 $\mathbf{2}$

 24

¹ MPI_FILE_WRITE.

Implementations of data access routines may buffer data to improve performance. This
 does not affect reads, as the data is always available in the user's buffer after a read operation
 completes. For writes, however, the MPI_FILE_SYNC routine provides the only guarantee
 that data has been transferred to the storage device.

⁷ Positioning

6

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,

27 28

29

30 31

32

33

34

37

38

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.

35 36 Synchronism

MPI supports blocking and nonblocking I/O routines.

A blocking I/O call will not return until the I/O request is completed.

 $new_file_offset = old_file_offset + \frac{elements(datatype)}{elements(etype)} \times count$

A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete.
 Given suitable hardware, this allows the transfer of data out of and into the user's buffer
 to proceed concurrently with computation. A separate request complete call (MPI_WAIT,
 MPI_TEST, or any of their variants) is needed to complete the I/O request, i.e., to confirm
 that the data has been read or written and that it is safe for the user to reuse the buffer.
 The nonblocking versions of the routines are named MPI_FILE_IXXX, where the I stands
 for immediate.

It is erroneous to access the local buffer of a nonblocking data access operation, or to
 use that buffer as the source or target of other communications, between the initiation and
 completion of the operation.

The split collective routines support a restricted form of "nonblocking" operations for collective data access (see Section 13.4.5).

Coordination

Every noncollective data access routine MPI_FILE_XXX has a collective counterpart. For most routines, this counterpart is MPI_FILE_XXX_ALL or a pair of MPI_FILE_XXX_BEGIN and MPI_FILE_XXX_END. The counterparts to the MPI_FILE_XXX_SHARED routines are MPI_FILE_XXX_ORDERED.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 13.6.4 for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, fh. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: buf, count, and datatype. Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass offset as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 and Section 4.1.11. The data is accessed from those parts of the file specified by the current view (Section 13.3). The type signature of datatype must match the type signature of some number of contiguous copies of the etype of the current view. As in a receive, it is erroneous to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI_TEST, MPI_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10–17.1.20. (End of advice to users.)

For blocking routines, status is returned directly. For nonblocking routines and split collective routines, status is returned when the operation is completed. The number of datatype entries and predefined elements accessed by the calling process can be extracted from status by using MPI_GET_COUNT and MPI_GET_ELEMENTS (or

Unofficial Draft for Comment Only

 $\overline{7}$

 24

 31

 $45 \\ 46$

1 2 3 4 5 6 7 8 9 10 11	same as for returns MF in the state passed to fields of sta When read is less The amount	or other operations — norma PI_ERR_IN_STATUS. The user us argument if the return valu MPI_TEST_CANCELLED to d atus are undefined. reading, a program can detect s than the amount requested.	The interpretation of the MPI_ERROR field is the ally undefined, but meaningful if an MPI routine can pass (in C and Fortran) MPI_STATUS_IGNORE e of this argument is not needed. The status can be letermine if the operation was cancelled. All other ct the end of file by noting that the amount of data Writing past the end of file increases the file size. e amount requested, unless an error is raised (or a
12 13	13.4.2 D	ata Access with Explicit Offs	ets
14 15 16		DE_SEQUENTIAL mode was sputines in this section.	becified when the file was opened, it is erroneous to
17 18	MPI_FILE	_READ_AT(fh, offset, buf, cou	nt, datatype, status)
19	IN	fh	file handle (handle)
20	IN	offset	file offset (integer)
21 22	OUT	buf	initial address of buffer (choice)
23	IN	count	number of elements in buffer (integer)
24	IN	datatype	datatype of each buffer element (handle)
25 26	OUT	status	status object (Status)
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46	F08 bind MPI_File_ TYPE(INTEG TYPE(INTEG TYPE(INTEG F binding MPI_FILE_ <type INTEG</type 	<pre>File_read_at(MPI_File fh,</pre>	, INTENT(IN) :: offset uf) :: datatype) :: ierror COUNT, DATATYPE, STATUS, IERROR) STATUS(MPI_STATUS_SIZE), IERROR OFFSET
47 48	IVI#1_1	ILL_READ_AT reads a file be	eginning at the position specified by offset.

<pre>N fh file handle (handle) N offset file offset (integer) OUT buf initial address of buffer (choice) N count mumber of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	<pre>N fh file handle (handle) N offset file offset (integer) OUT buf initial address of buffer (choice) N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	MPL FILF	READ_AT_ALL(fh, offset, buf,	count datatype status)	1
<pre>N offset file offset (integer) OUT buf initial address of buffer (choice) N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	<pre>N offset file offset (integer) OUT buf initial address of buffer (cholec) N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>		× ×	,	2
OUTbufinitial address of buffer (choice)INcountnumber of elements in buffer (integer)INdatatypedatatype of each buffer element (handle)OUTstatusstatus object (Status)OUTstatusstatus object (Status)C bindingint count, MPI_Datatype datatype, MPI_Status *status)F08 bindingMPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)TYPE(MPI_File), INTENT(IN) :: fhINTEORER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offsetTYPE(MPI_Datatype), INTENT(IN) :: datatypeTYPE(MPI_Status) :: statusINTEGER, OPTIONAL, INTENT(OUT) :: ierrorFF bindingMPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> LUP(*)INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERRORINTEGER (KIND=MPI_OFFSET_KIND) OFFSETMPI_FILE_READ_AT_ALLL is a collective version of the blocking MPI_FILE_READ_ATMPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_ATinterface.MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_ATINOUTfhfile handle (handle)INbufinitial address of buffer (choice)INcountnumber of elements in buffer (integer)INdatatypedatatype of each buffer (choice)INcountnumber of elements in buffer (integer)INdatatypeOUTstatusstatusstatus object (Status)CbuffOUTsta</type>	OUT buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status) FO8 binding MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(MPI_File, INTENT(IN) :: count INTEGER, INTENT(IN) :: count TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUWT, DATATYPE, STATUS, IERROR INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER FH, COUNT, DATATYPE, STATUS (MPI_STATUS_SIZE), IERROR INTEGER FH, COUNT, ATATYPE, STATUS (MPI_STATUS_SIZE), IERROR INNOUT fh file handle (handle) IN IN offset file offset (integer) IN IN upf initial address of buffer (choice) IN IN offset file offset (integer) IN N				3
IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) OUT status status object (Status) C binding int count, MPI_Datatype datatype, MPI_Status *status) FO8 binding MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER (KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(MPI_Datatype), INTENT(IN) :: duf intreget, intremt(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype intreget, intremt(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: datatype intreget, oPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) ctype> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR interface. MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface. MPI_FILE_READ_AT_(fh, offset, buf, count, datatype, status) intotal address of buffer (choice) in IN offset file offset (integer) in in IN count number of elements in buffer (integer) i	<pre>N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>				
<pre>N datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	<pre>N datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>				
OUT status status object (Status) C binding int count, MPI_Pile fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh file INTEGER, (KIND-MPI_OFFSET_KIND), INTENT(IN) :: offset file TYPE(*), DIMENSION() :: buf file INTEGER, INTENT(IN) :: count file TYPE(MPI_Datatype), INTENT(IN) :: datatype file TYPE(MPI_Status) :: status intreger, opTIONAL, INTENT(OUT) :: ierror F binding mPI_FILE_READ_AT_ALL(FH, OFFSET, EUF, COUNT, DATATYPE, STATUS, IERROR INTEGER, H, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR file INTEGER (KIND=MPI_OFFSET_KIND) OFFSET mPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer clement (handle) OUT status status object (Status)	OUT status status object (Status) C binding int count, MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding met count, MPI_File, NorseT_KIND), INTENT(IN) :: offset TYPE(MPI_File), INTENT(IN) :: buf met count TYPE(MPI_Datatype), INTENT(IN) :: datatype met count TYPE(MPI_Status) :: status met count INTEGER, OPTIONAL, INTENT(OUT) :: ierror file F binding met count, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR NPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR met count, INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL (FH, OFFSet, Status (MPI_STATUS_SIZE), IERROR met count, MPI_FILE_READ_AT_ALL (FH, offset, buf, count, datatype, status) INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT INN buf mitial address of buffer (choice) met count IN woffset file				7
C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER FH, COUNT, DATATYPE, STATUS(MPI_FILE_READ_AT interface. MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface. MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_FILe_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 45</type>	C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(MPI_Datatype), INTENT(IN) :: offset TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) INTEGER H, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface. MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_FILE_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding</type>	IN	datatype	datatype of each buffer element (handle)	8
C binding 11 int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface. MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) INOUT fh</type>	C binding int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER,(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface. MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) INOUT IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer choice) IN datatype otatus status status Status Status Status Status Status Status Status Status</type>	OUT	status	status object (Status)	
<pre>C Dinning int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	<pre>C Dinning int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>				
<pre>int count, MPI_Datatype datatype, MPI_Status *status) FOS binding MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface. MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 45 </type></pre>	<pre>int count, MPI_Datatype datatype, MPI_Status *status) F08 binding MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface. MPI_FILE_VRRITE_AT(fh, offset, buf, court, datatype, status) INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) F08 binding</type></pre>			fb MDT Offerst offerst model which	
MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) 11 TYPE(MPI_File), INTENT(IN) :: fh 11 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 11 TYPE(*), DIMENSION() :: buf 11 INTEGER, INTENT(IN) :: count 12 TYPE(MPI_Datatype), INTENT(IN) :: datatype 12 TYPE(MPI_Status) :: status 12 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23 F binding 24 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 24 (type> BUF(*)) 1NTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 27 INTEGER KI, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 27 INTEGER KIND=OFFSET_KIND) OFFSET 28 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 31 interface. 32 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 33 INOUT fh file offset (integer) 34 IN buf initial address of buffer (choice) 34 IN count number of elements in buffer (integer) 35 IN datatype datatype of each buffer eleme	F08 binding 11 MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) 11 TYPE(MPI_File), INTENT(IN) :: fh 11 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 15 TYPE(*), DIMENSION() :: buf 16 INTEGER, INTENT(IN) :: count 17 TYPE(MPI_Datatype), INTENT(IN) :: datatype 17 TYPE(MPI_Status) :: status 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 F binding 18 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 20 <type> BUF(*) 10 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 11 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 11 INTEGER KIND=MPI_OFFSET_KIND) OFFSET 11 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 33 INOUT fh file handle (handle) 34 IN buf initial address of buffer (choice) 34 IN count number of elements in buffer (integer) 34 IN buf initial address of buffer (choice) 35 IN datatype</type>	int MPI_F			
<pre>MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <pre></pre></pre>	<pre>MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <pre></pre></pre>	T ee 14 4		pe auturype, in i_boards (boards)	14
TYPE(MPI_File), INTENT(IN) :: fh 17 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 18 TYPE(*), DIMENSION() :: buf 19 INTEGER, INTENT(IN) :: count 20 TYPE(MPI_Datatype), INTENT(IN) :: datatype 21 TYPE(MPI_Datatype), INTENT(UN) :: datatype 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 F binding 24 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 25 (type> BUF(*) 26 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 27 INTEGER (KIND=MPI_OFFSET_KIND) OFFSET 28 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 34 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 INOUT fh file handle (handle) 35 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN count number of elements in buffer (integer) 36 IN count number of elements in buffer (integer) 37 IN datatype datatype of each buffer element (handle) 41 OUT status status object (Status) 42 C binding 43 int MPI_FILe_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 45	TYPE(MPI_File), INTENT(IN) :: fh INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) 44 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 77 INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 78 interface. 78 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 78 IN offset file offset (integer) IN buf initial address of buffer (choice) IN datatype datatype of each buffer (integer) IN datatype datatype of each buffer (integer) IN datatype datatype of each buffer (integer) IN datatype status object (Status) C binding 74 int count, MPI_Datatype datatype, MPI_Status *status) F08 binding 74</type>		0		15
<pre>INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	<pre>INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset TYPE(*), DIMENSION() :: buf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>			V 1	
TYPE(*), DIMENSION() :: buf 19 INTEGER, INTENT(IN) :: count 20 TYPE(MPI_Datatype), INTENT(IN) :: datatype 21 TYPE(MPI_Status) :: status 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23 F binding 24 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 26 <type> BUF(*) 27 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 27 INTEGER (KIND=MPI_OFFSET_KIND) OFFSET 28 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 31 interface. 33 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 34 INOUT fh file handle (handle) IN offset file offset (integer) 36 IN out number of elements in buffer (integer) 36 IN datatype datatype of each buffer element (handle) 36 OUT status status object (Status) 32 The MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 44</type>	<pre>TYPE(*), DIMENSION() :: buf</pre>		-		
<pre>TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>	<pre>TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>				
TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER (KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface. MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 45</type>	<pre>TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>		•		20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror 24 PF binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <pre></pre>	INTEGER, OPTIONAL, INTENT(OUT) :: ierror F binding MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) <type> BUF(*) 26 INTEGER FH, COUNT, DATATYPE, STATUS_SIZE), IERROR 27 INTEGER (KIND=MPI_OFFSET_KIND) OFFSET 28 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_WRITE_AT(fh, offset, buf, court, datatype, status) 33 INOUT fh file offset (integer) IN offset file offset (integer) IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) C binding 44 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding 47</type>		01	:: datatype	21
F binding 24 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 25 <type> BUF(*) 26 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 27 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 28 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_WRITE_AT(fh, offset, buf, court, datatype, status) 33 INOUT fh file handle (handle) IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 38 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C binding 44 44 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 44</type>	F binding 44 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 56 <type> BUF(*) 77 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 77 INTEGER (KIND=MPI_OFFSET_KIND) OFFSET 78 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 78 interface. 71 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 73 INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) C binding 11 int count, MPI_Datatype datatype, MPI_Status *status) 44 F08 binding 44</type>			·· jerror	
F binding 25 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 26 <type> BUF(*) 27 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 28 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 34 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 38 IN datatype datatype of each buffer element (handle) 34 OUT status status object (Status) 44 MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 44</type>	P binding 25 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 26 <type> BUF(*) 26 INTEGER FH, COUNT, DATATYPE, STATUS (MPI_STATUS_SIZE), IERROR 27 INTEGER (KIND=MPI_OFFSET_KIND) OFFSET 28 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 33 INOUT fh file handle (handle) IN offset file offset (integer) IN buf initial address of buffer (choice) IN count number of elements in buffer (integer) IN datatype datatype of each buffer element (handle) OUT status status object (Status) C binding 44 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) 44 F08 binding 47</type>				
<type> BUF(*)26INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR27INTEGER(KIND=MPI_OFFSET_KIND) OFFSET29MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT30interface.31MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)33INOUTfhfile handle (handle)INoffsetfile offset (integer)INbufinitial address of buffer (choice)INbufnumber of elements in buffer (integer)INdatatypedatatype of each buffer element (handle)OUTstatusstatus object (Status)Cbinditint MPI_File_write_at (MPI_File fh, MPI_Offset offset, const void *buf,34</type>	<type> BUF(*)26INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR27INTEGER(KIND=MPI_OFFSET_KIND) OFFSET28MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT30interface.31MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)33INOUTfhfile handle (handle)INoffsetfile offset (integer)INbufinitial address of buffer (choice)INcountnumber of elements in buffer (integer)INdatatypedatatype of each buffer element (handle)OUTstatusstatus object (Status)Cbinding44int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)44F08 binding47</type>				
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 27 INTEGER (KIND=MPI_OFFSET_KIND) OFFSET 28 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_WRITE_AT(fh, offset, buf, court, datatype, status) 33 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 IN muther of elements in buffer (integer) 41 IN datatype datatype of each buffer element (handle) 42 OUT status status object (Status) 42 C binding 43 44 int MPI_File_ite_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 43	INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface. MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) NOUT fh file handle (handle) N offset file offset (integer) N buf initial address of buffer (choice) N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding			OF, COUNI, DAIAIIPE, SIAIOS, IERROR)	26
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 29 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 33 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 38 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C binding int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 43	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 29 MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 33 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C binding int count, MPI_Datatype datatype, MPI_Status *status) 44 F08 binding 47 47	• -		TATUS(MPI_STATUS_SIZE), IERROR	27
MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_VRITE_AT(fh, offset, buf, court, datatype, status) 33 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 38 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 31 OUT status status object (Status) 41 Att MPI_EILE_WRITE_at(MPI_FILE fh, MPI_Offset offset, const void *buf, 43	MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT 30 interface. 31 MPI_FILE_WRITE_AT(fh, offset, buf, court, datatype, status) 32 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 38 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 44 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) 44 F08 bindits K 44 45	INTEG	ER(KIND=MPI_OFFSET_KIND)	OFFSET	
interface.	interface. 31 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) NOUT fh file handle (handle) N offset file offset (integer) N buf initial address of buffer (choice) N count number of elements in buffer (integer) N datatype datatype of each buffer element (handle) OUT status status object (Status) C binding int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype, MPI_Status *status) F08 binding IN terms and terms	MPI F	ILE READ AT ALL is a colle	ctive version of the blocking MPI FILE READ AT	
NPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 33 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 41 OUT status status object (Status) 42 ***********************************	MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 33 INOUT fh file handle (handle) 34 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 Z binding int count, MPI_Datatype datatype, MPI_Status *status) 43 F08 bindits F08 bindits 44 47			· · · · · · · · · · · · · · · · · · ·	
MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 34 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C bint MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 43	MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status) 34 INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (choice) 38 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C binding true count, MPI_Datatype datatype, MPI_Status *status) 44 F08 binding 47				32
INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 L 43 44 44 IN MPI_File_th, MPI_Offset offset, const void *buf, 45	INOUT fh file handle (handle) 35 IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C binding true count, MPI_Datatype datatype, MPI_Status *status) 43 F08 binding 47		WRITE AT(fh_offset_buf_cou	nt datatype status)	33
INoffsetfile offset (integer)36INbufinitial address of buffer (choice)37INcountnumber of elements in buffer (integer)39INdatatypedatatype of each buffer element (handle)40OUTstatusstatus object (Status)4142434444INHPI_File_th, MPI_Offset offset, const void *buf,45	IN offset file offset (integer) 36 IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (choice) 38 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 42 C binding int count, MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) 44 F08 binding 47		,	··· ,	
IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 L 41 42 43 C binding true_at(MPI_File fh, MPI_Offset offset, const void *buf, 43	IN buf initial address of buffer (choice) 37 IN count number of elements in buffer (choice) 38 IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 OUT status status object (Status) 42 C binding int count, MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) 43 F08 binding 47				
IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C binding 43 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 45	IN count number of elements in buffer (integer) 39 IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C binding 43 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) 43 F08 binding 47				
<pre>IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C binding int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 45</pre>	IN datatype datatype of each buffer element (handle) 40 OUT status status object (Status) 41 C binding 42 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) 44 F08 binding 47	IN	but	initial address of buffer (choice)	38
OUT status status object (Status) 41 C binding 43 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 44	OUT status status object (Status) 41 OUT status 42 C binding 43 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) 44 F08 binding 47	IN	count	number of elements in buffer (integer)	39
OUT status status object (Status) 42 C binding 43 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 44	OUT status status object (Status) 42 C binding 43 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 44 int count, MPI_Datatype datatype, MPI_Status *status) 46 F08 binding 47	IN	datatype	datatype of each buffer element (handle)	
C binding 43 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 45	C binding int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status) F08 binding	OUT	status	status object (Status)	
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 44	<pre>int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>				
45	int count, MPI_Datatype datatype, MPI_Status *status) 43 46 F08 binding 47				44
int count MDT Dotations datations MDT Status tetatic)	F08 binding	·			
40	r os binding		-	pe uararype, mrījotatus *stātus)	
F08 binding 47			-		
-	MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) 48	WLT_LITe_A			

```
1
         TYPE(MPI_File), INTENT(IN) :: fh
\mathbf{2}
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
3
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
4
         INTEGER, INTENT(IN) :: count
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         TYPE(MPI_Status) :: status
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     F binding
9
     MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
10
         <type> BUF(*)
11
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
12
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
13
14
         MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset.
15
16
     MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)
17
18
       INOUT
                fh
                                           file handle (handle)
19
       IN
                offset
                                           file offset (integer)
20
                                           initial address of buffer (choice)
       IN
                buf
21
22
       IN
                count
                                           number of elements in buffer (integer)
23
       IN
                                           datatype of each buffer element (handle)
                datatype
24
       OUT
                status
                                           status object (Status)
25
26
27
     C binding
28
     int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
29
                    int count, MPI_Datatype datatype, MPI_Status *status)
30
     F08 binding
^{31}
     MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
32
         TYPE(MPI_File), INTENT(IN) :: fh
33
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
34
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
35
         INTEGER, INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Status) :: status
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     F binding
41
     MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
42
         <type> BUF(*)
43
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
44
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45
         MPI_FILE_WRITE_AT_ALL is a collective version of the blocking
46
     MPI_FILE_WRITE_AT interface.
47
48
```

MPI_FILE	_IREAD_AT(fh, offset, buf, cou	int, datatype, request)	1	
IN	fh	file handle (handle)	2	
IN	offset	file offset (integer)	3 4	
OUT	buf	initial address of buffer (choice)	5	
IN	count	number of elements in buffer (integer)	6	
IN	datatype	datatype of each buffer element (handle)	7	
OUT	request	request object (handle)	8 9	
001	request	request object (nandie)	10	
C bindin	g		11	
int MPI_H	File_iread_at(MPI_File fh	, MPI_Offset offset, void *buf, int count,	12	
	MPI_Datatype datatyp	e, MPI_Request *request)	13 14	
F08 bind	ing		15	
		, count, datatype, request, ierror)	16	
	(MPI_File), INTENT(IN) ::		17	
	<pre>SER(KIND=MPI_OFFSET_KIND) (*), DIMENSION(), ASYNCI</pre>	-	18	
	ER, INTENT(IN) :: count		19 20	
TYPE	(MPI_Datatype), INTENT(IN)) :: datatype	21	
	(MPI_Request), INTENT(OUT)	-	22	
INTEC	ER, OPTIONAL, INTENT(OUT)) :: ierror	23	
F binding			24	
		, COUNT, DATATYPE, REQUEST, IERROR)	25 26	
• 1	>> BUF(*) GER FH, COUNT, DATATYPE, I	REQUEST TERROR	27	
	ER(KIND=MPI_OFFSET_KIND)		28	
			29	
	TILE_IREAD_AT IS a HOLIDIOCK	ing version of the MPI_FILE_READ_AT interface.	30 31	
			32	
MPI_FILE	_IREAD_AT_ALL(fh, offset, bu	f, count, datatype, request)	33	
IN	fh	file handle (handle)	34	
IN	offset	file offset (integer)	35	
OUT	buf	initial address of buffer (choice)	36 37	
IN	count	number of elements in buffer (integer)	38	
IN	datatype	datatype of each buffer element (handle)	39	
OUT	request	request object (handle)	40	
C bindin	g		42 43	
int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,				
<pre>int count, MPI_Datatype datatype, MPI_Request *request) 4</pre>				
F08 bind	ing		46	
MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)47TYPE(MPI_File), INTENT(IN) :: fh48				

```
1
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::
                                                            offset
\mathbf{2}
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
3
         INTEGER, INTENT(IN) :: count
4
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
5
         TYPE(MPI_Request), INTENT(OUT) ::
                                                request
6
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
7
     F binding
8
     MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
9
         <type> BUF(*)
10
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
11
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
12
13
         MPI_FILE_IREAD_AT_ALL is a nonblocking version of MPI_FILE_READ_AT_ALL. See
14
     Section 13.6.5 for semantics of nonblocking collective file operations.
15
16
     MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
17
18
       INOUT
                fh
                                            file handle (handle)
19
       IN
                offset
                                            file offset (integer)
20
                                            initial address of buffer (choice)
       IN
                buf
21
22
       IN
                count
                                            number of elements in buffer (integer)
23
       IN
                                            datatype of each buffer element (handle)
                datatype
24
       OUT
                request
                                            request object (handle)
25
26
27
     C binding
28
     int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
29
                    int count, MPI_Datatype datatype, MPI_Request *request)
30
     F08 binding
^{31}
     MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
32
         TYPE(MPI_File), INTENT(IN) :: fh
33
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
34
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
35
         INTEGER, INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Request), INTENT(OUT) ::
                                                request
38
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
39
40
     F binding
^{41}
     MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
42
         <type> BUF(*)
43
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
44
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
45
         MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.
46
47
48
```

MPI_FILE_IWRITE_AT_ALL(fh, offset, buf, count, datatype, request)			
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 9
OUT	request	request object (handle)	10
C binding	ŷ		11
		e fh, MPI_Offset offset, const void *buf,	12
	int count, MPI_Dataty	ype datatype, MPI_Request *request)	13
F08 bindi	ng		14 15
	0	buf, count, datatype, request, ierror)	16
	<pre>MPI_File), INTENT(IN) ::</pre>		17
	ER(KIND=MPI_OFFSET_KIND),		18
		C(IN), ASYNCHRONOUS :: buf	19
	ER, INTENT(IN) :: count MPI_Datatype), INTENT(IN)	·· datatype	20 21
	MPI_Request), INTENT(OUT)	V -	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			23
F binding	,		24
		BUF, COUNT, DATATYPE, REQUEST, IERROR)	25
	> BUF(*)		26
	ER FH, COUNT, DATATYPE, F		27 28
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET			29
MPI_F	ILE_IWRITE_AT_ALL is a not	nblocking version of MPI_FILE_WRITE_AT_ALL.	30
			31
13.4.3 Da	ata Access with Individual Fil	e Pointers	32
MPI maint	ains one individual file point	er per process per file handle. The current value	33 34
	-	fiset in the data access routines described in this	34 35
		ate the individual file pointers maintained by MPI.	36
	l file pointer is not used nor u	*	37
The individual file pointer routines have the same semantics as the data access with explicit offset routines described in Section 13.4.2, with the following modification:			38
explicit on	set fournes described in seen	on 19.4.2, with the following modification.	39
		rrent value of the MPI-maintained individual file	40 41
point	er.		41
			43 44

to point to the next etype after the last one that will be accessed. The file pointer is updated relative to the current view of the file. If MPL MODE SEQUENTIAL mode was specified when the file was opened, it is erroneous

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.

Unofficial Draft for Comment Only

45

46

```
1
     MPI_FILE_READ(fh, buf, count, datatype, status)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       OUT
                buf
                                            initial address of buffer (choice)
4
5
       IN
                count
                                            number of elements in buffer (integer)
6
       IN
                                            datatype of each buffer element (handle)
                datatype
7
       OUT
                                            status object (Status)
                status
8
9
     C binding
10
     int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
11
                    MPI_Status *status)
12
13
     F08 binding
14
     MPI_File_read(fh, buf, count, datatype, status, ierror)
15
         TYPE(MPI_File), INTENT(IN) :: fh
16
         TYPE(*), DIMENSION(...)
                                    :: buf
17
         INTEGER, INTENT(IN) :: count
18
         TYPE(MPI_Datatype), INTENT(IN) ::
                                                datatype
19
         TYPE(MPI_Status) :: status
20
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
21
     F binding
22
23
     MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
^{24}
          <type> BUF(*)
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
25
26
         MPI_FILE_READ reads a file using the individual file pointer.
27
28
     Example 13.2 The following Fortran code fragment is an example of reading a file until
29
     the end of file is reached:
30
31
         Read a preexisting input file until all data has been read.
     !
32
         Call routine "process_input" if all requested data is read.
     !
33
     Т
         The Fortran 90 "exit" statement exits the loop.
34
35
                       bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
            integer
36
            parameter (bufsize=100)
37
                       localbuffer(bufsize)
            real
38
            integer (kind=MPI_OFFSET_KIND) zero
39
40
            zero = 0
41
42
            call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
43
                                 MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
44
            call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
45
                                     MPI_INFO_NULL, ierr)
46
            totprocessed = 0
47
            do
48
               call MPI_FILE_READ(myfh, localbuffer, bufsize, MPI_REAL, &
```

```
1
                               status, ierr)
                                                                                        \mathbf{2}
          call MPI_GET_COUNT(status, MPI_REAL, numread, ierr)
                                                                                        3
          call process_input(localbuffer, numread)
          totprocessed = totprocessed + numread
                                                                                        4
          if (numread < bufsize) exit
                                                                                        5
                                                                                        6
      enddo
                                                                                        7
                                                                                        8
      write(6,1001) numread, bufsize, totprocessed
1001 format("No more data: read", I3, "and expected", I3, &
                                                                                        9
                                                                                        10
              "Processed total of", I6, "before terminating job.")
                                                                                        11
      call MPI_FILE_CLOSE(myfh, ierr)
                                                                                        12
                                                                                        13
                                                                                        14
                                                                                        15
MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
                                                                                        16
  INOUT
           fh
                                      file handle (handle)
                                                                                        17
                                                                                        18
  OUT
           buf
                                      initial address of buffer (choice)
                                                                                        19
  IN
           count
                                      number of elements in buffer (integer)
                                                                                        20
                                      datatype of each buffer element (handle)
  IN
           datatype
                                                                                        21
                                                                                        22
  OUT
                                      status object (Status)
           status
                                                                                        23
                                                                                        ^{24}
C binding
                                                                                        25
int MPI_File_read_all(MPI_File fh, void *buf, int count,
                                                                                        26
              MPI_Datatype datatype, MPI_Status *status)
                                                                                        27
F08 binding
                                                                                        28
                                                                                        29
MPI_File_read_all(fh, buf, count, datatype, status, ierror)
                                                                                        30
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
                                                                                        31
    INTEGER, INTENT(IN) :: count
                                                                                        32
                                                                                        33
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                        34
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        35
                                                                                        36
F binding
                                                                                        37
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
                                                                                        38
    <type> BUF(*)
                                                                                        39
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                        40
                                                                                        41
    MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
                                                                                        42
                                                                                        43
                                                                                        44
                                                                                        45
                                                                                        46
```

```
1
     MPI_FILE_WRITE(fh, buf, count, datatype, status)
2
       INOUT
                 fh
                                             file handle (handle)
3
       IN
                 buf
                                             initial address of buffer (choice)
4
5
       IN
                 count
                                             number of elements in buffer (integer)
6
                 datatype
       IN
                                             datatype of each buffer element (handle)
7
       OUT
                 status
                                             status object (Status)
8
9
10
     C binding
11
     int MPI_File_write(MPI_File fh, const void *buf, int count,
                    MPI_Datatype datatype, MPI_Status *status)
12
13
     F08 binding
14
     MPI_File_write(fh, buf, count, datatype, status, ierror)
15
          TYPE(MPI_File), INTENT(IN) :: fh
16
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
17
          INTEGER, INTENT(IN) :: count
18
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
          TYPE(MPI_Status) :: status
20
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     F binding
23
     MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
^{24}
          <type> BUF(*)
25
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
26
          MPI_FILE_WRITE writes a file using the individual file pointer.
27
28
29
     MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
30
       INOUT
                 fh
                                             file handle (handle)
^{31}
       IN
                 buf
                                             initial address of buffer (choice)
32
33
       IN
                                             number of elements in buffer (integer)
                 count
34
       IN
                                             datatype of each buffer element (handle)
                 datatype
35
       OUT
                                             status object (Status)
                 status
36
37
38
     C binding
39
     int MPI_File_write_all(MPI_File fh, const void *buf, int count,
40
                    MPI_Datatype datatype, MPI_Status *status)
41
     F08 binding
42
     MPI_File_write_all(fh, buf, count, datatype, status, ierror)
43
          TYPE(MPI_File), INTENT(IN) :: fh
44
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
45
          INTEGER, INTENT(IN) :: count
46
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
          TYPE(MPI_Status) :: status
48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                        2
F binding
MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
                                                                                        4
    <type> BUF(*)
                                                                                        5
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
                                                                                        6
    MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
                                                                                        7
                                                                                        8
face.
                                                                                        9
                                                                                        10
MPI_FILE_IREAD(fh, buf, count, datatype, request)
                                                                                        11
  INOUT
                                                                                       12
           fh
                                      file handle (handle)
                                                                                       13
  OUT
           buf
                                      initial address of buffer (choice)
                                                                                       14
  IN
           count
                                      number of elements in buffer (integer)
                                                                                       15
                                                                                       16
  IN
           datatype
                                      datatype of each buffer element (handle)
                                                                                        17
  OUT
                                      request object (handle)
           request
                                                                                       18
                                                                                        19
C binding
                                                                                       20
int MPI_File_iread(MPI_File fh, void *buf, int count,
                                                                                       21
              MPI_Datatype datatype, MPI_Request *request)
                                                                                       22
                                                                                       23
F08 binding
                                                                                       24
MPI_File_iread(fh, buf, count, datatype, request, ierror)
                                                                                       25
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                        26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                       27
    INTEGER, INTENT(IN) :: count
                                                                                       28
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       29
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                       30
    INTEGER, OPTIONAL, INTENT(OUT) ::
                                           ierror
                                                                                       31
F binding
                                                                                       32
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
                                                                                       33
    <type> BUF(*)
                                                                                       34
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
                                                                                       35
                                                                                       36
    MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.
                                                                                       37
                                                                                       38
Example 13.3 The following Fortran code fragment illustrates file pointer update seman-
                                                                                       39
tics:
                                                                                        40
    Read the first twenty real words in a file into two local
                                                                                       41
I.
1
    buffers. Note that when the first MPI_FILE_IREAD returns,
                                                                                       42
    the file pointer has been updated to point to the
!
                                                                                       43
    eleventh real word in the file.
I.
                                                                                       44
                                                                                        45
                 bufsize, req1, req2
      integer
                                                                                        46
      integer, dimension(MPI_STATUS_SIZE) :: status1, status2
                                                                                        47
      parameter (bufsize=10)
                                                                                        48
```

```
1
            real
                       buf1(bufsize), buf2(bufsize)
\mathbf{2}
            integer (kind=MPI_OFFSET_KIND) zero
3
4
            zero = 0
5
            call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
6
                                 MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
7
            call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
8
                                     MPI_INFO_NULL, ierr)
9
            call MPI_FILE_IREAD(myfh, buf1, bufsize, MPI_REAL, &
10
                                  req1, ierr)
11
            call MPI_FILE_IREAD(myfh, buf2, bufsize, MPI_REAL, &
12
                                  req2, ierr)
13
14
            call MPI_WAIT(req1, status1, ierr)
15
            call MPI_WAIT(req2, status2, ierr)
16
17
            call MPI_FILE_CLOSE(myfh, ierr)
18
19
20
     MPI_FILE_IREAD_ALL(fh, buf, count, datatype, request)
21
22
       INOUT
                fh
                                            file handle (handle)
23
       OUT
                buf
                                            initial address of buffer (choice)
24
       IN
                                            number of elements in buffer (integer)
                count
25
26
       IN
                                            datatype of each buffer element (handle)
                datatype
27
       OUT
                request
                                            request object (handle)
28
29
     C binding
30
     int MPI_File_iread_all(MPI_File fh, void *buf, int count,
^{31}
                    MPI_Datatype datatype, MPI_Request *request)
32
33
     F08 binding
34
     MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
35
         TYPE(MPI_File), INTENT(IN) :: fh
36
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
37
         INTEGER, INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Request), INTENT(OUT) ::
                                                request
40
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
41
     F binding
42
     MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
43
          <type> BUF(*)
44
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
45
46
         MPI_FILE_IREAD_ALL is a nonblocking version of MPI_FILE_READ_ALL.
47
48
```

MPI_FILE_IWRITE(fh, buf, count, datatype, request)					
INOUT	fh	file handle (handle)	2 3		
IN	buf	initial address of buffer (choice)	4		
IN	count	number of elements in buffer (integer)	5		
IN	datatype	datatype of each buffer element (handle)	6		
OUT	request	request object (handle)	7 8		
		roqueer exject (name)	9		
C binding	r 5		10		
<pre>int MPI_File_iwrite(MPI_File fh, const void *buf, int count,</pre>					
	MPI_Datatype datatyp	e, MPI_Request *request)	12 13		
F08 binding					
		atatype, request, ierror)	15		
	<pre>MPI_File), INTENT(IN) :: *), DIMENSION(), INTEN</pre>	fh T(IN), ASYNCHRONOUS :: buf	16		
	ER, INTENT(IN) :: count		17 18		
	MPI_Datatype), INTENT(IN		18		
	MPI_Request), INTENT(OUT)	-	20		
INTEG	ER, OPTIONAL, INTENT(OUT)) :: ierror	21		
F binding			22		
	IWRITE(FH, BUF, CUUNT, D. > BUF(*)	ATATYPE, REQUEST, IERROR)	23 24		
• -	ER FH, COUNT, DATATYPE, 1	REQUEST, IERROR	25		
		version of the MPI_FILE_WRITE interface.	26		
1011 1_1			27		
			28 29		
	IWRITE_ALL(fh, buf, count, c		30		
INOUT	fh	file handle (handle)	31		
IN	buf	initial address of buffer (choice)	32		
IN	count	number of elements in buffer (integer)	33 34		
IN	datatype	datatype of each buffer element (handle)	35		
OUT	request	request object (handle)	36		
			37		
C binding			38 39		
int MPI_F		fh, const void *buf, int count, e, MPI_Request *request)	40		
		o, m 1_moquobo / toquobo,	41		
F08 bind	•	t datatune request ierror)	42 43		
<pre>MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror) TYPE(MPI_File), INTENT(IN) :: fh</pre>					
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf					
INTEGER, INTENT(IN) :: count					
TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Request), INTENT(OUT) :: request					
	········ · · · · · · · · · · · · · ·	, ±044000	48		

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     F binding
3
     MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
4
          <type> BUF(*)
5
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
6
7
          MPI_FILE_IWRITE_ALL is a nonblocking version of MPI_FILE_WRITE_ALL.
8
9
     MPI_FILE_SEEK(fh, offset, whence)
10
11
       INOUT
                 fh
                                              file handle (handle)
12
       IN
                 offset
                                              file offset (integer)
13
                                              update mode (state)
       IN
                 whence
14
15
16
     C binding
17
     int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
18
     F08 binding
19
     MPI_File_seek(fh, offset, whence, ierror)
20
          TYPE(MPI_File), INTENT(IN) :: fh
21
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
22
          INTEGER, INTENT(IN) :: whence
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
     F binding
26
     MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
27
          INTEGER FH, WHENCE, IERROR
28
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
29
          MPI_FILE_SEEK updates the individual file pointer according to whence, which has the
30
     following possible values:
^{31}
32
         • MPI_SEEK_SET: the pointer is set to offset
33
         • MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
34
35
         • MPI_SEEK_END: the pointer is set to the end of file plus offset
36
37
          The offset can be negative, which allows seeking backwards. It is erroneous to seek to
     a negative position in the view.
38
39
40
     MPI_FILE_GET_POSITION(fh, offset)
41
42
       IN
                 fh
                                              file handle (handle)
43
       OUT
                 offset
                                              offset of individual pointer (integer)
44
45
     C binding
46
     int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
47
48
     F08 binding
```

MPI_File_get_position(fh, offset, ierror)			
TYPE(MPI_File), INTENT(IN) :: fh			
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset			
IN	TEGER, OPTIONAL, I	NTENT(OUT) :: ierror	4
F bind	ling		5 6
	LE_GET_POSITION(FH	I, OFFSET, IERROR)	7
IN	INTEGER FH, IERROR		
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET			8 9
М	MPI_FILE_GET_POSITION returns, in offset, the current position of the individual file		
	pointer in etype units relative to the current view.		
pointer	in ctype units relativ		12
A	dvice to users. The	e offset can be used in a future call to MPI_FILE_SEEK using	13
v	$hence = MPI_SEEK_SE$	ET to return to the current position. To set the displacement to	14
t	he current file pointer	position, first convert offset into an absolute byte position using	15
		_OFFSET, then call MPI_FILE_SET_VIEW with the resulting	16
d	isplacement. (End of	advice to users.)	17
			18
			19
MPI_F	LE_GET_BYTE_OFF	SET(fh, offset, disp)	20 21
IN	fh	file handle (handle)	21
			23
IN	offset	offset (integer)	24
OUT	disp	absolute byte position of offset (integer)	25
			26
C bin	ding		27
int MF		fset(MPI_File fh, MPI_Offset offset,	28
	MPI_Offset	*disp)	29
F08 b	inding		30
MPI_Fi	le_get_byte_offset	(fh, offset, disp, ierror)	31
	PE(MPI_File), INTE	-	32
IN	TEGER(KIND=MPI_OFF	'SET_KIND), INTENT(IN) :: offset	33
IN	TEGER(KIND=MPI_OFF	'SET_KIND), INTENT(OUT) :: disp	34
IN	TEGER, OPTIONAL, I	NTENT(OUT) :: ierror	35 36
F bind	ling		37
	•	(FH, OFFSET, DISP, IERROR)	38
	TEGER FH, IERROR		39
IN	TEGER(KIND=MPI_OFF	SET_KIND) OFFSET, DISP	40
Γ.4		OFESET converts a view volative effect into an absolute bet	41
MPI_FILE_GET_BYTE_OFFSET converts a view-relative offset into an absolute byte position. The absolute byte position (from the beginning of the file) of offset relative to the			42
current view of fh is returned in disp .			43
current view of in is returned in disp.			

13.4.4 Data Access with Shared File Pointers

MPI maintains exactly one shared file pointer per collective MPI_FILE_OPEN (shared among processes in the communicator group). The current value of this pointer implicitly specifies 48

Unofficial Draft for Comment Only

45

```
1
      the offset in the data access routines described in this section. These routines only use and
\mathbf{2}
      update the shared file pointer maintained by MPI. The individual file pointers are not used
3
      nor updated.
4
          The shared file pointer routines have the same semantics as the data access with explicit
5
      offset routines described in Section 13.4.2, with the following modifications:
6
         • the offset is defined to be the current value of the MPI-maintained shared file pointer,
7
8
         • the effect of multiple calls to shared file pointer routines is defined to behave as if the
9
           calls were serialized, and
10
         • the use of shared file pointer routines is erroneous unless all processes use the same
11
           file view.
12
13
      For the noncollective shared file pointer routines, the serialization ordering is not determin-
14
      istic. The user needs to use other synchronization means to enforce a specific order.
15
          After a shared file pointer operation is initiated, the shared file pointer is updated to
16
      point to the next etype after the last one that will be accessed. The file pointer is updated
17
      relative to the current view of the file.
18
19
      Noncollective Operations
20
21
22
      MPI_FILE_READ_SHARED(fh, buf, count, datatype, status)
23
^{24}
        INOUT
                  fh
                                                file handle (handle)
25
        OUT
                  buf
                                                initial address of buffer (choice)
26
        IN
                  count
                                                number of elements in buffer (integer)
27
        IN
                                                datatype of each buffer element (handle)
28
                  datatype
29
        OUT
                  status
                                                status object (Status)
30
^{31}
      C binding
32
      int MPI_File_read_shared(MPI_File fh, void *buf, int count,
33
                      MPI_Datatype datatype, MPI_Status *status)
34
35
      F08 binding
36
      MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
37
          TYPE(MPI_File), INTENT(IN) :: fh
38
          TYPE(*), DIMENSION(...)
                                       :: buf
39
          INTEGER, INTENT(IN) :: count
40
          TYPE(MPI_Datatype), INTENT(IN) ::
                                                     datatype
41
          TYPE(MPI_Status) :: status
42
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                     ierror
43
      F binding
44
      MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
45
          <type> BUF(*)
46
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
47
48
          MPI_FILE_READ_SHARED reads a file using the shared file pointer.
```

MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status) ¹				
INOUT	fh	file handle (handle)	2	
IN	buf	initial address of buffer (choice)	3 4	
IN	count	number of elements in buffer (integer)	5	
IN	datatype	datatype of each buffer element (handle)	6	
OUT	status	status object (Status)	7	
			9	
C binding	g		10	
int MPI_H		e fh, const void *buf, int count,	11	
	MPI_Datatype datatyp	e, MPI_Status *status)	12 13	
F08 binding				
		nt, datatype, status, ierror)	14 15	
	<pre>(MPI_File), INTENT(IN) ::</pre>		16	
	(*), DIMENSION(), INTEN ER, INTENT(IN) :: count	I(IN) :: DUI	17	
	(MPI_Datatype), INTENT(IN)	:: datatype	18	
	(MPI_Status) :: status	51	19 20	
INTEC	ER, OPTIONAL, INTENT(OUT)	:: ierror	20	
F binding	g		22	
	-	INT, DATATYPE, STATUS, IERROR)	23	
• 1	> BUF(*)		24	
INTEC	ER FH, COUNT, DATATYPE, S	STATUS(MPI_STATUS_SIZE), IERROR	25	
MPI_I	FILE_WRITE_SHARED writes	a file using the shared file pointer.	26 27	
			28	
MPI_FILE	IREAD_SHARED(fh, buf, cour	nt, datatype, request)	29	
INOUT	fh	file handle (handle)	30 31	
OUT	buf	initial address of buffer (choice)	31	
	count	number of elements in buffer (integer)	33	
			34	
IN	datatype	datatype of each buffer element (handle)	35	
OUT	request	request object (handle)	36	
	_		37 38	
C binding		e fh, void *buf, int count,	39	
1110 111 1_1		e, MPI_Request *request)	40	
FOR hind			41	
F08 bind	0	nt, datatype, request, ierror)	42	
	(MPI_File), INTENT(IN) ::		43 44	
	(*), DIMENSION(), ASYNCH		44 45	
INTEGER, INTENT(IN) :: count				
TYPE(MPI_Datatype), INTENT(IN) :: datatype				
TYPE(MPI_Request), INTENT(OUT) :: request				

```
576
```

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
     F binding
3
     MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
4
          <type> BUF(*)
5
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
6
7
          MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
8
     interface.
9
10
     MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
11
12
       INOUT
                 fh
                                              file handle (handle)
13
       IN
                 buf
                                              initial address of buffer (choice)
14
       IN
                 count
                                              number of elements in buffer (integer)
15
16
       IN
                 datatype
                                              datatype of each buffer element (handle)
17
       OUT
                 request
                                              request object (handle)
18
19
     C binding
20
     int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
21
                     MPI_Datatype datatype, MPI_Request *request)
22
23
     F08 binding
^{24}
     MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
25
          TYPE(MPI_File), INTENT(IN) :: fh
26
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
27
          INTEGER, INTENT(IN) :: count
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          TYPE(MPI_Request), INTENT(OUT) ::
                                                  request
30
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                  ierror
^{31}
     F binding
32
     MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
33
          <type> BUF(*)
34
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
35
36
          MPI_FILE_IWRITE_SHARED is a nonblocking version of the
37
     MPI_FILE_WRITE_SHARED interface.
38
39
     Collective Operations
40
     The semantics of a collective access using a shared file pointer is that the accesses to the
41
     file will be in the order determined by the ranks of the processes within the group. For each
42
     process, the location in the file at which data is accessed is the position at which the shared
43
     file pointer would be after all processes whose ranks within the group less than that of this
44
     process had accessed their data. In addition, in order to prevent subsequent shared offset
45
     accesses by the same processes from interfering with this collective access, the call might
46
47
     return only after all the processes within the group have initiated their accesses. When the
48
```

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)									
			18						
INOUT	fh	file handle (handle)	19						
OUT	buf	initial address of buffer (choice)	20						
IN	count	number of elements in buffer (integer)	21						
IN	datatura		22						
IIN	datatype	datatype of each buffer element (handle)	23						
OUT	status	status object (Status)	24						
			25						
C binding									
int MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)									
					F08 binding				
					MPI_File	_read_ordered(fh, buf, co	ount, datatype, status, ierror)	31	
TYPE	(MPI_File), INTENT(IN) ::	fh	32						
TYPE(*), DIMENSION() :: buf									
INTE	GER, INTENT(IN) :: count	;	34						
TYPE	(MPI_Datatype), INTENT(IN) :: datatype	35						
TYPE(MPI_Status) :: status									
INTEGER, OPTIONAL, INTENT(OUT) :: ierror									
T 1 · 1 ·			38						
F bindin	0		39						
<pre>MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>									

MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED interface.

 $\mathbf{2}$

```
1
     MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       IN
                buf
                                            initial address of buffer (choice)
4
5
                                            number of elements in buffer (integer)
       IN
                count
6
       IN
                datatype
                                            datatype of each buffer element (handle)
7
       OUT
                status
                                            status object (Status)
8
9
10
     C binding
11
     int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
                    MPI_Datatype datatype, MPI_Status *status)
12
13
     F08 binding
14
     MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
15
          TYPE(MPI_File), INTENT(IN) :: fh
16
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
17
          INTEGER, INTENT(IN) :: count
18
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
          TYPE(MPI_Status) :: status
20
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                ierror
21
22
     F binding
23
     MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
^{24}
          <type> BUF(*)
25
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
26
          MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED
27
     interface.
28
29
     Seek
30
31
     If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
32
     to call the following two routines (MPI_FILE_SEEK_SHARED and
33
     MPI_FILE_GET_POSITION_SHARED).
34
35
     MPI_FILE_SEEK_SHARED(fh, offset, whence)
36
37
       INOUT
                fh
                                            file handle (handle)
38
       IN
                offset
                                            file offset (integer)
39
       IN
                whence
                                            update mode (state)
40
41
42
     C binding
43
     int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
44
     F08 binding
45
     MPI_File_seek_shared(fh, offset, whence, ierror)
46
          TYPE(MPI_File), INTENT(IN) :: fh
47
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
48
```

	GER, INTENT(I		$\frac{1}{2}$
	GER, UPIIUNAL	, INTENT(OUT) :: ierror	3
F bindir	ıg		4
		FH, OFFSET, WHENCE, IERROR)	5
	CGER FH, WHENC	-	6
INTE	GER(KIND=MPI_	OFFSET_KIND) OFFSET	7
	FILE_SEEK_SH	ARED updates the shared file pointer according to whence, which e values:	8 9
• MP	_SEEK_SET: the	pointer is set to offset	10 11
• MP	_SEEK_CUR: the	e pointer is set to the current pointer position plus offset	12 13
• MP	_SEEK_END: the	e pointer is set to the end of file plus offset	14 15
MPI.	_FILE_SEEK_SH	ARED is collective; all the processes in the communicator group	16
associate	d with the file ha	andle fh must call MPI_FILE_SEEK_SHARED with the same values	17
for offset	and whence.		18
The	offset can be neg	gative, which allows seeking backwards. It is erroneous to seek to	19
a negativ	e position in the	view.	20
			21
MPI FILF	E GET POSITIO	N_SHARED(fh, offset)	22
			23
IN	fh	file handle (handle)	24
OUT	offset	offset of shared pointer (integer)	25
			26 27
C bindi	0		28
int MPI_	File_get_posi	tion_shared(MPI_File fh, MPI_Offset *offset)	29
F08 bin	ding		30
MPI_File	e_get_position	_shared(fh, offset, ierror)	31
TYPE	E(MPI_File), I	NTENT(IN) :: fh	32
INTE	GER(KIND=MPI_	OFFSET_KIND), INTENT(OUT) :: offset	33
INTE	GER, OPTIONAL	, INTENT(OUT) :: ierror	34
F bindir	ıg		35
	•	_SHARED(FH, OFFSET, IERROR)	36
	GER FH, IERRO		37
INTE	GER(KIND=MPI_	OFFSET_KIND) OFFSET	38
MDI		CITION SHAPED noturns in effect the summent position of the	39 40
		SITION_SHARED returns, in offset, the current position of the be units relative to the current view.	40
A 7	vice to	he effect can be used in a fature sall to MDL FUE CEEK CHADED	42
		he offset can be used in a future call to MPI_FILE_SEEK_SHARED _SEEK_SET to return to the current position. To set the displace-	43
	0	t file pointer position, first convert offset into an absolute byte	44 45
		FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with	45 46

the resulting displacement. (End of advice to users.)

1 13.4.5 Split Collective Data Access Routines 2 MPI provides a restricted form of "nonblocking collective" I/O operations for all data ac-3 cesses using split collective data access routines. These routines are referred to as "split" 4 collective routines because a single collective operation is split in two: a begin routine and 5an end routine. The begin routine begins the operation, much like a nonblocking data access 6 (e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching 7 test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must 8 not use the buffer passed to a begin routine while the routine is outstanding; the operation 9 must be completed with an end routine before it is safe to free buffers, etc. 10 Split collective data access operations on a file handle fh are subject to the semantic 11 rules given below. 1213 • On any MPI process, each file handle may have at most one active split collective 14operation at any time. 1516• Begin calls are collective over the group of processes that participated in the collective 17 open and follow the ordering rules for collective calls. 18 • End calls are collective over the group of processes that participated in the collective 19open and follow the ordering rules for collective calls. Each end call matches the 20preceding begin call for the same collective operation. When an "end" call is made, 21exactly one unmatched "begin" call for the same operation must precede it. 2223• An implementation is free to implement any split collective data access routine using 24the corresponding blocking collective routine when either the begin call (e.g., 25MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is 26issued. The begin and end calls are provided to allow the user and MPI implementation 27to optimize the collective operation. 2829• Split collective operations do not match the corresponding regular collective opera-30 tion. For example, in a single collective read operation, an MPI_FILE_READ_ALL 31on one process does not match an MPI_FILE_READ_ALL_BEGIN/ 32 MPI_FILE_READ_ALL_END pair on another process. 33 • Split collective routines must specify a buffer in both the begin and end routines. 34 By specifying the buffer that receives data in the end routine, we can avoid the 35 problems described in "A Problem with Code Movements and Register Optimization," 36 Section 17.1.17, but not all of the problems, such as those described in Sections 17.1.12, 37 17.1.13, and 17.1.16. 38 39 • No collective I/O operations are permitted on a file handle concurrently with a split 40 collective access on that file handle (i.e., between the begin and end of the access). 41 That is 4243 MPI_File_read_all_begin(fh, ...); 4445MPI_File_read_all(fh, ...); 46 . . . 47 MPI_File_read_all_end(fh, ...); 48

is erroneous.

• In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

The arguments for these routines have the same meaning as for the equivalent collective versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL). The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation that, when completed with the matching end routine (i.e., MPI_FILE_READ_ALL_END) produces the result as defined for the equivalent collective routine (i.e., MPI_FILE_READ_ALL_END). For the number of consistency computing (Section 12.6.1), a metched pair of colling

For the purpose of consistency semantics (Section 13.6.1), a matched pair of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access.

MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)

C binding

F08 binding

```
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
   TYPE(MPI_File), INTENT(IN) :: fh
   INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
   TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
   INTEGER, INTENT(IN) :: count
   TYPE(MPI_Datatype), INTENT(IN) :: datatype
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

 $\mathbf{2}$

```
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
     C binding
8
     int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
9
     F08 binding
10
     MPI_File_read_at_all_end(fh, buf, status, ierror)
11
         TYPE(MPI_File), INTENT(IN) :: fh
12
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
13
         TYPE(MPI_Status) :: status
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     F binding
17
     MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
18
          <type> BUF(*)
19
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
20
21
22
     MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
23
       INOUT
                                            file handle (handle)
^{24}
                fh
25
       IN
                offset
                                            file offset (integer)
26
       IN
                buf
                                            initial address of buffer (choice)
27
       IN
                count
                                            number of elements in buffer (integer)
28
29
       IN
                datatype
                                            datatype of each buffer element (handle)
30
^{31}
     C binding
32
     int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
33
                    const void *buf, int count, MPI_Datatype datatype)
34
     F08 binding
35
36
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
37
         TYPE(MPI_File), INTENT(IN) :: fh
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
38
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
39
40
         INTEGER, INTENT(IN) :: count
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
43
     F binding
44
     MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
45
         <type> BUF(*)
46
         INTEGER FH, COUNT, DATATYPE, IERROR
47
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
48
```

MPI_FILE	_WRITE_AT_ALL_E	ND(fh, buf, status)	1
INOUT	fh	file handle (handle)	2
IN	buf	initial address of buffer (choice)	3 4
OUT	status	status object (Status)	5
			6
C bindin	g		7
int MPI_	File_write_at_all_	_end(MPI_File fh, const void *buf,	8
	MPI_Status *	status)	9 10
F08 bind	ling		10
	•	(fh, buf, status, ierror)	12
	(MPI_File), INTENT		13
		, INTENT(IN), ASYNCHRONOUS :: buf	14
	(MPI_Status) :: s		15
INTE	GER, OPTIONAL, INT	TENT(OUT) :: ierror	16
F bindin	g		17 18
		(FH, BUF, STATUS, IERROR)	18
• -	e> BUF(*)		20
INTE	GER FH, STATUS(MP)	I_STATUS_SIZE), IERROR	21
			22
	READ ALL RECIN	fh, buf, count, datatype)	23
			24
INOUT	fh	file handle (handle)	25
OUT	buf	initial address of buffer (choice)	26 27
IN	count	number of elements in buffer (integer)	28
IN	datatype	datatype of each buffer element (handle)	29
			30
C bindin	0		31
int MPI_	-	in(MPI_File fh, void *buf, int count,	32 33
	MPI_Datatype	datatype)	34
F08 bind	ling		35
	•	n, buf, count, datatype, ierror)	36
	(MPI_File), INTENT		37
		, ASYNCHRONOUS :: buf	38
INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype			
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			40
			41 42
<pre>F binding MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)</pre>			
1111			46
			47

```
1
     MPI_FILE_READ_ALL_END(fh, buf, status)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       OUT
                buf
                                            initial address of buffer (choice)
4
5
       OUT
                                            status object (Status)
                status
6
7
     C binding
8
     int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
9
     F08 binding
10
     MPI_File_read_all_end(fh, buf, status, ierror)
11
          TYPE(MPI_File), INTENT(IN) :: fh
12
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
13
          TYPE(MPI_Status) :: status
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     F binding
17
     MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
18
          <type> BUF(*)
19
          INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
20
21
22
     MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
23
       INOUT
                                            file handle (handle)
^{24}
                fh
25
       IN
                buf
                                            initial address of buffer (choice)
26
       IN
                count
                                            number of elements in buffer (integer)
27
       IN
                datatype
                                            datatype of each buffer element (handle)
28
29
30
     C binding
^{31}
     int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
32
                    MPI_Datatype datatype)
33
     F08 binding
34
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
35
          TYPE(MPI_File), INTENT(IN) :: fh
36
          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
37
          INTEGER, INTENT(IN) :: count
38
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     F binding
42
     MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
43
          <type> BUF(*)
44
          INTEGER FH, COUNT, DATATYPE, IERROR
45
46
47
48
```

MPI_FILE	_WRITE_ALL_END	(fh, buf, status)	1
INOUT	fh	file handle (handle)	2
IN	buf	initial address of buffer (choice)	3
OUT	status	status object (Status)	5
			6
C bindin	ıg		7
int MPI_	File_write_all_er	nd(MPI_File fh, const void *buf,	8
	MPI_Status	*status)	9 10
F08 bind	ling		10
MPI_File	_write_all_end(fl	n, buf, status, ierror)	12
	(MPI_File), INTEN		13
		.), INTENT(IN), ASYNCHRONOUS :: buf	14
	(MPI_Status) ::		15
INTE	GER, OPTIONAL, IN	NTENT(OUT) :: ierror	16
F bindin	g		17 18
		H, BUF, STATUS, IERROR)	19
• 1	e> BUF(*)		20
	GER FH, SIAIUS(MI	PI_STATUS_SIZE), IERROR	21
			22
	READ ORDERED	_BEGIN(fh, buf, count, datatype)	23
INOUT	fh		24
		file handle (handle)	25 26
OUT	buf	initial address of buffer (choice)	27
IN	count	number of elements in buffer (integer)	28
IN	datatype	datatype of each buffer element (handle)	29
			30
C bindin	0		31
int MPI_		<pre>l_begin(MPI_File fh, void *buf, int count,</pre>	32 33
	MPI_Datatyp	e datatype)	34
F08 bind	•		35
		gin(fh, buf, count, datatype, ierror)	36
	(MPI_File), INTEN		37
		.), ASYNCHRONOUS :: buf	38 39
INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype			
INTEGER, OPTIONAL, INTENT(OUT) :: ierror			40 41
			42
F bindin	•	GIN(FH, BUF, COUNT, DATATYPE, IERROR)	43
	e> BUF(*)	in (in, boi, cooki, brinili, ilidion)	44
• 1	INTEGER FH, COUNT, DATATYPE, IERROR		
			46
			47

```
586
```

```
1
     MPI_FILE_READ_ORDERED_END(fh, buf, status)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       OUT
                buf
                                            initial address of buffer (choice)
4
5
       OUT
                                            status object (Status)
                status
6
7
     C binding
8
     int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
9
     F08 binding
10
     MPI_File_read_ordered_end(fh, buf, status, ierror)
11
         TYPE(MPI_File), INTENT(IN) :: fh
12
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
13
         TYPE(MPI_Status) :: status
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     F binding
17
     MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
18
          <type> BUF(*)
19
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
20
21
22
     MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
23
                                            file handle (handle)
       INOUT
^{24}
                fh
25
       IN
                buf
                                            initial address of buffer (choice)
26
       IN
                count
                                            number of elements in buffer (integer)
27
       IN
                datatype
                                            datatype of each buffer element (handle)
28
29
30
     C binding
^{31}
     int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
32
                    MPI_Datatype datatype)
33
     F08 binding
34
     MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
35
         TYPE(MPI_File), INTENT(IN) :: fh
36
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
37
         INTEGER, INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     F binding
42
     MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
43
         <type> BUF(*)
44
         INTEGER FH, COUNT, DATATYPE, IERROR
45
46
47
48
```

C binding

F08 binding

```
MPI_File_write_ordered_end(fh, buf, status, ierror)
   TYPE(MPI_File), INTENT(IN) :: fh
   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
   TYPE(MPI_Status) :: status
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

F binding

13.5 File Interoperability

At the most basic level, file interoperability is the ability to read the information previously written to a file — not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 13.5.2) as well as the data conversion functions (Section 13.5.3).

Interoperability within a single MPI environment (which could be considered "operability") ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 13.6.1), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar

Unofficial Draft for Comment Only

1 2

4

5

6 7

9

10

11

12

13

14

15

16 17

18

19

20

21 22 23

24 25

26

27

28

29

30

 31

32

33

34

35

36

37

38

39

40 41

42 43

1 operations similar to POSIX cp, rm, and mv can be performed on the file. Furthermore, it $\mathbf{2}$ is expected that the facility provided maintains the correspondence between absolute byte 3 offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the 4 MPI environment are at byte offset 102 outside the MPI environment). As an example, $\mathbf{5}$ a simple off-line conversion utility that transfers and converts files between the native file 6 system and the MPI environment would suffice, provided it maintained the offset coherence 7mentioned above. In a high-quality implementation of MPI, users will be able to manipulate 8 MPI files using the same or similar tools that the native file system offers for manipulating 9 its files. 10 The remaining aspect of file interoperability, converting between different machine

¹¹ representations, is supported by the typing information specified in the etype and filetype. ¹² This facility allows the information in files to be shared between any two applications, ¹³ regardless of whether they use MPI, and regardless of the machine architectures on which ¹⁴ they run.

¹⁵ MPI supports multiple data representations: "native," "internal," and "external32." ¹⁶ An implementation may support additional data representations. MPI also supports user-¹⁷ defined data representations (see Section 13.5.3). The "native" and "internal" data repre-¹⁸ sentations are implementation dependent, while the "external32" representation is common ¹⁹ to all MPI implementations and facilitates file interoperability. The data representation is ²⁰ specified in the datarep argument to MPI_FILE_SET_VIEW.

- Advice to users. MPI is not guaranteed to retain knowledge of what data representation was used when a file is written. Therefore, to correctly retrieve file data, an MPI application is responsible for specifying the same data representation as was used to create the file. (*End of advice to users.*)
- "native" Data in this representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.
 - Advice to users. This data representation should only be used in a homogeneous MPI environment, or when the MPI application is capable of performing the data type conversions itself. (*End of advice to users.*)
 - Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be typed as MPI_BYTE to ensure that the message routines do not perform any type conversions on the data. (*End of advice to implementors.*)
- ³⁹
 "internal" This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file. The environment in which the resulting file can be reused is implementation-defined and must be documented by the implementation.
- Rationale. This data representation allows the implementation to perform I/O
 efficiently in a heterogeneous environment, though with implementation-defined
 restrictions on how the file can be reused. (*End of rationale.*)

Unofficial Draft for Comment Only

21

22

23

 24

25

31

32

33 34

35

36

37

 $\mathbf{2}$

 31

Advice to implementors. Since "external32" is a superset of the functionality provided by "internal," an implementation may choose to implement "internal" as "external32." (*End of advice to implementors.*)

"external32" This data representation states that read and write operations convert all data from and to the "external32" representation defined in Section 13.5.2. The data conversion rules for communication also apply to these conversions (see Section 3.3.2). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process's native representation.

This data representation has several advantages. First, all processes reading the file in a heterogeneous MPI environment will automatically have the data converted to their respective native representations. Second, the file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file.

The disadvantage of this data representation is that data precision and I/O performance may be lost in data type conversions.

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be converted to and from the "external32" representation in the client, and sent as type MPI_BYTE. This will avoid possible double data type conversions and the associated further loss of precision and performance. (*End of advice to implementors.*)

13.5.1 Datatypes for File Interoperability

If the file data representation is other than "native," care must be taken in constructing etypes and filetypes. Any of the datatype constructor functions may be used; however, for those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI_FILE_GET_TYPE_EXTENT can be used to calculate the extents of datatypes in the file. For etypes and filetypes that are portable datatypes (see Section 2.4), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

Advice to users. One can logically think of the file as if it were stored in the memory of a file server. The etype and filetype are interpreted as if they were defined at this file server, by the same sequence of calls used to define them at the calling process. If the data representation is "native", then this logical file server runs on the same architecture as the calling process, so that these types define the same data layout on the file as they would define in the memory of the calling process. If the etype and filetype are portable datatypes, then the data layout defined in the file is the same as would be defined in the calling process memory, up to a scaling factor. The routine MPI_FILE_GET_TYPE_EXTENT can be used to calculate this scaling factor. Thus, two equivalent, portable datatypes will define the same data layout in the file, even in a heterogeneous environment with "internal", "external32", or user defined data representations. Otherwise, the etype and filetype must be constructed so that

1 their typemap and extent are the same on any architecture. This can be achieved 2 if they have an explicit upper bound and lower bound (defined using 3 MPI_TYPE_CREATE_RESIZED). This condition must also be fulfilled by any datatype 4 that is used in the construction of the etype and filetype, if this datatype is replicated 5contiguously, either explicitly, by a call to MPI_TYPE_CONTIGUOUS, or implicitly, 6 by a blocklength argument that is greater than one. If an etype or filetype is not 7 portable, and has a typemap or extent that is architecture dependent, then the data 8 layout specified by it on a file is implementation dependent. 9 File data representations other than "native" may be different from corresponding 10 data representations in memory. Therefore, for these file data representations, it is 11 important not to use hardwired byte offsets for file positioning, including the initial 12displacement that specifies the view. When a portable datatype (see Section 2.4) is 13 used in a data access operation, any holes in the datatype are scaled to match the data 14representation. However, note that this technique only works when all the processes 15that created the file view build their etypes from the same predefined datatypes. For 16example, if one process uses an etype built from MPI_INT and another uses an etype 17 built from MPI_FLOAT, the resulting views may be nonportable because the relative 18 sizes of these types may differ from one data representation to another. (End of advice 19 to users.) 202122MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent) 23 24 IN fh file handle (handle) 25IN datatype datatype (handle) 26OUT extent datatype extent (integer) 2728C binding 29 int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype, 30 MPI_Aint *extent) 31 32 F08 binding 33 MPI_File_get_type_extent(fh, datatype, extent, ierror) 34 TYPE(MPI_File), INTENT(IN) :: fh 35 TYPE(MPI_Datatype), INTENT(IN) :: datatype 36 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 F binding MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR) 40INTEGER FH, DATATYPE, IERROR 41 INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT 4243 Returns the extent of datatype in the file fh. This extent will be the same for all 44processes accessing the file fh. If the current view uses a user-defined data representation 45(see Section 13.5.3), MPI uses the dtype_file_extent_fn callback to calculate the extent. 46 47Advice to implementors. In the case of user-defined data representations, the extent 48 of a derived datatype can be calculated by first determining the extents of the prede-

fined 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 [58].

All signed numerals (e.g., MPI_INT, MPI_REAL) have the sign bit at the most significant bit. MPI_COMPLEX and MPI_DOUBLE_COMPLEX have the sign bit of the real and imaginary parts at the most significant bit of each part.

According to IEEE specifications [37], the "NaN" (not a number) is system dependent. It should not be interpreted within MPI as anything other than "NaN."

Advice to implementors. The MPI treatment of "NaN" is similar to the approach used in XDR (see https://www.ietf.org/rfc/rfc1832.txt). (End of advice to implementors.)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

Advice to implementors. All bytes of LOGICAL and bool must be checked to determine the value. (*End of advice to implementors.*)

Advice to users. The type MPI_PACKED is treated as bytes and is not converted. The user should be aware that MPI_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

The sizes of the predefined datatypes returned from MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_COMPLEX, and MPI_TYPE_CREATE_F90_INTEGER are defined in Section 17.1.9, page 643.

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.

Unofficial Draft for Comment Only

 $41 \\ 42$

 $45 \\ 46$

Туре 	Length	Optional Type	Length
MPI_PACKED	1	MPI_INTEGER1	1
MPI_BYTE	1	MPI_INTEGER2	2
MPI_CHAR	1	MPI_INTEGER4	4
MPI_UNSIGNED_CHAR	1	MPI_INTEGER8	8
MPI_SIGNED_CHAR	1	MPI_INTEGER16	16
MPI_WCHAR	2		
MPI_SHORT	2	MPI_REAL2	2
MPI_UNSIGNED_SHORT	2	MPI_REAL4	4
MPI_INT	4	MPI_REAL8	8
MPI_UNSIGNED	4	MPI_REAL16	16
MPI_LONG	4		10
MPI_UNSIGNED_LONG	4	MPI_COMPLEX4	2*2
MPI_LONG_LONG_INT		MPI_COMPLEX8	2*2 2*4
		MPI_COMPLEX16	
MPI_UNSIGNED_LONG_LONG	8	_	2*8
MPI_FLOAT	4	MPI_COMPLEX32	2*16
MPI_DOUBLE	8		
1PI_LONG_DOUBLE	16		
MPI_C_BOOL	1		
MPI_INT8_T	1	C++ Types	Length
MPI_INT16_T	2		
MPI_INT32_T	4	MPI_CXX_BOOL	1
MPI_INT64_T	8	MPI_CXX_FLOAT_COMPLEX	2*4
MPI_UINT8_T	1	MPI_CXX_DOUBLE_COMPLEX	
MPI_UINT16_T	2	MPI_CXX_LONG_DOUBLE_COMF	
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 2*4		
MPI_C_DOUBLE_COMPLEX	2*4 2*8		
MPI_C_DOUBLE_COMPLEX MPI_C_LONG_DOUBLE_COMPLEX			
MFI_C_LONG_DOODLE_COMPLEA	2*10		
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		
		sizes of predefined datatypes	

13.5.3	User-Defined Data Represen	tations	1	
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 writte	n in a representation unknown to the implementation.	5 6	
		s allow the user to insert a third party converter into	7	
	stream to do the data representation		8	
1			9	
MPI_RE	EGISTER_DATAREP(datarep, dtype_file_extent_fn, e	read_conversion_fn, write_conversion_fn, extra_state)	10 11 12	
IN	datarep	data representation identifier (string)	13	
IN	read_conversion_fn	function invoked to convert from file representation to native representation (function)	14 15 16	
IN	write_conversion_fn	function invoked to convert from native representation to file representation (function)	17	
IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as represented in the file (function)	19 20	
IN	extra_state	extra state	21 22	
C bind int MP	I_Register_datarep(const MPI_Datarep_conver MPI_Datarep_conver	<pre>sion_function *read_conversion_fn, sion_function *write_conversion_fn, _function *dtype_file_extent_fn,</pre>	24 25 26 25 28 28	
F08 bi	nding		30	
MPI_Re	gister_datarep(datarep, r	ead_conversion_fn, write_conversion_fn, fn, extra_state, ierror)) :: datarep	31 32 33	
	DCEDURE(MPI_Datarep_conve	-	34	
PR	CEDURE(MPI_Datarep_conve	rsion_function) :: write_conversion_fn	35	
	CEDURE(MPI_Datarep_exten	• -	30 3'	
	FEGER(KIND=MPI_ADDRESS_KI FEGER, OPTIONAL, INTENT(O		38	
F bind			39	
	GISTER_DATAREP(DATAREP, R	EAD_CONVERSION_FN, WRITE_CONVERSION_FN, FN, EXTRA_STATE, IERROR)	40 41	
CH	ARACTER*(*) DATAREP	···· <u>-</u> ·······,·····	42 43	
		, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN	44	
	FEGER(KIND=MPI_ADDRESS_KI FEGER IERROR	ND) EXTRA_STATE	45	
			46	
Τh	e call associates read_convers	ion_fn, write_conversion_fn, and dtype_file_extent_fn	47	

The call associates read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn ⁴⁷ with the data representation identifier datarep. datarep can then be used as an argument ⁴⁸

1 to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conver- $\mathbf{2}$ sion functions to convert all data items accessed between file data representation and na-3 tive representation. MPI_REGISTER_DATAREP is a local operation and only registers the 4 data representation for the calling MPI process. If datarep is already defined, an error in the error class MPI_ERR_DUP_DATAREP is raised using the default file error handler $\mathbf{5}$ 6 (see Section 13.7). The length of a data representation string is limited to the value of 7MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64. 8 No routines are provided to delete data representations and free the associated resources; 9 it is not expected that an application will generate them in significant numbers. 10 11Extent Callback 12typedef int MPI_Datarep_extent_function(MPI_Datatype datatype, 13 MPI_Aint *file_extent, void *extra_state); 1415ABSTRACT INTERFACE 16SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state, 17 ierror) 18 TYPE(MPI_Datatype) :: datatype 19 INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state 20INTEGER :: ierror 21SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR) 22 INTEGER DATATYPE, IERROR 23INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE 2425The function dtype_file_extent_fn must return, in file_extent, the number of bytes re-26quired to store datatype in the file representation. The function is passed, in extra_state, 27the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call 28this routine with predefined datatypes employed by the user. 29 30 Datarep Conversion Functions 31 typedef int MPI_Datarep_conversion_function(void *userbuf, 32 33 MPI_Datatype datatype, int count, void *filebuf, 34 MPI_Offset position, void *extra_state); 35ABSTRACT INTERFACE 36 SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, 37 filebuf, position, extra_state, ierror) 38 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 39 TYPE(C_PTR), VALUE :: userbuf, filebuf 40 TYPE(MPI_Datatype) :: datatype 41 INTEGER :: count, ierror 42INTEGER(KIND=MPI_OFFSET_KIND) :: position 43 INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state 44 SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF, 4546POSITION, EXTRA_STATE, IERROR) 47 <TYPE> USERBUF(*), FILEBUF(*) 48 INTEGER COUNT, DATATYPE, IERROR

INTEGER(KIND=MPI_OFFSET_KIND) POSITION INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

The function read_conversion_fn must convert from file data representation to native representation. Before calling this routine, MPI allocates and fills filebuf with count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. The function must copy all count data items from filebuf to userbuf in the distribution described by datatype, converting each data item from file representation to native representation. datatype will be equivalent to the datatype that the user passed to the read function. If the size of datatype is less than the size of the count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf. The conversion function must begin storing converted data at the location in userbuf specified by position into the (tiled) datatype.

Advice to users. Although the conversion functions have similarities to MPI_PACK and MPI_UNPACK, one should note the differences in the use of the arguments count and position. In the conversion functions, count is a count of data items (i.e., count of typemap entries of datatype), and position is an index into this typemap. In MPI_PACK, incount refers to the number of whole datatypes, and position is a number of bytes. (*End of advice to users.*)

Advice to implementors. A converted read operation could be implemented as follows:

- 1. Get file extent of all data items
- 2. Allocate a filebuf large enough to hold all count data items
- 3. Read data from file into filebuf
- 4. Call read_conversion_fn to convert data and place it into userbuf
- 5. Deallocate filebuf

(End of advice to implementors.)

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call, and the userbuf pointer will be unchanged.

Rationale. Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion function can use the **position** to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

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

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.*)

4 5

1

2

3

6 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 $\overline{7}$ hold **count** contiguous data items. The type of each data item matches the corresponding 8 9 entry for the predefined datatype in the type signature of datatype. The function must copy 10 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 repre-11sentation. If the size of datatype is less than the size of count data items, the conversion 12function must treat datatype as being contiguously tiled over the userbuf. 13

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.

¹⁸ The predefined constant MPI_CONVERSION_FN_NULL may be used as either ¹⁹ write_conversion_fn or read_conversion_fn. In that case, MPI will not attempt to invoke ²⁰ write_conversion_fn or read_conversion_fn, respectively, but will perform the requested data ²¹ 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.

²⁶ 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.

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.

38 39

13.5.4 Matching Data Representations

⁴⁰ 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 48 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 17.1.9).
- 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 17.1.9 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

File Consistency 13.6.1

Consistency semantics define the outcome of multiple accesses to a single file. All file accesses in MPI are relative to a specific file handle created from a collective open. MPI provides three levels of consistency: sequential consistency among all accesses using a single 27file handle, sequential consistency among all accesses using file handles created from a single 28collective open with atomic mode enabled, and user-imposed consistency among accesses 29other than the above. Sequential consistency means the behavior of a set of operations will 30 be as if the operations were performed in some serial order consistent with program order; 31each access appears atomic, although the exact ordering of accesses is unspecified. Userimposed consistency may be obtained using program order and calls to MPI_FILE_SYNC. 33

Let FH_1 be the set of file handles created from one particular collective open of the 34file FOO, and FH_2 be the set of file handles created from a different collective open of 35 FOO. Note that nothing restrictive is said about FH_1 and FH_2 : the sizes of FH_1 and 36 FH_2 may be different, the groups of processes used for each open may or may not intersect, 37 the file handles in FH_1 may be destroyed before those in FH_2 are created, etc. Consider 38 the following three cases: a single file handle (e.g., $fh_1 \in FH_1$), two file handles created 39 from a single collective open (e.g., $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$), and two file handles from 40 different collective opens (e.g., $fh_1 \in FH_1$ and $fh_2 \in FH_2$). 41

For the purpose of consistency semantics, a matched pair (Section 13.4.5) of split col-42lective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access operation. Similarly, a non-44blocking 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.

Unofficial Draft for Comment Only

1 2

3

4

 $\mathbf{5}$

6 $\overline{7}$

8

9

10 11

12

13 14

15

1617

18

192021

2223

 24

25

26

32

43

45

46

47

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.

⁹ Let SEQ_{fh} be a sequence of file operations on a single file handle, bracketed by ¹⁰ MPI_FILE_SYNCs on that file handle. (Both opening and closing a file implicitly perform ¹¹ an MPI_FILE_SYNC.) SEQ_{fh} is a "write sequence" if any of the data access operations in ¹² the sequence are writes or if any of the file manipulation operations in the sequence change ¹³ the state of the file (e.g., MPI_FILE_SET_SIZE or MPI_FILE_PREALLOCATE). Given two ¹⁴ sequences, SEQ_1 and SEQ_2 , we say they are not *concurrent* if one sequence is guaranteed ¹⁵ 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.

19

23

²⁰ Case 1: $fh_1 \in FH_1$ All operations on fh_1 are sequentially consistent if atomic mode is ²¹ set. If nonatomic mode is set, then all operations on fh_1 are sequentially consistent if they ²² are either nonconcurrent, nonconflicting, or both.

²⁴ Case 2: $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$ Assume A_1 is a data access operation using fh_{1a} , ²⁵ and A_2 is a data access operation using fh_{1b} . If for any access A_1 , there is no access A_2 ²⁶ that conflicts with A_1 , then MPI guarantees sequential consistency.

²⁷ However, unlike POSIX semantics, the default MPI semantics for conflicting accesses ²⁸ do not guarantee sequential consistency. If A_1 and A_2 conflict, sequential consistency can be ²⁹ guaranteed by either enabling atomic mode via the MPI_FILE_SET_ATOMICITY routine, ³⁰ or meeting the condition described in Case 3 below.

31 32

33

34

Case 3: $fh_1 \in FH_1$ and $fh_2 \in FH_2$ Consider access to a single file using file handles from distinct collective opens. In order to guarantee sequential consistency, MPI_FILE_SYNC must be used (both opening and closing a file implicitly perform an MPI_FILE_SYNC).

³⁵ Sequential consistency is guaranteed among accesses to a single file if for any write ³⁶ sequence SEQ_1 to the file, there is no sequence SEQ_2 to the file which is *concurrent* with ³⁷ SEQ_1 . To guarantee sequential consistency when there are write sequences,

³⁸ MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency ³⁹ of the sequences.

- 42
- 43 44
- 45

46

- 40
- 48

1

2

3

⁴⁰ See the examples in Section 13.6.11 for further clarification of some of these consistency semantics.

MPI_FILE_SET_ATOMICITY(fh, flag)				
INOUT	fh	file handle (handle)	2	
IN	flag	true to set atomic mode, false to set nonatomic mode	3 4	
	J. J	(logical)	5	
			6	
C binding	•		7	
int MPI_F	File_set_atomicity(MPI_Fi	le fh, int flag)	8	
F08 bind	ing		9 10	
	_set_atomicity(fh, flag,		10	
	TYPE(MPI_File), INTENT(IN) :: fh			
	CAL, INTENT(IN) :: flag	N	13	
INTEG	ER, OPTIONAL, INTENT(OUT) :: ierror	14	
F binding	g		15	
	_SET_ATOMICITY(FH, FLAG,	IERROR)	16	
	SER FH, IERROR		17 18	
LUGIC	CAL FLAG		19	
		created by one collective open. The consistency	20	
		sing FH is set by collectively calling	21	
		MPI_FILE_SET_ATOMICITY is collective; all pro- values for fh and flag. If flag is true, atomic mode is	22	
	is false, nonatomic mode is se		23	
		s for an open file only affects new data accesses.	24 25	
		eed to abide by the consistency semantics in effect	25 26	
-	_	a accesses and split collective operations that have	27	
-		re only guaranteed to abide by nonatomic mode	28	
consistenc	y semantics.		29	
Advi	<i>ce to implementors</i> . Since the	e semantics guaranteed by atomic mode are stronger	30	
	-	mic mode, an implementation is free to adhere to	31	
	8	semantics for outstanding requests. (End of advice	32 33	
to in	n plementors.)		34	
			35	
			36	
MPI_FILE	MPI_FILE_GET_ATOMICITY(fh, flag)			
IN	fh	file handle (handle)	38	
OUT	flag	true if atomic mode, false if nonatomic mode (logical)	39	
001	106		40 41	
C binding			42	
int MPI_File_get_atomicity(MPI_File fh, int *flag)			43	
F08 binding			44	
	MPI_File_get_atomicity(fh, flag, ierror)			
	TYPE(MPI_File), INTENT(IN) :: fh			
	LOGICAL, INTENT(OUT) :: flag			

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
      F binding
3
     MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
4
          INTEGER FH, IERROR
5
          LOGICAL FLAG
6
7
          MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access
8
      operations on the set of file handles created by one collective open. If flag is true, atomic
9
      mode is enabled; if flag is false, nonatomic mode is enabled.
10
11
     MPI_FILE_SYNC(fh)
12
13
       INOUT
                  fh
                                               file handle (handle)
14
15
      C binding
16
      int MPI_File_sync(MPI_File fh)
17
     F08 binding
18
     MPI_File_sync(fh, ierror)
19
          TYPE(MPI_File), INTENT(IN) ::
20
                                               fh
21
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                    ierror
22
      F binding
23
     MPI_FILE_SYNC(FH, IERROR)
^{24}
          INTEGER FH, IERROR
25
26
          Calling MPI_FILE_SYNC with fh causes all previous writes to fh by the calling process
      to be transferred to the storage device. If other processes have made updates to the storage
27
      device, then all such updates become visible to subsequent reads of fh by the calling process.
28
      MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see
29
30
      above).
^{31}
          MPI_FILE_SYNC is a collective operation.
          The user is responsible for ensuring that all nonblocking requests and split collective
32
      operations on fh have been completed before calling MPI_FILE_SYNC — otherwise, the call
33
34
     to MPI_FILE_SYNC is erroneous.
35
36
      13.6.2
              Random Access vs. Sequential Files
37
      MPI distinguishes ordinary random access files from sequential stream files, such as pipes
38
      and tape files. Sequential stream files must be opened with the MPI_MODE_SEQUENTIAL
39
      flag set in the amode. For these files, the only permitted data access operations are shared
40
      file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the
41
      notion of file pointer is not meaningful; therefore, calls to MPI_FILE_SEEK_SHARED and
42
      MPI_FILE_GET_POSITION_SHARED are erroneous, and the pointer update rules specified
43
      for the data access routines do not apply. The amount of data accessed by a data access
44
      operation will be the amount requested unless the end of file is reached or an error is raised.
45
46
           Rationale. This implies that reading on a pipe will always wait until the requested
47
           amount of data is available or until the process writing to the pipe has issued an end
48
           of file. (End of rationale.)
```

 $\mathbf{2}$

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

Finally, for some sequential files, such as those corresponding to magnetic tapes or streaming network connections, writes to the file may be destructive. In other words, a write may act as a truncate (a MPI_FILE_SET_SIZE with size set to the current position) followed by the write.

13.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

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.

13.6.4 Collective File Operations

Collective file operations are subject to the same restrictions as collective communication operations. For a complete discussion, please refer to the semantics set forth in Section 5.14.

Collective file operations are collective over a duplicate of the communicator used to open the file — this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

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.

Nonblocking collective file operations are subject to the same restrictions as blocking collective I/O operations. All processes belonging to the group of the communicator that was used to open the file must call collective I/O operations (blocking and nonblocking) in the same order. This is consistent with the ordering rules for collective operations in threaded environments. For a complete discussion, please refer to the semantics set forth in Section 5.14.

Nonblocking collective I/O operations do not match with blocking collective I/O operations. Multiple nonblocking collective I/O operations can be outstanding on a single file handle. High quality MPI implementations should be able to support a large number of pending nonblocking I/O operations.

All nonblocking collective I/O calls are local and return immediately, irrespective of the status of other processes. The call initiates the operation which may progress independently of any communication, computation, or I/O. The call returns a request handle, which must be passed to a completion call. Input buffers should not be modified and output buffers should not be accessed before the completion call returns. The same progress rules described 48

1 for nonblocking collective operations apply for nonblocking collective I/O operations. For $\mathbf{2}$ a complete discussion, please refer to the semantics set forth in Section 5.12.

3 4

5

6

7

8

9

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. 10

11

1213

14

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.)

151617

13.6.7 Miscellaneous Clarifications

18 Once an I/O routine completes, it is safe to free any opaque objects passed as arguments 19to that routine. For example, the comm and info used in an MPI_FILE_OPEN, or the etype 20and filetype used in an MPI_FILE_SET_VIEW, can be freed without affecting access to the 21file. Note that for nonblocking routines and split collective operations, the operation must 22be completed before it is safe to reuse data buffers passed as arguments.

23As in communication, datatypes must be committed before they can be used in file 24 manipulation or data access operations. For example, the etype and filetype must be com-25mitted before calling MPI_FILE_SET_VIEW, and the datatype must be committed before 26calling MPI_FILE_READ or MPI_FILE_WRITE.

2728

29

33

13.6.8 MPI_Offset Type

MPI_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest 30 31 file supported by MPI. Displacements and offsets are always specified as values of type MPI_Offset. 32

In Fortran, the corresponding integer is an integer with kind parameter

34MPI_OFFSET_KIND, which is defined in the mpi_f08 module, the mpi module and the mpif.h include file. 35

In Fortran 77 environments that do not support KIND parameters, MPI_Offset argu-36 37 ments should be declared as an INTEGER of suitable size. The language interoperability implications for MPI_Offset are similar to those for addresses (see Section 17.2). 38

39 40

41

13.6.9 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not 42how that file structure is to be stored on one or more disks. Specification of the physical 43 file structure was avoided because it is expected that the mapping of files to disks will be 44 system specific, and any specific control over file layout would therefore restrict program 45 portability. However, there are still cases where some information may be necessary to 46 optimize file layout. This information can be provided as *hints* specified via info when a file 47is created (see Section 13.2.8). 48

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI *size changing* routines, such as MPI_FILE_SET_SIZE. A call to a size changing routine does not necessarily change the file size. For example, calling MPI_FILE_PREALLOCATE with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since MPI_FILE_OPEN if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

- One plus the displacement of the high byte.
- The size immediately after the size changing routine, or MPI_FILE_OPEN, returned.

When applying consistency semantics, calls to MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE are considered writes to the file (which conflict with operations 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).

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.*)

File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.

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.*)

13.6.11 Examples

The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address

- conflicting accesses on file handles obtained from a single collective open, and
- all accesses on file handles obtained from two separate collective opens.

The simplest way to achieve consistency for conflicting accesses is to obtain sequential consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 integers. If the latter, every element of **b** will be 5. If nonatomic mode is set, the results of the read are undefined.

 31

```
1
     /* Process 0 */
\mathbf{2}
     int i, a[10];
3
     int TRUE = 1;
4
\mathbf{5}
     for (i=0;i<10;i++)</pre>
6
        a[i] = 5;
7
8
     MPI_File_open(MPI_COMM_WORLD, "workfile",
9
                    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
10
     MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
11
     MPI_File_set_atomicity(fh0, TRUE);
12
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
13
     /* MPI_Barrier(MPI_COMM_WORLD); */
14
     /* Process 1 */
15
16
     int b[10];
     int TRUE = 1;
17
     MPI_File_open(MPI_COMM_WORLD, "workfile",
18
19
                     MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
     MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
20
21
     MPI_File_set_atomicity(fh1, TRUE);
     /* MPI_Barrier(MPI_COMM_WORLD); */
22
     MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
23
^{24}
     A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
25
     temporal order with, for example, calls to MPI_BARRIER.
26
27
          Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
28
          order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
29
          received by process 1 using MPI_RECV. (End of advice to users.)
30
31
         Alternatively, a user can impose consistency with nonatomic mode set:
32
33
     /* Process 0 */
34
     int i, a[10];
35
     for (i=0;i<10;i++)</pre>
36
        a[i] = 5;
37
38
     MPI_File_open(MPI_COMM_WORLD, "workfile",
39
                     MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
40
     MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
41
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
42
     MPI_File_sync(fh0);
43
     MPI_Barrier(MPI_COMM_WORLD);
44
     MPI_File_sync(fh0);
45
46
     /* Process 1 */
47
     int b[10];
48
     MPI_File_open(MPI_COMM_WORLD, "workfile",
```

```
1
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
                                                                                      2
MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                      3
MPI_File_sync(fh1);
MPI_Barrier(MPI_COMM_WORLD);
                                                                                      4
MPI_File_sync(fh1);
                                                                                      5
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
                                                                                      6
                                                                                      7
The "sync-barrier-sync" construct is required because:
                                                                                      8
                                                                                      9
   • The barrier ensures that the write on process 0 occurs before the read on process 1.
                                                                                      10
   • The first sync guarantees that the data written by all processes is transferred to the
                                                                                      11
     storage device.
                                                                                      12
                                                                                      13
   • The second sync guarantees that all data which has been transferred to the storage
                                                                                      14
     device is visible to all processes. (This does not affect process 0 in this example.)
                                                                                      15
    The following program represents an erroneous attempt to achieve consistency by elim-
                                                                                      16
inating the apparently superfluous second "sync" call for each process.
                                                                                      17
                                                                                      18
/* ----- THIS EXAMPLE IS ERRONEOUS ----- */
                                                                                      19
/* Process 0 */
                                                                                      20
int i, a[10];
                                                                                      21
for (i=0;i<10;i++)</pre>
                                                                                      22
   a[i] = 5;
                                                                                      23
                                                                                      24
MPI_File_open(MPI_COMM_WORLD, "workfile",
                                                                                      25
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
                                                                                      26
MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                      27
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
                                                                                      28
MPI_File_sync(fh0);
                                                                                      29
MPI_Barrier(MPI_COMM_WORLD);
                                                                                      30
/* Process 1 */
                                                                                      31
int b[10];
                                                                                      32
MPI_File_open(MPI_COMM_WORLD, "workfile",
                                                                                      33
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
                                                                                      34
MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                      35
MPI_Barrier(MPI_COMM_WORLD);
                                                                                      36
MPI_File_sync(fh1);
                                                                                      37
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
                                                                                      38
                                                                                      39
/* ----- THIS EXAMPLE IS ERRONEOUS ----- */
                                                                                      40
                                                                                      41
```

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 44 as a temporally synchronizing function. When using such an implementation, the 45 "sync-barrier-sync" construct above can be replaced by a single "sync." The results of 46 using such code with an implementation for which MPI_FILE_SYNC is not temporally 47 synchronizing is undefined. (*End of advice to users.*) 48

Unofficial Draft for Comment Only

42

```
606
```

```
1
     Asynchronous I/O
\mathbf{2}
     The behavior of asynchronous I/O operations is determined by applying the rules specified
3
     above for synchronous I/O operations.
4
         The following examples all access a preexisting file "myfile." Word 10 in myfile initially
5
     contains the integer 2. Each example writes and reads word 10.
6
         First consider the following code fragment:
7
8
     int a = 4, b, TRUE=1;
9
     MPI_File_open(MPI_COMM_WORLD, "myfile",
10
                     MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
11
     MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
12
     /* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */
13
     MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
14
     MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &regs[1]);
15
     MPI_Waitall(2, reqs, statuses);
16
17
     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 corre-
18
     sponding request complete routine. Thus, executing either the read before the write, or the
19
     write before the read is consistent with program order. If atomic mode is set, then MPI
20
21
     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
22
     something other than 2 or 4 due to the conflicting data access.
23
         Similarly, the following code fragment does not order file accesses:
^{24}
25
     int a = 4, b;
26
     MPI_File_open(MPI_COMM_WORLD, "myfile",
27
                     MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
28
     MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
29
     /* MPI_File_set_atomicity(fh, TRUE); Use this to set atomic mode. */
30
     MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
31
     MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
32
     MPI_Wait(&regs[0], &status);
33
     MPI_Wait(&reqs[1], &status);
34
35
     If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee
36
     sequential consistency in nonatomic mode.
37
         On the other hand, the following code fragment:
38
39
     int a = 4, b;
     MPI_File_open(MPI_COMM_WORLD, "myfile",
40
                     MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
41
     MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
42
     MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
43
     MPI_Wait(&reqs[0], &status);
44
     MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]);
45
     MPI_Wait(&regs[1], &status);
46
47
     defines the same ordering as:
48
```

```
1
int a = 4, b;
                                                                                             \mathbf{2}
MPI_File_open(MPI_COMM_WORLD, "myfile",
                                                                                             3
                MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                             4
MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status );
                                                                                             5
MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status );
                                                                                             6
Since
                                                                                             9
   • nonconcurrent operations on a single file handle are sequentially consistent, and
                                                                                             10
                                                                                             11
   • the program fragments specify an order for the operations,
                                                                                             12
MPI guarantees that both program fragments will read the value 4 into b. There is no need
                                                                                             13
to set atomic mode for this example.
                                                                                             14
    Similar considerations apply to conflicting accesses of the form:
                                                                                             15
                                                                                             16
MPI_File_iwrite_all(fh,...);
                                                                                             17
MPI_File_iread_all(fh,...);
                                                                                             18
MPI_Waitall(...);
                                                                                             19
    In addition, as mentioned in Section 13.6.5, nonblocking collective I/O operations have
                                                                                             20
to be called in the same order on the file handle by all processes.
                                                                                             21
    Similar considerations apply to conflicting accesses of the form:
                                                                                             22
                                                                                             23
MPI_File_write_all_begin(fh,...);
                                                                                             ^{24}
MPI_File_iread(fh,...);
                                                                                             25
MPI_Wait(fh,...);
                                                                                             26
MPI_File_write_all_end(fh,...);
                                                                                             27
                                                                                             28
    Recall that constraints governing consistency and semantics are not relevant to the
                                                                                             29
following:
                                                                                             30
MPI_File_write_all_begin(fh,...);
                                                                                             31
MPI_File_read_all_begin(fh,...);
                                                                                             32
                                                                                             33
MPI_File_read_all_end(fh,...);
                                                                                             34
MPI_File_write_all_end(fh,...);
                                                                                             35
since split collective operations on the same file handle may not overlap (see Section 13.4.5).
                                                                                             36
                                                                                             37
        I/O Error Handling
                                                                                             38
13.7
                                                                                             39
By default, communication errors are fatal — MPI_ERRORS_ARE_FATAL is the default error
                                                                                             40
handler associated with MPI_COMM_WORLD. I/O errors are usually less catastrophic (e.g.,
                                                                                             41
                                                                                             42
"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.
                                                                                             43
                                                                                             44
```

Advice to users.MPI does not specify the state of a computation after an erroneous45MPI call has occurred.A high-quality implementation will support the I/O error46handling facilities, allowing users to write programs using common practice for I/O.47(End of advice to users.)48

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.

8 I/O error handling differs from communication error handling in another important 9 aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN. 10 The default file error handler has two purposes: when a new file handle is created (by 11MPI_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 1213(e.g., MPI_FILE_OPEN or MPI_FILE_DELETE) use the default file error handler. The de-14fault file error handler can be changed by specifying MPI_FILE_NULL as the fh argument 15to MPI_FILE_SET_ERRHANDLER. The current value of the default file error handler can 16be determined by passing MPI_FILE_NULL as the fh argument to

¹⁷ 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.)

23 24 25

26 27

28

29

30

31 32

33 34

35 36

37

38

18 19

20

21

22

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().

```
39
      /*
40
       *
41
       *
         Function:
                                   double_buffer
42
       *
         Synopsis:
43
       *
44
       *
               void double_buffer(
45
                         MPI_File fh,
                                                                            ** IN
       *
46
       *
                         MPI_Datatype buftype,
                                                                            **
                                                                              ΙN
47
                          int bufcount
       *
                                                                           ** IN
48
               )
       *
```

1

 $\mathbf{2}$

		9
		10
MPI_ERR_FILE	Invalid file handle	11
MPI_ERR_NOT_SAME	Collective argument not identical on all	12
	processes, or collective routines called in	13
	a different order by different processes	14
MPI_ERR_AMODE	Error related to the amode passed to	15
	MPI_FILE_OPEN	16
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to	17
	MPI_FILE_SET_VIEW	18
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	19
	a file which supports sequential access only	20
MPI_ERR_NO_SUCH_FILE	File does not exist	21
MPI_ERR_FILE_EXISTS	File exists	22
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	23
MPI_ERR_ACCESS	Permission denied	24
MPI_ERR_NO_SPACE	Not enough space	25
MPI_ERR_QUOTA	Quota exceeded	26
MPI_ERR_READ_ONLY	Read-only file or file system	27
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	28
	the file is currently open by some process	29
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	30
	tered because a data representation identi-	31
	fier that was already defined was passed to	32
	MPI_REGISTER_DATAREP	33
MPI_ERR_CONVERSION	An error occurred in a user supplied data	34
	conversion function.	35
MPI_ERR_IO	Other I/O error	36
	,	37
Table 13.3	3: I/O Error Classes	38
		39
		40
		41

```
1
     *
\mathbf{2}
     * Description:
3
            Performs the steps to overlap computation with a collective write
     *
4
     *
            by using a double-buffering technique.
5
6
     * Parameters:
7
     *
            fh
                               previously opened MPI file handle
8
            buftype
                               MPI datatype for memory layout
     *
9
                               (Assumes a compatible view has been set on fh)
     *
10
     *
                            # buftype elements to transfer
            bufcount
11
     *-----*/
12
13
     /* this macro switches which buffer "x" is pointing to */
14
    #define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
15
16
    void double_buffer(MPI_File fh, MPI_Datatype buftype, int bufcount)
17
    {
18
19
                                /* status for MPI calls */
       MPI_Status status;
       float *buffer1, *buffer2; /* buffers to hold results */
20
21
       float *compute_buf_ptr; /* destination buffer */
22
                                  /* for computing */
23
       float *write_buf_ptr; /* source for writing */
^{24}
                                 /* determines when to quit */
       int done;
25
26
       /* buffer initialization */
27
       buffer1 = (float *)
28
                          malloc(bufcount*sizeof(float));
29
       buffer2 = (float *)
30
                          malloc(bufcount*sizeof(float));
^{31}
       compute_buf_ptr = buffer1; /* initially point to buffer1 */
32
       write_buf_ptr = buffer1; /* initially point to buffer1 */
33
34
35
       /* DOUBLE-BUFFER prolog:
36
            compute buffer1; then initiate writing buffer1 to disk
        *
37
        */
38
        compute_buffer(compute_buf_ptr, bufcount, &done);
39
       MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
40
41
       /* DOUBLE-BUFFER steady state:
42
        * Overlap writing old results from buffer pointed to by write_buf_ptr
43
        * with computing new results into buffer pointed to by compute_buf_ptr.
44
        *
45
        * There is always one write-buffer and one compute-buffer in use
46
        * during steady state.
47
        */
48
       while (!done) {
```

}

```
TOGGLE_PTR(compute_buf_ptr);
compute_buffer(compute_buf_ptr, bufcount, &done);
MPI_File_write_all_end(fh, write_buf_ptr, &status);
TOGGLE_PTR(write_buf_ptr);
MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
}
/* DOUBLE-BUFFER epilog:
    * wait for final write to complete.
    */
MPI_File_write_all_end(fh, write_buf_ptr, &status);
/* buffer cleanup */
free(buffer1);
free(buffer2);
```



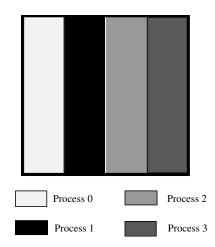


Figure 13.4: Example array file layout

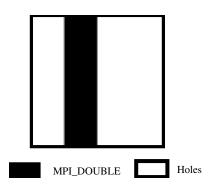


Figure 13.5: Example local array filetype for process 1

Unofficial Draft for Comment Only

1 2

3

4

5

6 7

8 9

10

11

12 13

14 15

16

17

18 19

32

33 34

 $46 \\ 47$

```
1
          Assume we are writing out a 100 \times 100 2D array of double precision floating point num-
\mathbf{2}
     bers that is distributed among 4 processes such that each process has a block of 25 columns
3
     (e.g., process 0 has columns 0–24, process 1 has columns 25–49, etc.; see Figure 13.4).
4
     To create the filetypes for each process one could use the following C program (see Sec-
\mathbf{5}
     tion 4.1.3):
6
         double subarray[100][25];
7
         MPI_Datatype filetype;
8
         int sizes[2], subsizes[2], starts[2];
9
10
         int rank;
11
         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12
         sizes[0]=100; sizes[1]=100;
13
         subsizes[0]=100; subsizes[1]=25;
14
         starts[0]=0; starts[1]=rank*subsizes[1];
15
16
         MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
17
                                      MPI_DOUBLE, &filetype);
18
19
          Or, equivalently in Fortran:
20
21
             double precision subarray(100,25)
22
              integer filetype, rank, ierror
23
             integer sizes(2), subsizes(2), starts(2)
24
25
             call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
26
             sizes(1)=100
27
             sizes(2)=100
28
             subsizes(1)=100
29
             subsizes(2)=25
30
             starts(1)=0
31
             starts(2)=rank*subsizes(2)
32
33
             call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
34
                          MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
                                                                                 &
35
                          filetype, ierror)
36
37
          The generated filetype will then describe the portion of the file contained within the
38
     process's subarray with holes for the space taken by the other processes. Figure 13.5 shows
39
     the filetype created for process 1.
40
41
42
43
44
45
46
47
```

Chapter 14

Deprecated Interfaces

14.1Deprecated since MPI-2.0

The following function is deprecated and is superseded by MPI_COMM_CREATE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 17.2.7. The language bindings are modified.

MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)

IN	copy_fn	Copy callback function for $keyval$ (function)
IN	delete_fn	Delete callback function for $keyval$ (function)
OUT	keyval	key value for future access (integer)
IN	extra_state	Extra state for callback functions

C binding

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn,
             MPI_Delete_function *delete_fn, int *keyval,
             void *extra_state)
```

```
F binding
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
   EXTERNAL COPY_FN, DELETE_FN
   INTEGER KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

The copy_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP. copy_fn should be of type MPI_Copy_function, which is defined as follows:

```
43
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                                                                                   44
                               void *extra_state, void *attribute_val_in,
                                                                                   45
                               void *attribute_val_out, int *flag)
                                                                                   46
```

A Fortran declaration for such a function is as follows: For this routine, an interface within the mpi_f08 module was never defined.

```
614
                                             CHAPTER 14. DEPRECATED INTERFACES
1
     SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
\mathbf{2}
                    ATTRIBUTE_VAL_OUT, FLAG, IERR)
3
          INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
4
          ATTRIBUTE_VAL_OUT, IERR
5
          LOGICAL FLAG
6
          copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or
7
     FORTRAN; MPI_NULL_COPY_FN is a function that does nothing other than returning
8
     flag = 0 and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag =
9
     1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note
10
     that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated.
11
          Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn
12
     function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call
13
     is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function,
14
     which is defined as follows:
15
16
     typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
17
                                         void *attribute_val, void *extra_state);
18
19
          A Fortran declaration for such a function is as follows:
20
     For this routine, an interface within the mpi_f08 module was never defined.
21
     SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
22
          INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
23
^{24}
         delete_fn may be specified as MPI_NULL_DELETE_FN from either C or FORTRAN;
25
     MPI_NULL_DELETE_FN is a function that does nothing, other than returning
26
     MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated.
27
          The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL
28
     in MPI-2.0. The language independent definition of the deprecated function is the same as
29
     of the new function, except of the function name. The language bindings are modified.
30
^{31}
32
     MPI_KEYVAL_FREE(keyval)
33
       INOUT
                 kevval
                                             Frees the integer key value (integer)
34
35
     C binding
36
     int MPI_Keyval_free(int *keyval)
37
38
     F binding
39
     MPI_KEYVAL_FREE(KEYVAL, IERROR)
40
          INTEGER KEYVAL, IERROR
41
          The following function is deprecated and is superseded by MPI_COMM_SET_ATTR in
42
     MPI-2.0. The language independent definition of the deprecated function is the same as of
43
     the new function, except of the function name. The language bindings are modified.
44
45
46
47
48
```

MPI_ATTF	R_PUT(comm, keyval, attribute	e_val)	1	
INOUT	comm	communicator to which attribute will be attached (han- dle)	2 3	
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE (integer)	4 5 6	
IN	attribute_val	attribute value	7	
C binding int MPI_A	•	nt keyval, void* attribute_val)	9 10	
F binding	-		11 12 13	
	ER COMM, KEYVAL, ATTRIBU		14	
MPI-2.0. 1	The language independent def	d and is superseded by MPI_COMM_GET_ATTR in nition of the deprecated function is the same as of name. The language bindings are modified.	15 16 17 18	
	CET (comme lossed attribute	vial flag)	19 20	
IN	R_GET(comm, keyval, attribute	_,	20	
	comm	communicator to which attribute is attached (handle)	22	
IN	keyval	key value (integer)	23	
OUT	attribute_val	attribute value, unless $flag = false$	24 25	
OUT	flag	true if an attribute value was extracted; false if no attribute is associated with the key (logical)	26 27	
C binding int MPI_Attr_get(MPI_Comm comm, int keyval, void* attribute_val, int *flag)				
F binding	C C	no noyvar, vorav aborrbabo_var, int virag,	30 31	
	GET(COMM, KEYVAL, ATTRIB	UTE VAL. FLAG. IERROR)	32	
INTEG	GER COMM, KEYVAL, ATTRIBU CAL FLAG		33 34	
The fo	ollowing function is deprecated	and is superseded by MPI_COMM_DELETE_ATTR	35	
		lefinition of the deprecated function is the same as	36 37	
		ion name. The language bindings are modified.	38 39	
MPI_ATTR_DELETE(comm, keyval) 40				
INOUT	comm	communicator to which attribute is attached (handle)	41 42	
IN	keyval	The key value of the deleted attribute (integer)	43	
			44	
C binding int MPI_Attr_delete(MPI_Comm comm, int keyval) 45				
F binding	F binding			

Unofficial Draft for Comment Only

```
1
     MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
\mathbf{2}
          INTEGER COMM, KEYVAL, IERROR
3
4
              Deprecated since MPI-2.2
\mathbf{5}
      14.2
6
      The entire set of C++ language bindings have been removed. See Chapter 15, Removed
\overline{7}
     Interfaces for more information.
8
          The following function typedefs have been deprecated and are superseded by new
9
      names. Other than the typedef names, the function signatures are exactly the same; the
10
      names were updated to match conventions of other function typedef names.
11
12
                             Deprecated Name
                                                                        New Name
13
                   MPI_Comm_errhandler_fn
                                                MPI_Comm_errhandler_function
14
                   MPI_File_errhandler_fn
                                                MPI_File_errhandler_function
15
                   MPI_Win_errhandler_fn
                                                MPI_Win_errhandler_function
16
17
      14.3
              Deprecated since MPI-3.2
18
19
      Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed
20
     in a future version of the MPI specification.
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 15

Removed Interfaces

15.1 Removed MPI-1 Bindings

15.1.1 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.

15.1.2 Removed MPI-1 Functions

Table 15.1 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 15.1: Removed MPI-1 functions and their replacements

15.1.3 Removed MPI-1 Datatypes

Table 15.2 shows the removed MPI-1 datatypes and their replacements.

15.1.4 Removed MPI-1 Constants

Table 15.3 shows the removed MPI-1 constants. There are no MPI-2 replacements.

 $44 \\ 45$

	618 CHAPTER 15. REMOVED INTERFACE	S
1	Removed MPI-2 Replacement	
2	MPI_LB MPI_TYPE_CREATE_RESIZED	
3	MPI_UB MPI_TYPE_CREATE_RESIZED	
4		
5		
6	Table 15.2: 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 13	MPI_COMBINER_STRUCT_INTEGER	
14		
15	Table 15.3: Removed MPI-1 constants	
16		
17 18	15.1.5 Removed MPI-1 Callback Prototypes	
19	Table 15.4 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 15.4: Removed MPI-1 callback prototypes and their replacements	
25		
26		
27	15.2 C++ Bindings	
28 29		
29 30	The $C++$ bindings were deprecated as of MPI-2.2. The $C++$ bindings are removed in the transmission of trans	
31	MPI-3.0. The namespace is still reserved, however, and bindings may only be provided by	у
32	an implementation as described in the MPI-2.2 standard.	
33		
34		
35		
36		
37		
38		
39		
40 41		
41		
43		
44		
45		
46		
47		
48		

Chapter 16				
Backward Incompatibilities				
16.1 Backward Incompatible since MPI-3.2 The default communicator where errors are raised when not involving a communicator, window, or file was changed from MPI_COMM_WORLD to MPI_COMM_SELF.				

Unofficial Draft for Comment Only

Chapter 17

Language Bindings

17.1 Fortran Support

17.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [40] + TS 29113 [41].

Rationale. Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 + TS 29113, the major new language features used are the ASYNCHRONOUS attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 17.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

- 1. USE mpi_f08: This method is described in Section 17.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.
- 2. USE mpi: This method is described in Section 17.1.3 and requires compile-time argument checking. Handles are defined as INTEGER. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.
- 3. **INCLUDE 'mpif.h':** This method is described in Section 17.1.4. The use of the include file mpif.h is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

Compliant MPI-3 implementations providing a Fortran interface must provide one or both of the following:

- The USE mpi_f08 Fortran support method.
- The USE mpi and INCLUDE 'mpif.h' Fortran support methods.

Section 17.1.6 describes restrictions if the compiler does not support all the needed features. Application subroutines and functions may use either one of the modules or the mpif.h include file. An implementation may require the use of one of the modules to prevent type mismatch errors.

Advice to users. Users are advised to utilize one of the MPI modules even if mpif.h enforces type checking on a particular system. Using a module provides several potential advantages over using an include file; the mpi_f08 module offers the most robust and complete Fortran support. (End of advice to users.)

In a single application, it must be possible to link together routines which USE mpi_f08, USE mpi, and INCLUDE 'mpif.h'.

The LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED is set to

19.TRUE. if all buffer choice arguments are defined in explicit interfaces with assumed-type 20and assumed-rank [41]; otherwise it is set to .FALSE.. The LOGICAL compile-time constant 21MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was 22added to the choice buffer arguments of all nonblocking interfaces and the underlying 23Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of 24 TS 29113), otherwise it is set to .FALSE.. These constants exist for each Fortran support 25method, but not in the C header file. The values may be different for each Fortran support 26method. All other constants and the integer values of handles must be the same for each 27Fortran support method. 28

Section 17.1.2 through 17.1.4 define the Fortran support methods. The Fortran in-29terfaces of each MPI routine are shorthands. Section 17.1.5 defines the corresponding 30 full interface specification together with the specific procedure names and implications for 31 the profiling interface. Section 17.1.6 the implementation of the MPI routines for differ-32 ent versions of the Fortran standard. Section 17.1.7 summarizes major requirements for 33 valid MPI-3.0 implementations with Fortran support. Section 17.1.8 and Section 17.1.9 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 17.1.10 through 17.1.19 give an overview and details on known prob-41 lems when using Fortran together with MPI; Section 17.1.20 compares the Fortran problems 42with those in C.

43 44

45

17.1.2 Fortran Support Through the mpi_f08 Module

An MPI implementation providing a Fortran interface must provide a module named mpi_f08
 that can be used in a Fortran program. Section 17.1.6 describes restrictions if the compiler
 does not support all the needed features. Within all MPI function specifications, the first

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4 5

6 7

8

9

10

11

12

13

14

15 16

17

of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments which are not TYPE(*), with the following exception:

Only one Fortran interface is defined for functions that are deprecated as of MPI-3.0. This interface must be provided as an explicit interface according to the rules defined for the mpi module, see Section 17.1.3.

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 17.1.6 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 17.1.6 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 17.1.3. (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.*)

17.1.3 Fortran Support Through the mpi Module

An MPI implementation providing a Fortran interface must provide a module named mpi that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists. 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 17.1.6 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 17.1.12 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 17.1.6 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 17.1.6 for details.

An MPI implementation may provide other features in the mpi module that enhance the usability of MPI while maintaining adherence to the standard. For example, it may provide INTENT information in these interface blocks.

- Advice to implementors. The appropriate INTENT may be different from what is given in the MPI language-neutral bindings. Implementations must choose INTENT so that the function adheres to the MPI standard, e.g., by defining the INTENT as provided in the mpi_f08 bindings. (End of advice to implementors.)
- Rationale. The intent given by the MPI generic interface is not precisely defined
 and does not in all cases correspond to the correct Fortran INTENT. For instance,
 receiving into a buffer specified by a datatype with absolute addresses may require
 associating MPI_BOTTOM with a dummy OUT argument. Moreover, "constants" such
 MPI_BOTTOM and MPI_STATUS_IGNORE are not constants as defined by Fortran,
 but "special addresses" used in a nonstandard way. Finally, the MPI-1 generic intent

Unofficial Draft for Comment Only

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23

24

25

26

27 28

29

30

31

32

33 34

35

36

37 38

39

40

41

was changed in several places in MPI-2. For instance, MPI_IN_PLACE changes the intent of an OUT argument to be INOUT. (End of rationale.)

Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 that "INTENT(OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument should retain its value rather than being redefined, INTENT(INOUT) should be used rather than INTENT(OUT), even if there is no explicit reference to the value of the dummy argument. Furthermore, INTENT(INOUT) is not equivalent to omitting the IN-TENT attribute, because INTENT(INOUT) always requires that the associated actual argument is definable." Applications that include mpif.h may not expect that 11 INTENT (OUT) is used. In particular, output array arguments are expected to keep their 12content as long as the MPI routine does not modify them. To keep this behavior, it is 13 recommended that implementations not use INTENT(OUT) in the mpi module and the 14mpif.h include file, even though INTENT(OUT) is specified in an interface description of the mpi_f08 module. (End of advice to implementors.)

Fortran Support Through the mpif.h Include File 17.1.4

The use of the mpif.h include file is strongly discouraged and may be deprecated in a future version of MPI.

An MPI implementation providing a Fortran interface must provide an include file named mpif.h that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is supported by this include file. This include file must:

- Define all named MPI constants. • Declare MPI functions that return a value. • Define all handles as INTEGER. • Be valid and equivalent for both fixed and free source form. For each MPI routine, an implementation can choose to use an implicit or explicit interface for the second Fortran binding (in deprecated routines, the first one may be omitted). • Set the LOGICAL compile-time constants MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING according to the same rules as for the mpi module. In the case of implicit interfaces for choice buffer or nonblocking routines, the constants must be set to .FALSE.. Advice to users. Instead of using mpif.h, the use of the mpi_f08 or mpi module is strongly encouraged for the following reasons: • Most mpif.h implementations do not include compile-time argument checking. • Therefore, many bugs in MPI applications remain undetected at compile-time,
 - such as:
 - Missing ierror as last argument in most Fortran bindings.

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

5

6

 $\overline{7}$

8

9

10

15

16 17 18

19

20

21

22

23

24

2526

2728

29

30 31

32 33

34

35

36

37

38

39 40

41

4243

44

45

46 47

	028 CHAPTER 17. LANGU	AGE DINDINGS
12	 Declaration of a status as an INTEGER variable instead of a with size MPI_STATUS_SIZE. 	an INTEGER array
$\frac{3}{4}$	- Incorrect argument positions; e.g., interchanging the cour	nt and
5	datatype arguments.– Passing incorrect MPI handles; e.g., passing a datatype inst	tood of a commu
6	- I assing incorrect wirt nancies, e.g., passing a datatype ins nicator.	tead of a commu-
7 8 9 10	• The migration from mpif.h to the mpi module should be related ward (i.e., substituting include 'mpif.h' after an implicit statement) as long as the application	statement by use
11 12 13	• Migrating portable and correctly written applications to the m expected to be difficult. No compile or runtime problems show an mpif.h include file was always allowed to provide explicit F	ild occur because
14 15	(End of advice to users.)	
16 17 18 19 20	Rationale. With MPI-3.0, the mpif.h include file was not depredered and the mpif.h and the mpimplemented so that essentially the same library implementation of can be used. (<i>End of rationale.</i>)	i module may be
21 22	17.1.5 Interface Specifications, Procedure Names, and the Profiling Interface Specifications, Procedure Names, Procedure	terface
23 24 25 26 27 28 29 30 31 32	The Fortran interface specification of each MPI routine specifies the routine be called by the application program, and the names and types of the du- together with additional attributes. The Fortran standard allows a given to be implemented with several methods, e.g., within or outside of a module BIND(C), or the buffers with or without TS 29113. Such implementation different binary interfaces and different specific procedure names. The several implementation schemes together with the rules for the specific and its implications for the profiling interface are specified within this sec implementation details.	ummy arguments Fortran interface e, with or without n decisions imply requirements for procedure names
33 34	<i>Rationale.</i> This section was introduced in MPI-3.0 on Sep. 21, 2012. for implementing the three Fortran support methods have been:	The major goals
35 36 37	• Portable implementation of the wrappers from the MPI Fortran MPI routines in C.	interfaces to the
38 39	• Binary backward compatible implementation path when swite MPI_SUBARRAYS_SUPPORTED from .FALSE. to .TRUE	ching
40 41 42 43	• The Fortran PMPI interface need not be backward compatible must be included that a tools layer can use to examine the M the specific procedure names and interfaces used.	,
44	• No performance drawbacks.	
45	• Consistency between all three Fortran support methods.	
46 47	• Consistent with Fortran $2008 + TS 29113$.	
48		

CHAPTER 17. LANGUAGE BINDINGS

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.
1B	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().
2A	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 extensions like !\$PRAGMA IGNORE_TKR, which provides a call-by-reference argument without type, kind, and dimension checking.
2B	MPI_ISEND_FTS	Fortran interface and arguments, as in Annex A.4, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with TYPE(*), DIMENSION().

Table 17.1: 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 17.1. Case is not significant in the names.

Note that for the deprecated routines in Section 14.1, 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 17.1. However, the MPI library may contain multiple implementation schemes from Table 17.1.

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_lsend. The specific procedure names for these PMPI_Xxxx procedures must be different from the specific procedure names for the MPI_Xxxx procedures and are not specified by this standard.

³⁵ A user-written or middleware profiling routine should provide the same specific Fortran ³⁶ procedure names and calling conventions, and therefore can interpose itself as the MPI ³⁷ library routine. The profiling routine can internally call the matching

PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments, choice buffer arguments, or that are attribute caching routines (

MPI_{COMM|WIN|TYPE}_{SET|GET}_ATTR). In this case, the profiling software should
 invoke the corresponding PMPI routine using the same Fortran support method as used in
 the calling application program, because the C, mpi_f08 and mpi callback prototypes are
 different or the meaning of the choice buffer or attribute_val arguments are different.

- ⁴⁵ Advice to users. Although for each support method and MPI routine (e.g.,
- ⁴⁶ MPI_ISEND in mpi_f08), multiple routines may need to be provided to intercept ⁴⁷ the specific procedures in the MPI library (e.g., MPI_Isend_f08 and MPI_Isend_f08ts), ⁴⁸ each profiling routine itself uses only one support method (e.g., mpi_f08) and calls

Unofficial Draft for Comment Only

7

8

the real MPI routine through the one PMPI routine defined in this support method (i.e., PMPI_lsend in this example). (*End of advice to users.*)

Advice to implementors. If all of the following conditions are fulfilled:

- the handles in the mpi_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a non-optional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as OPTIONAL,
- the internal argument passing mechanism for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

Advice to implementors. In the Fortran support method mpif.h, compile-time argument checking can be also implemented for all routines. For mpif.h, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that mpif.h must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the two longest subroutine declaration statements are

```
SUBROUTINE PMPI_Dist_graph_create_adjacent(a,b,c,d,e,f,g,h,i,j,k)
SUBROUTINE PMPI_Rget_accumulate(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 71 and 66 characters. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. The longest of such interface definitions is

INTERFACE PMPI_Rget_accumulate
SUBROUTINE PMPI_Rget_accumulate_fts(a,b,c,d,e,f,g,h,i,j,k,l,m,n)

with 70 characters. In principle, continuation lines would be possible in mpif.h (spaces in columns 73–131, & in column 132, and in column 6 of the continuation line) but this would not be valid if the source line length is extended with a compiler flag to 132 characters. Column 133 is also not available for the continuation character because lines longer than 132 characters are invalid with some compilers by default.

The longest specific procedure names are PMPI_Dist_graph_create_adjacent_f08 and PMPI_File_write_ordered_begin_f08ts both with 35 characters in the mpi_f08 module.

For example, the interface specifications together with the specific procedure names can be implemented with

Unofficial Draft for Comment Only

 31

 $46 \\ 47$

```
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. 14and profiling routines that are part of the application and have access to the appli-15cation data. With MPI-3.0, new Fortran interfaces and implementation schemes were 16 introduced that have several implications on how Fortran MPI routines are internally 17 implemented and optimized. For profiling layers, these schemes imply that several in-18 ternal interfaces with different specific procedure names may need to be intercepted, 19 as shown in the example code above. Therefore, for wrapper routines that are part 20of a Fortran application, it may be more convenient to make the name shift within 21the application, i.e., to substitute the call to the MPI routine (e.g., MPI_ISEND) by a 22call to a user-written profiling wrapper with a new name (e.g., X_MPI_ISEND) and to 23call the Fortran MPI_ISEND from this wrapper, instead of using the PMPI interface. 24 (End of advice to users.) 25

Advice to implementors. An implementation that provides a Fortran interface must provide a combination of MPI library and module or include file that uses the specific procedure names as described in Table 17.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

17.1.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

- For Fortran 77 with some extensions:
 - MPI identifiers may be up to 30 characters (31 with the profiling interface).
 - MPI identifiers may contain underscores after the first character.
 - An MPI subroutine with a choice argument may be called with different argument types.
 - Although not required by the MPI standard, the INCLUDE statement should be available for including mpif.h into the user application source code.

Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute addresses from MPI_ADDRESS and MPI_BOTTOM may cause problems if an address does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem is solved with MPI_GET_ADDRESS, but not for Fortran 77.)
45
46
47
48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 7

8 9

10

26

27

28

29

30 31

32 33

34

35

36 37

38

39 40

41

42

43

1	• For Fortran 90:
2	The major additional features that are needed from Fortran 90 are:
3 4	- The MODULE and INTERFACE concept.
5	- The KIND= and SELECTEDKIND concept.
6	- Fortran derived TYPEs and the SEQUENCE attribute.
7	- The OPTIONAL attribute for dummy arguments.
8 9	- Cray pointers, which are a non-standard compiler extension, are needed for the
10 11	use of MPI_ALLOC_MEM.
12	With these features, $MPI-1.1$ – $MPI-2.2$ can be implemented without restrictions.
13	MPI-3.0 can be implemented with some restrictions. The Fortran support methods
14	are abbreviated with $S1 = \text{the mpi_f08}$ module, $S2 = \text{the mpi}$ module, and $S3 = \text{the mpi}$
15	mpif.f include file. If not stated otherwise, restrictions exist for each method which prevent implementing the complete semantics of MPI-3.0.
16	prevent implementing the complete semantics of twirt 5.0.
17 18	$-$ MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non-
19	contiguous subarrays cannot be used as buffers in nonblocking routines, RMA,
20	or split-collective I/O.
21	- S1, S2, and S3 can be implemented, but for S1, only a preliminary implementa-
22	tion is possible.
23	- In this preliminary interface of S1, the following changes are necessary:
24	\ast TYPE(*), DIMENSION() is substituted by non-standardized extensions
25 26	like !\$PRAGMA IGNORE_TKR.
27	* The ASYNCHRONOUS attribute is omitted.
28	* PROCEDURE() callback declarations are substituted by EXTERNAL .
29	- The specific procedure names are specified in Section 17.1.5.
30	- Due to the rules specified in Section 17.1.5, choice buffer declarations should be
31 32	implemented only with non-standardized extensions like !\$PRAGMA IGNORE_TKR
33	(as long as $F2008+TS$ 29113 is not available).
34	In S2 and S3: Without such extensions, routines with choice buffers should be
35	provided with an implicit interface, instead of overloading with a different MPI function for each possible buffer type (as mentioned in Section 17.1.11). Such
36	overloading would also imply restrictions for passing Fortran derived types as
37	choice buffer, see also Section 17.1.15.
38	Only in S1: The implicit interfaces for routines with choice buffer arguments
39 40	imply that the ierror argument cannot be defined as OPTIONAL. For this reason,
40	it is recommended not to provide the mpi_f08 module if such an extension is not
42	available.
43	- The ASYNCHRONOUS attribute can not be used in applications to protect buffers
44	in nonblocking MPI calls $(S1-S3)$.
45	– The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE
46	routines is not available.
47	
48	

- In S1 and S2, the definition of the handle types (e.g., TYPE(MPI_Comm) and the status type TYPE(MPI_Status) must be modified: The SEQUENCE attribute must be used instead of BIND(C) (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the mpi_f08 module for Fortran 90 compilers. In this case, the mpi_f08 handle types and all routines, constants and types related to TYPE(MPI_Status) (see Section 17.2.5) are also not available in the mpi module and mpif.h.	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.	11 12
For Fortran 2003: The major features that are needed from Fortran 2003 are:	13 14 15
- Interoperability with C, i.e.,	16
* BIND(C) derived types.	17
* The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.	18 19
 The ability to define an ABSTRACT INTERFACE and to use it for PROCEDURE dummy arguments. 	20 21
 The ability to overload the operators .EQ. and .NE. to allow the comparison of derived types (used in MPI-3.0 for MPI handles). 	22 23
 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. 	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
 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. 	30 31 32 33 34
MPI-3.0 can be implemented with the following restrictions:	35
	36 37
- MPI_SUBARRAYS_SUPPORTED equals .FALSE	38
 For S1, only a preliminary implementation is possible. The following changes are necessary: 	39 40
* TYPE(*), DIMENSION() is substituted by non-standardized extensions	41

- * TYPE(*), DIMENSION(..) is substituted by non-standardized extensions like !\$PRAGMA IGNORE_TKR.
- The specific procedure names are specified in Section 17.1.5.
- With S1, the ASYNCHRONOUS is required as specified in the second Fortran interfaces. With S2 and S3 the implementation can also add this attribute if explicit interfaces are used.

1	- The ASYNCHRONOUS Fortran attribute can be used in applications to try to protect
2	buffers in nonblocking MPI calls, but the protection can work only if the compiler
3	is able to protect asynchronous Fortran I/O and makes no difference between such
4	asynchronous Fortran I/O and MPI communication.
5	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
6	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
7	be used only for Fortran types that are C compatible.
8	 The same restriction as for Fortran 90 applies if non-standardized extensions like
9	
10	!\$PRAGMA IGNORE_TKR are not available.
11	• For Fortran $2008 + TS 29113$ and later and
12	For Fortran 2003 + TS 29113:
13	The major feature that are needed from TS 29113 are:
14	TYDE (+) DIMENSION () is available
15	- TYPE(*), DIMENSION() is available.
16	- The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI com-
17	munication.
18	$-$ The array dummy argument of the <code>ISO_C_BINDING</code> intrinsic <code>C_F_POINTER</code> is not
19	restricted to Fortran types for which a corresponding type in C exists.
20	Using these features, MPI-3.0 can be implemented without any restrictions.
21	
22	$-$ With S1, MPI_SUBARRAYS_SUPPORTED equals .true The asynchronous at-
23	tribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR)
24	binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
25	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
26	be used for any Fortran type.
27	$-$ With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation
28	dependent. A high quality implementation will also provide
29	$MPI_SUBARRAYS_SUPPORTED ==.TRUE.$ and will use the
30	ASYNCHRONOUS attribute in the same way as in S1.
31	– If non-standardized extensions like !\$PRAGMA IGNORE_TKR are not available then
32	S2 must be implemented with TYPE(*), DIMENSION().
33	Advise to implementations. If MOL CUDADDAYC CUDDODTED FALGE the shellow
34	Advice to implementors. If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice
35	argument may be implemented with an explicit interface using compiler directives,
36	for example:
37	INTERFACE
38	SUBROUTINE MPI(buf,)
39	!DEC\$ ATTRIBUTES NO_ARG_CHECK :: buf
40	!\$PRAGMA IGNORE_TKR buf
41	!DIR\$ IGNORE_TKR buf
42	!IBM* IGNORE_TKR buf
43	REAL, DIMENSION(*) :: buf
44	! declarations of the other arguments
45	END SUBROUTINE
46 47	END INTERFACE
47	
	(End of advice to implementors)

(End of advice to implementors.)

17.1.7 Requirements on Fortran Compilers

 $\mathsf{MPI-3.0}$ (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

Advice to users. Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI libraries are shipped together with special compilation scripts (e.g., mpif90, mpicc) that set these options automatically. (End of advice to users.)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by MPI_GET_VERSION, if all the solutions described in Sections 17.1.11 through 17.1.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the mpi_f08 and mpi modules, and mpif.h) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 29113 [41] are available. This is required only for mpi_f08. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the mpi_f08 module with MPI_SUBARRAYS_SUPPORTED set to .FALSE..
- "Simply contiguous" arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the ASYNCHRONOUS attribute. See Section 17.1.12 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 661 and Section 17.1.8, and DD on page 662) solve the problems described in Section 17.1.17.		
5 6 7 8 9	• The problems with temporary data movement (described in detail in Section 17.1.18) 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 17.1.19) are resolved without any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in invoking MPI procedures.		
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	17.1.8 Additional Support for Fortran Register-Memory-Synchronization		
24 25 26 27	As described in Section 17.1.17, 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 35	<pre>F08 binding MPI_F_sync_reg(buf) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf</pre>		
36 37 38	<pre>F binding MPI_F_SYNC_REG(BUF)</pre>		
39 40 41 42 43	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.		
44 45 46 47	<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>)		

Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is explicitly allowed to implement this routine within the mpi_f08 module according to the definition for the mpi module or mpif.h to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (End of advice to implementors.)

Rationale. This routine is not defined with TYPE(*), DIMENSION(*), i.e., assumed size instead of assumed rank, because this would restrict the usability to "simply contiguous" arrays and would require overloading with another interface for scalar arguments. (End of rationale.)

If only a part of an array (e.g., defined by a subscript triplet) is Advice to users. used in a nonblocking routine, it is recommended to pass the whole array to MPI_F_SYNC_REG anyway to minimize the overhead of this no-operation call. Note that this routine need not be called if MPI_ASYNC_PROTECTS_NONBLOCKING is .TRUE. and the application fully uses the facilities of ASYNCHRONOUS arrays. (End of advice to users.)

Additional Support for Fortran Numeric Intrinsic Types 17.1.9

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI_INTEGER, MPI_REAL, MPI_INT, MPI_DOUBLE, etc., as well as the optional types MPI_REAL4, MPI_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL, and 2728CHARACTER) with an optional integer KIND parameter that selects from among one or more 29variants. The specific meaning of different KIND values themselves are implementation 30 dependent and not specified by the language. Fortran provides the KIND selection functions 31selected_real_kind for REAL and COMPLEX types, and selected_int_kind for INTEGER types that allow users to declare variables with a minimum precision or number of digits. 33 These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX, and 34 INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE 35 36 PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two 37 declarations are equivalent:

double precision x real(KIND(0.0d0)) x

41 MPI provides two orthogonal methods for handling communication buffers of numeric 42intrinsic types. The first method (see the following section) can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained 4344with the selected_int_kind or selected_real_kind functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation 4546conversion in heterogeneous environments. The second method (see "Support for size-47specific MPI Datatypes" on page 644) gives the user complete control over communication 48 by exposing machine representations.

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

 $\mathbf{5}$

6

 $\overline{7}$

8

9

10

11

1213

14

15

16

17

18

1920

21

22

23 24

25

26

32

38

39

1 2

Parameterized Datatypes with Specified Precision and Exponent Range

MPI provides named datatypes corresponding to standard Fortran 77 numeric types: 3

MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and 4

MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides rep-5resentation conversion in heterogeneous environments. The mechanism described in this 6 section extends this model to support portable parameterized numeric types. 7

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.)
- 23Advice to users. selected_real_kind() maps a large number of (p,r) pairs to a 24much 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 26view 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. 28The corresponding MPI datatypes match if and only if they have the same (p,r) value 29 (REAL and COMPLEX) or r value (INTEGER). Thus MPI has many more datatypes than 30 there are fundamental language types. (End of advice to users.)
- 3132

19

20

21

22

25

27

```
33
```

3435

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

IN	р	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI datatype (handle)

```
40
     C binding
```

41 int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)

```
42
     F08 binding
43
```

```
MPI_Type_create_f90_real(p, r, newtype, ierror)
44
         INTEGER, INTENT(IN) :: p, r
45
```

```
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
46
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
```

```
48
     F binding
```

MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR

This function returns a predefined MPI datatype that matches a REAL variable of KIND selected_real_kind(p, r). In the model described above it returns a handle for the element D(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. In communication, an MPI datatype A returned by MPI_TYPE_CREATE_F90_REAL matches a datatype B if and only if B was returned by MPI_TYPE_CREATE_F90_REAL called with the same values for 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 643.

It is erroneous to supply values for p and r not supported by the compiler.

MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)

IN	р	precision, in decimal digits (integer)
IN	r	decimal exponent range (integer)
OUT	newtype	the requested MPI data type (handle)

C binding

int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)

F08 binding

<pre>MPI_Type_create_f90_complex(p, r, newtype, ierror)</pre>
INTEGER, INTENT(IN) :: p, r
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

F binding

```
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
INTEGER P, R, NEWTYPE, IERROR
```

This function returns a predefined MPI datatype that matches a COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 643.

It is erroneous to supply values for \boldsymbol{p} and \boldsymbol{r} not supported by the compiler.

MPI_TYPE_CREATE_F90_INTEGER(r, newtype) 42 IN r decimal exponent range, i.e., number of decimal digits 43 OUT newtype the requested MPI datatype (handle) 44 47 47 46 47 47 47

C binding

 $\mathbf{2}$

```
1
     int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
2
     F08 binding
3
     MPI_Type_create_f90_integer(r, newtype, ierror)
4
          INTEGER, INTENT(IN) :: r
5
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
6
          INTEGER, OPTIONAL, INTENT(OUT) ::
                                                 ierror
7
8
     F binding
9
     MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
10
          INTEGER R, NEWTYPE, IERROR
11
         This function returns a predefined MPI datatype that matches a INTEGER variable of
12
     KIND selected_int_kind(r). Matching rules for datatypes created by this function are
13
     analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL.
14
     Restrictions on using the returned datatype with the "external 32" data representation are
15
     given on page 643.
16
         It is erroneous to supply a value for r that is not supported by the compiler.
17
         Example:
18
19
         integer
                        longtype, quadtype
20
         integer, parameter :: long = selected_int_kind(15)
21
         integer(long) ii(10)
22
        real(selected_real_kind(30)) x(10)
23
         call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
24
         call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
25
         . . .
26
27
         call MPI_SEND(ii, 10, longtype, ...)
28
         call MPI_SEND(x, 10, quadtype, ...)
29
30
           Advice to users.
                              The datatypes returned by the above functions are predefined
31
          datatypes. They cannot be freed; they do not need to be committed; they can be
32
          used with predefined reduction operations. There are two situations in which they
33
          behave differently syntactically, but not semantically, from the MPI named predefined
34
          datatypes.
35
            1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to
36
               retrieve the values of p and r.
37
38
            2. Because the datatypes are not named, they cannot be used as compile-time
39
               initializers or otherwise accessed before a call to one of the
40
               MPI_TYPE_CREATE_F90_XXX routines.
41
          If a variable was declared specifying a non-default KIND value that was not obtained
42
           with selected_real_kind() or selected_int_kind(), the only way to obtain a
43
          matching MPI datatype is to use the size-based mechanism described in the next
44
          section.
45
46
           (End of advice to users.)
47
48
```

Advice to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/ INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (*End of advice to implementors.*)

Rationale. The MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 13.5.2) 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	(p > 33) or (r	> 4931) then external32 represe	ntation			
		is undefined				
else if	(p > 15) or (r	> 307) then external32_size =	16			
else if	(p > 6) or (r	> 37) then external32_size =	8			
else		external32_size =	4			
For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for MPI_TYPE_CREATE_F90_REAL. For MPI_TYPE_CREATE_F90_INTEGER:						
if else if		<pre>external32 representation is und external32_size = 16</pre>	efined			

		\ - ·	/		0	
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 external 32 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) 46 in operations that require the external 32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL, and many MPI_FILE functions, 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

35

36

when the "external32" data representation is used. The ranges for which the external32
 representation is undefined are reserved for future standardization.

3 4

5

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.

⁹ We assume that for each **typeclass** (integer, real, complex) and each word size there is ¹⁰ a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler, ¹¹ MPI must provide a named size-specific datatype. The name of this datatype is of the form ¹² MPI_<TYPE>n in C and Fortran where <TYPE> is one of REAL, INTEGER and COMPLEX, ¹³ and **n** is the length in bytes of the machine representation. This datatype locally matches ¹⁴ all variables of type (**typeclass**, **n**) in Fortran. The list of names for such types includes:

- ¹⁵ MPI_REAL4
- ¹⁶ MPI_REAL8
- ¹⁷ MPI_REAL16
- ¹⁸ MPI_COMPLEX8
- MPI_COMPLEX16
- MPI_COMPLEX32
- MPI_INTEGER1
- MPI_INTEGER2
- MPI_INTEGER4
- MPI_INTEGER8
- MPI_INTEGER16

²⁷ One datatype is required for each representation supported by the Fortran compiler.

Rationale. Particularly for the longer floating-point types, C and Fortran may use different representations. For example, a Fortran compiler may define a 16-byte REAL type with 33 decimal digits of precision while a C compiler may define a 16-byte long double type that implements an 80-bit (10 byte) extended precision floating point value. Both of these types are 16 bytes long, but they are not interoperable. Thus, these types are defined by Fortran, even though C may define types of the same length. (End of rationale.)

35 36 37

38

39

40

41

42

43

28

29

30

31

32

33 34

> To 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 representation is of size n. These datatypes may also be used for variables declared with KIND=INT8/16/32/64 or KIND=REAL32/64/128, which are defined in the ISO_FORTRAN_ENV intrinsic module. Note that the MPI datatypes and the REAL*n, INTEGER*n declarations count bytes whereas the Fortran KIND values count bits. All these datatypes are predefined. The following functions allow a user to obtain a size-specific MPI datatype for any

⁴⁴ intrinsic Fortran type.

- 45
- 46
- 47
- 48

MPI_SIZE	EOF(x, size)		1
IN	x	a Fortran variable of numeric intrinsic type (choice)	2
OUT	size	size of machine representation of that type (integer)	$\frac{3}{4}$
001		the of machine representation of that type (moger)	4 5
F08 bine	ding		6
	eof(x, size, ierror)		7
	C(*), DIMENSION() :: x CGER, INTENT(OUT) :: size		8 9
	GER, OPTIONAL, INTENT(OUT		10
F bindir			11
	COF(X, SIZE, IERROR)		12
• 1	be> X(*)		13 14
INTE	CGER SIZE, IERROR		15
		bytes of the machine representation of the given	16
variable.	It is a generic Fortran routine	and has a Fortran binding only.	17 18
Adu	vice to users. This function is	similar to the C <i>sizeof</i> operator but behaves slightly	19
		ument, it returns the size of the base element, not	20
the	size of the whole array. (End	of advice to users.)	21
Rat	<i>ionale.</i> This function is not a	vailable in other languages because it would not be	22 23
	ful. (End of rationale.)	5 5	24
			25
			26
MPI_TYF	PE_MATCH_SIZE(typeclass, siz	e, datatype)	27 28
IN	typeclass	generic type specifier (integer)	29
IN	size	size, in bytes, of representation (integer)	30
OUT	datatype	datatype with correct type, size (handle)	31 32
			33
C bindii	0		34
		lass, int size, MPI_Datatype *datatype)	35
F08 bine			36 37
• -	e_match_size(typeclass, si GER, INTENT(IN) :: typec		38
	C(MPI_Datatype), INTENT(OU		39
INTE	CGER, OPTIONAL, INTENT(OUT) :: ierror	40
F bindir	ıg		41 42
	E_MATCH_SIZE(TYPECLASS, SI		43
INTE	CGER TYPECLASS, SIZE, DATA	TYPE, IERROR	44
		S_{REAL} , MPI_TYPECLASS_INTEGER and	45
	ý –	ing to the desired typeclass . The function returns	$46 \\ 47$
an MPI d	atatype matching a local varia	tble of type ($\mathbf{typeclass}, \mathbf{size}$).	48

```
1
          This function returns a reference (handle) to one of the predefined named datatypes, not
\mathbf{2}
      a duplicate. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to obtain a
3
      size-specific type that matches a Fortran numeric intrinsic type by first calling MPI_SIZEOF
4
      in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find
\mathbf{5}
      a suitable datatype. In C, one can use the C function sizeof(), instead of MPI_SIZEOF.
6
      In addition, for variables of default kind the variable's size can be computed by a call to
\overline{7}
      MPI_TYPE_GET_EXTENT, if the typeclass is known. It is erroneous to specify a size not
8
      supported by the compiler.
9
           Rationale. This is a convenience function. Without it, it can be tedious to find the
10
           correct named type. See note to implementors below. (End of rationale.)
11
12
           Advice to implementors. This function could be implemented as a series of tests.
13
14
           int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
15
           {
16
             switch(typeclass) {
17
                  case MPI_TYPECLASS_REAL: switch(size) {
18
                    case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
19
                    case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
20
                    default: error(...);
21
                  }
22
                  case MPI_TYPECLASS_INTEGER: switch(size) {
23
                      case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
24
                      case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
25
                      default: error(...);
26
                  }
27
                 ... etc. ...
28
               }
29
30
               return MPI_SUCCESS;
31
           }
32
33
           (End of advice to implementors.)
34
35
      Communication With Size-specific Types
36
37
      The usual type matching rules apply to size-specific datatypes: a value sent with datatype
38
      MPI_{TYPE>n} can be received with this same datatype on another process. Most modern
39
      computers use 2's complement for integers and IEEE format for floating point. Thus, com-
40
      munication using these size-specific datatypes will not entail loss of precision or truncation
41
      errors.
42
43
           Advice to users. Care is required when communicating in a heterogeneous environ-
44
           ment. Consider the following code:
45
46
           real(selected_real_kind(5)) x(100)
47
           call MPI_SIZEOF(x, size, ierror)
48
           call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
```

CHAPTER 17. LANGUAGE BINDINGS

```
if (myrank .eq. 0) then
    ... initialize x ...
    call MPI_SEND(x, xtype, 100, 1, ...)
else if (myrank .eq. 1) then
    call MPI_RECV(x, xtype, 100, 0, ...)
                                                                              5
                                                                              6
endif
                                                                              7
```

This may not work in a heterogeneous environment if the value of size is not the same on process 1 and process 0. There should be no problem in a homogeneous environment. To communicate in a heterogeneous environment, there are at least four options. The first is to declare variables of default type and use the MPI datatypes for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second is to use selected_real_kind or selected_int_kind and with the functions of the previous section. The third is to declare a variable that is known to be the same size on all architectures (e.g., selected_real_kind(12) on almost all compilers will result in an 8-byte representation). The fourth is to carefully check representation size before communication. This may require explicit conversion to a variable of size that can be communicated and handshaking between sender and receiver to agree on a size.

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
real(selected_real_kind(5)) x(100)
                                                                              24
call MPI_SIZEOF(x, size, ierror)
                                                                              25
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
                                                                               26
                                                                              27
if (myrank .eq. 0) then
                                                                              28
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
                                                              &
                                                                              29
                       MPI_MODE_CREATE+MPI_MODE_WRONLY,
                                                              &
                                                                              30
                       MPI_INFO_NULL, fh, ierror)
                                                                               31
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
                                                                               32
                           MPI_INFO_NULL, ierror)
                                                                               33
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
                                                                              34
   call MPI_FILE_CLOSE(fh, ierror)
                                                                              35
endif
                                                                              36
                                                                              37
call MPI_BARRIER(MPI_COMM_WORLD, ierror)
                                                                              38
                                                                              39
if (myrank .eq. 1) then
                                                                               40
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY, &
                                                                              41
                  MPI_INFO_NULL, fh, ierror)
                                                                              42
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32', &
                                                                               43
                           MPI_INFO_NULL, ierror)
                                                                               44
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
                                                                               45
   call MPI_FILE_CLOSE(fh, ierror)
                                                                               46
endif
                                                                               47
```

1

 $\mathbf{2}$

3

4

8

9 10

11

1213

1415

16

17

18

19

20

21

22 23

If processes 0 and 1 are on different machines, this code may not work as expected if the size is different on the two machines. (*End of advice to users.*)

2 3

5

6

7

8

16 17

18

19

20

21

22

23 24

25

26

27

28 29

30

 31

32

33

1

⁴ 17

17.1.10 Problems With Fortran Bindings for MPI

This section discusses a number of problems that may arise when using MPI in a Fortran program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It is intended to clarify, not add to, this standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these may cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. With Fortran 2008 and the new semantics defined in TS 29113, most violations are resolved, and this is hinted at in an addendum to each item. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail.

The following MPI features are inconsistent with Fortran 90 and Fortran 77.

- 1. An MPI subroutine with a choice argument may be called with different argument types. When using the mpi_f08 module together with a compiler that supports Fortran 2008 + TS 29113, this problem is resolved.
 - 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(..).
- 3. Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied on entrance to or exit from the subroutine. This problem is solved with the use of the ASYNCHRONOUS attribute.
- 4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the ASYNCHRONOUS attribute as specified in TS 29113.
- 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.
- 39 6. The memory allocation routine MPI_ALLOC_MEM cannot be used from 40 Fortran 77/90/95 without a language extension (for example, Cray pointers) that 41 allows the allocated memory to be associated with a Fortran variable. Therefore, 42address sized integers were used in MPI-2.0 – MPI-2.2. In Fortran 2003, 43 TYPE (C_PTR) entities were added, which allow a standard-conforming implementation 44 of the semantics of MPI_ALLOC_MEM. In MPI-3.0 and later, MPI_ALLOC_MEM has 45an additional, overloaded interface to support this language feature. The use of Cray 46 pointers is deprecated. The mpi_f08 module only supports TYPE(C_PTR) pointers. 47
- ⁴⁸ Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.

Unofficial Draft for Comment Only

- MPI identifiers exceed 6 characters.
- MPI identifiers may contain underscores after the first character.
- MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.
- Many routines in MPI have KIND-parameterized integers (e.g., MPI_ADDRESS_KIND and MPI_OFFSET_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, INTEGER*8 or INTEGER should be used instead.

MPI-1 contained several routines that take address-sized information as input or return address-sized information as output. In C such arguments were of type MPI_Aint and in Fortran of type INTEGER. On machines where integers are smaller than addresses, these routines can lose information. In MPI-2 the use of these functions has been deprecated and they have been replaced by routines taking INTEGER arguments of KIND=MPI_ADDRESS_KIND. A number of new MPI-2 functions also take INTEGER arguments of non-default KIND. See Section 2.6 and Section 4.1.1 for more information.

Sections 17.1.11 through 17.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 17.1.7.

17.1.11 Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 17.1.6). 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 do nmpi 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 :: 47 DIMS(*) and LOGICAL :: PERIODS(*). 48

Unofficial Draft for Comment Only

 $\overline{7}$

```
1
       USE mpi_f08
                           ! or USE mpi
\mathbf{2}
        INTEGER size
3
        CALL MPI_Cart_create(comm_old, 1, size, .TRUE., .TRUE., comm_cart, ierror)
4
     Although this is a non-conforming MPI call, compiler warnings are not expected (but may
5
6
     occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit
     interfaces.
7
8
9
     17.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets
10
     Arrays with subscript triplets describe Fortran subarrays with or without strides, e.g.,
11
12
         REAL a(100,100,100)
13
         CALL MPI_Send(a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL, ...)
14
15
     The handling of subscript triplets depends on the value of the constant
16
     MPI_SUBARRAYS_SUPPORTED:
17
18
         • If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:
19
           Choice buffer arguments are declared as TYPE(*), DIMENSION(...). For example,
20
           consider the following code fragment:
21
22
               REAL s(100), r(100)
23
               CALL MPI_Isend(s(1:100:5), 3, MPI_REAL, ..., rq, ierror)
24
               CALL MPI_Wait(rq, status, ierror)
25
               CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL, ..., rq, ierror)
26
               CALL MPI_Wait(rq, status, ierror)
27
28
           In this case, the individual elements s(1), s(6), and s(11) are sent between the start
29
           of MPI_ISEND and the end of MPI_WAIT even though the compiled code will not copy
30
           s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code
31
           will pass a descriptor to MPI_ISEND that allows MPI to operate directly on s(1), s(6),
32
           s(11), \ldots, s(96). The called MPI_ISEND routine will take only the first three of these
33
           elements due to the type signature "3, MPI_REAL".
34
35
           All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT,
36
           MPI_FILE_WRITE_ALL_BEGIN) behave as if the user-specified elements of choice
37
           buffers are copied to a contiguous scratch buffer in the MPI runtime environment.
38
           All datatype descriptions (in the example above, "3, MPI_REAL") read and store
39
           data from and to this virtual contiguous scratch buffer. Displacements in MPI de-
40
           rived datatypes are relative to the beginning of this virtual contiguous scratch buffer.
41
           Upon completion of a nonblocking receive operation (e.g., when MPI_WAIT on a cor-
42
           responding MPI_Request returns), it is as if the received data has been copied from
43
           the virtual contiguous scratch buffer back to the non-contiguous application buffer.
44
           In the example above, r(1), r(6), and r(11) are guaranteed to be defined with the
45
           received data when MPI_WAIT returns.
46
           Note that the above definition does not supercede restrictions about buffers used with
47
           non-blocking operations (e.g., those specified in Section 3.7.2).
48
```

Advice to implementors. The Fortran descriptor for TYPE(*), DIMENSION(...) arguments contains enough information that, if desired, the MPI library can make a real contiguous copy of non-contiguous user buffers when the nonblocking operation is started, and release this buffer not before the nonblocking communication has completed (e.g., the MPI_WAIT routine). Efficient implementations may avoid such additional memory-to-memory data copying. (End of advice to implementors.)

Rationale. If MPI_SUBARRAYS_SUPPORTED equals .TRUE., non-contiguous buffers are handled inside the MPI library instead of by the compiler through argument association conventions. Therefore, the scope of MPI library scratch buffers can be from the beginning of a nonblocking operation until the completion of the operation although beginning and completion are implemented in different routines. (*End of rationale.*)

• If MPI_SUBARRAYS_SUPPORTED equals .FALSE.:

In this case, the use of Fortran arrays with subscript triplets as actual choice buffer arguments in any nonblocking MPI operation (which also includes persistent request, and split collectives) may cause undefined behavior. They may, however, be used in blocking MPI operations.

Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

In Fortran, array data is not necessarily stored contiguously. For example, the array section A(1:N:2) involves only the elements of A with indices 1, 3, 5, The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., B(N)) or is of assumed size (e.g., B(*)). If necessary, they do this by making a copy of the array into contiguous memory.¹

Because MPI dummy buffer arguments are assumed-size arrays if MPI_SUBARRAYS_SUPPORTED equals .FALSE., this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

```
real a(100)
call MPI_IRECV(a(1:100:2), MPI_REAL, 50, ...)
```

Since the first dummy argument to MPI_IRECV is an assumed-size array (<type> buf(*)), the array section a(1:100:2) is copied to a temporary before being passed to MPI_IRECV, so that it is contiguous in memory. MPI_IRECV returns immediately, and data is copied from the temporary back into the array a. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem

 $\overline{7}$

 24

 31

¹Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the CONTIGUOUS attribute.

1	for MPI_ISEND since the temporary array may be deallocated before the data has all
2	been sent from it.
3	
4	Most Fortran 90 compilers do not make a copy if the actual argument is the whole
5	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.)
6	Also, many compilers treat allocatable arrays the same as they treat explicit-shape
7	arrays in this regard (though we know of one that does not). However, the same is not
8	true for assumed-shape and pointer arrays; since they may be discontiguous, copying
9	
10	is often done. It is this copying that causes problems for MPI as described in the
11	previous paragraph.
12	According to the Fortran 2008 Standard, Section 6.5.4, a "simply contiguous" array
	section is
13	Section 15
14	
15	<pre>name ([:,] [<subscript>]:[<subscript>] [,<subscript>])</subscript></subscript></subscript></pre>
16	
17	That is, there are zero or more dimensions that are selected in full, then one dimension
18	selected without a stride, then zero or more dimensions that are selected with a simple
19	subscript. The compiler can detect from analyzing the source code that the array is
	contiguous. Examples are
20	
21	A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
22	
23	Because of Fortran's column-major ordering, where the first index varies fastest, a
24	"simply contiguous" section of a contiguous array will also be contiguous.
25	simply contiguous section of a contiguous array will also be contiguous.
26	The same problem can occur with a scalar argument. A compiler may make a copy of
27	scalar dummy arguments within a called procedure when passed as an actual argument
28	to a choice buffer routine. That this can cause a problem is illustrated by the example
29	
30	real :: a
31	call user1(a,rq)
32	-
33	call MPI_WAIT(rq,status,ierr)
34	write (*,*) a
35	
36	<pre>subroutine user1(buf,request)</pre>
	<pre>call MPI_IRECV(buf,,request,)</pre>
37	end
38	
39	If a is copied, MPI_IRECV will alter the copy when it completes the communication
40	and will not alter a itself.
41	
42	Note that copying will almost certainly occur for an argument that is a non-trivial
43	expression (one with at least one operator or function call), a section that does not
44	select a contiguous part of its parent (e.g., A(1:n:2)), a pointer whose target is such
45	a section, or an assumed-shape array that is (directly or indirectly) associated with
	such a section.
46	
47	If a compiler option exists that inhibits copying of arguments, in either the calling or
48	called procedure, this must be employed.

Unofficial Draft for Comment Only

If a compiler makes copies in the calling procedure of arguments that are explicitshape or assumed-size arrays, "simply contiguous" array sections of such arrays, or scalars, and if no compiler option exists to inhibit such copying, then the compiler cannot be used for applications that use MPI_GET_ADDRESS, or any nonblocking MPI routine. If a compiler copies scalar arguments in the called procedure and there is no compiler option to inhibit this, then this compiler cannot be used for applications that use memory references across subroutine calls as in the example above.

17.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts

Fortran arrays with **vector** subscripts describe subarrays containing a possibly irregular set of elements

```
REAL a(100)
CALL MPI_Send(A((/7,9,23,81,82/)), 5, MPI_REAL, ...)
```

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.

17.1.14 Special Constants

MPI requires a number of special "constants" that cannot be implemented as normal Fortran constants, e.g., MPI_BOTTOM. The complete list can be found in Section 2.5.4. In C, these are implemented as constant pointers, usually as NULL and are used where the function prototype calls for a pointer to a variable, not the variable itself.

In Fortran, using special values for the constants (e.g., by defining them through **parameter** statements) is not possible because an implementation cannot distinguish these values from valid data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, the address of the actual choice buffer argument can be compared with the address of such a predefined static variable.

These special constants also cause an exception with the usage of Fortran INTENT: with USE mpi_f08, the attributes INTENT(IN), INTENT(OUT), and INTENT(INOUT) are used in the Fortran interface. In most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for dummy arguments that may be modified and allow one of these special constants as input, an INTENT is not specified.

17.1.15 Fortran Derived Types

MPI supports passing Fortran entities of BIND(C) and SEQUENCE derived types to choice dummy arguments, provided no type component has the ALLOCATABLE or POINTER attribute.

The following code fragment shows some possible ways to send scalars or arrays of interoperable derived type in Fortran. The example assumes that all data is passed by address.

```
type, BIND(C) :: mytype
integer :: i
real :: x
```

 24

```
1
            double precision :: d
2
            logical :: 1
3
         end type mytype
4
5
         type(mytype) :: foo, fooarr(5)
6
         integer :: blocklen(4), type(4)
7
         integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent
8
9
         call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
10
         call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
11
         call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
12
         call MPI_GET_ADDRESS(foo%1, disp(4), ierr)
13
14
         base = disp(1)
15
         disp(1) = disp(1) - base
16
         disp(2) = disp(2) - base
17
         disp(3) = disp(3) - base
18
         disp(4) = disp(4) - base
19
20
         blocklen(1) = 1
21
         blocklen(2) = 1
22
         blocklen(3) = 1
23
         blocklen(4) = 1
^{24}
25
         type(1) = MPI_INTEGER
26
         type(2) = MPI_REAL
27
         type(3) = MPI_DOUBLE_PRECISION
28
         type(4) = MPI_LOGICAL
29
30
         call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
31
         call MPI_TYPE_COMMIT(newtype, ierr)
32
33
         call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
34
         ! or
35
         call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
36
         ! expects that base == address(foo%i) == address(foo)
37
38
         call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
39
         call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
40
         extent = disp(2) - disp(1)
41
         1b = 0
42
         call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
43
         call MPI_TYPE_COMMIT(newarrtype, ierr)
44
45
         call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
46
         Using the derived type variable foo instead of its first basic type element foo%i may
47
```

 $_{47}$ be impossible if the MPI library implements choice buffer arguments through overloading

instead of using TYPE(*), DIMENSION(..), or through a non-standardized extension such as !\$PRAGMA IGNORE_TKR; see Section 17.1.6.

To use a derived type in an array requires a correct extent of the datatype handle to take care of the alignment rules applied by the compiler. These alignment rules may imply that there are gaps between the components of a derived type, and also between the subsuquent elements of an array of a derived type. The extent of an interoperable derived type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may be different because C and Fortran may apply different alignment rules. As recommended in the advice to users in Section 4.1.6, one should add an additional fifth structure element with one numerical storage unit at the end of this structure to force in most cases that the array of structures is contiguous. Even with such an additional element, one should 12keep this resizing due to the special alignment rules that can be used by the compiler for structures, as also mentioned in this advice.

Using the extended semantics defined in TS 29113, it is also possible to use entities or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer arguments; some additional constraints must be observed, e.g., no ALLOCATABLE or POINTER type components may exist. In this case, the **base** address in the example must be changed to become the address of foo instead of foo%i, because the Fortran compiler may rearrange type components or add padding. Sending the structure foo should then also be performed by providing it (and not foo%i) as actual argument for MPI_Send.

17.1.16 Optimization Problems, an Overview

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI_IRECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. These problems are independent of the Fortran support method; i.e., they occur with the mpi_f08 module, the mpi module, and the mpif.h include file.

This section shows four problematic usage areas (the abbreviations in parentheses are used in the table below):

- Use of nonblocking routines or persistent requests (Nonbl.).
- Use of one-sided routines (1-sided).
- Use of MPI parallel file I/O split collective operations (Split).
- Use of MPI_BOTTOM together with absolute displacements in MPI datatypes, or relative displacements between two variables in such datatypes (Bottom).

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 17.1.17.
- Temporary data movement and temporary memory modifications; see Section 17.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 17.1.19.

Unofficial Draft for Comment Only

1 $\mathbf{2}$

3

4

 $\mathbf{5}$

6 $\overline{7}$

8 9

10 11

13

14

15

16

17

18

1920

2122

23

 24

25

26

27

28

29

30

31

32

33 34

35

36 37

38 39

40

41 42

43 44

4546

1	Optimization .			nay cause	-	
2 3			to Nonbl.	llowing u 1-sided	sage are Split	eas Bottom
4	Cada maraman	4			_	1
5	Code movemen and register op		yes	yes	no	yes
6	Temporary dat	a movement	yes	yes	yes	no
7	Permanent data	a movement	yes	yes	yes	yes
8 9 10 11	Table 17.2: Occurrence	of Fortran of	ptimizatio	on probler	ns in se	everal usage areas
12 13 14	Table 17.2 shows the only us The solutions in the follo	-		-	-	
15 16 17	• to minimize the burder "Solutions" through " pages 658–663,		-	-	. –	
18 19	• to minimize the drawba	acks on comp	iler based	optimiza	tion, ar	nd
20 21	• to minimize the require	ements define	d in Secti	on 17.1.7.		
22 23	17.1.17 Problems with Coo	le Movement	and Reg	ister Opt	imizatio	on
24	Nonblocking Operations					
25 26 27 28 29 30	If a variable is local to a For compiler will assume that it of argument of the call. In the to save and restore certain re- held a valid copy of such a variable.	cannot be mo most commo gisters. Thus	dified by a n linkage s, the opti	a called su convention mizer wil	ibroutii on, the l assum	ne unless it is an actual subroutine is expected e that a register which
31 32	Example 17.1 Fortran 90	register optim	nization –	– extreme		
33	Source	compiled as		or	compil	led as
34 35 36	REAL :: buf, b1 call MPI_IRECV(buf,req)	REAL :: buf call MPI_IR register =	ECV(buf,.	.req) ca		buf, b1 _IRECV(buf,req)
37 38 39	<pre>call MPI_WAIT(req,) b1 = buf</pre>	call MPI_WA b1 = regist	IT(req,			_WAIT(req,)
40 41 42 43 44 45	Example 17.1 shows ext thread modifies buf between t But the compiler cannot see returned, and may schedule t has no reason to avoid using a nearder the instructions as ill	he invocation any possibilit the load of b u a register to h	of MPI_II by that built of earlier the rold buf a	RECV and f can be than type cross the	the con changed d in the call to f	npletion of MPI_WAIT. d after MPI_IRECV has e source. The compiler

 $_{46}$ reorder the instructions as illustrated in the rightmost column.

Due to valid compiler code movement optimizations in Example 17.2, the content of buf may already have been overwritten by the compiler when the content of buf is sent.

Unofficial Draft for Comment Only

Example 17.2 Similar exa	mple with MPI_ISEND	
Source	compiled as	with a possible MPI-internal execution sequence
<pre>REAL :: buf, copy buf = val call MPI_ISEND(buf,req) copy = buf</pre>	<pre>REAL :: buf, copy buf = val call MPI_ISEND(buf,req) copy= buf buf = val_overwrite</pre>	<pre>REAL :: buf, copy buf = val addr = &buf copy = buf buf = val_overwrite</pre>
<pre>call MPI_WAIT(req,)</pre>	<pre>call MPI_WAIT(req,)</pre>	call send(*addr) ! within ! MPI_WAIT
<pre>buf = val_overwrite</pre>		

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 17.1.17.

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 17.3 shows what Fortran compilers are allowed to do.

Unofficial Draft for Comment Only

 24

```
1
     Example 17.3 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
         In Example 17.3, the compiler does not invalidate the register because it cannot see
17
     that MPI_RECV changes the value of buf. The access to buf is hidden by the use of
18
     MPI_GET_ADDRESS and MPI_BOTTOM.
19
20
21
     Example 17.4 Similar example with MPI_SEND
22
     This source ...
                                                   can be compiled as:
23
^{24}
     ! buf contains val_old
                                                   ! buf contains val_old
25
     buf = val_new
26
     call MPI_SEND(MPI_BOTTOM,1,type,...)
                                                   call MPI_SEND(...)
27
     ! with buf as a displacement in type
                                                   ! i.e. val_old is sent
28
                                                   !
29
                                                   ! buf=val_new is moved to here
30
                                                   ! and detected as dead code
31
                                                   ! and therefore removed
32
                                                   Т
33
     buf = val_overwrite
                                                   buf = val_overwrite
34
35
         In Example 17.4, several successive assignments to the same variable buf can be com-
36
     bined in a way such that only the last assignment is executed. "Successive" means that
37
     no interfering load access to this variable occurs between the assignments. The compiler
38
     cannot detect that the call to MPI_SEND statement is interfering because the load access
39
     to buf is hidden by the usage of MPI_BOTTOM.
40
41
```

42 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 with the send and receive buffers used in nonblocking operations, or in operations in which MPI_BOTTOM is used, or if datatype handles that combine several variables are used:

Unofficial Draft for Comment Only

• Use of the Fortran ASYNCHRONOUS attribute.	1
• Use of the helper routing MDLE SYNC PEC, or an acquivelent user written dummu	2
• Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy routine.	3 4
• Declare the buffer as a Fortran module variable or within a Fortran common block.	5
• Declare the buller as a formall module variable of within a formal common block.	6
• Use of the Fortran VOLATILE attribute.	7
	8
Example 17.5 Protecting nonblocking communication with the ASYNCHRONOUS attribute.	9 10 11
NGE mai f09	12
<pre>USE mpi_f08 REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells REAL :: bnew(0:101)</pre>	13 14 15 16 17 18 19 20 21 22 23 24 25 26
bnew(i) = function(b(i-1), b(i), b(i+1))	27 28
END DO	29
#endif	30
	31
#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION	32
! Case (b)	33
DO i=2,99 ! compute only elements for which halo data is not needed	34
bnew(i) = function(b(i-1), b(i), b(i+1))	35
END DO	36
CALL MPI_Waitall(4, req,)	37
i=1 ! compute leftmost element	38
bnew(i) = function(b(i-1), b(i), b(i+1))	39
i=100 ! compute rightmost element	40
bnew(i) = function(b(i-1), b(i), b(i+1))	41
#endif	42

Each of these methods solves the problems of code movement and register optimization, but may incur various degrees of performance impact, and may not be usable in every application context. These methods may not be guaranteed by the Fortran standard, but they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated compiler suite according to the requirements listed in Section 17.1.7. The performance

impact of using MPI_F_SYNC_REG is expected to be low, that of using module variables
 or the ASYNCHRONOUS attribute is expected to be low to medium, and that of using the
 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.

6 7

8

The Fortran ASYNCHRONOUS Attribute

Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping 9 unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed 10 while the buffer is affected by a pending asynchronous Fortran input/output operation (since 11 Fortran 2003) or by an asynchronous communication (TS 29113 extension). Without the 12extensions specified in TS 29113, a Fortran compiler may totally ignore this attribute if the 13 Fortran compiler implements asynchronous Fortran input/output operations with blocking 14I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through 15code movements across routine calls, and the buffer itself from temporary and permanent 16data movements. If the choice buffer dummy argument of a nonblocking MPI routine is 17declared with ASYNCHRONOUS (which is mandatory for the mpi_f08 module, with allowable 18 exceptions listed in Section 17.1.6), then the compiler has to guarantee call by reference 19and should report a compile-time error if call by reference is impossible, e.g., if vector 20subscripts are used. The MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if both 21the protection of the actual buffer argument through ASYNCHRONOUS according to the TS 2229113 extension and the declaration of the dummy argument with ASYNCHRONOUS in the 23Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to 24 .FALSE.. 25

The ASYNCHRONOUS attribute has some restrictions. Section 5.4.2 of the TS 29113 specifies:

28"Asynchronous communication for a Fortran variable occurs through the action 29of procedures defined by means other than Fortran. It is initiated by execution 30 of an asynchronous communication initiation procedure and completed by exe-31cution of an asynchronous communication completion procedure. Between the 32 execution of the initiation and completion procedures, any variable of which any 33 part is associated with any part of the asynchronous communication variable is 34 a pending communication affector. Whether a procedure is an asynchronous 35communication initiation or completion procedure is processor dependent. 36

Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed."

In Example 17.5 Case (a) on page 659, the read accesses to b within function(b(i-1), b(i), b(i+1)) cannot be moved by compiler optimizations to before the wait call because b was declared as ASYNCHRONOUS. Note that only the elements 0, 1, 100, and 101 of b are involved in asynchronous communication but by definition, the total variable b is the pending communication affector and is usable for input and output asynchronous communication

Unofficial Draft for Comment Only

between the MPI_I... routines and MPI_Waitall. Case (a) works fine because the read accesses to **b** occur after the communication has completed.

In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to a pending communication affector while input communication (i.e., the two MPI_Irecv calls) is pending. This is a contradiction to the rule that for input communication, a pending communication affector shall not be referenced. The problem can be solved by using separate variables for the halos and the inner array, or by splitting a common array into disjoint subarrays which are passed through different dummy arguments into a subroutine, as shown in Example 17.9.

If one does not overlap communication and computation on the same variable, then all optimization problems can be solved through the ASYNCHRONOUS attribute.

The problems with MPI_BOTTOM, as shown in Example 17.3 and Example 17.4, can also be solved by declaring the buffer buf with the ASYNCHRONOUS attribute.

In some MPI routines, a buffer dummy argument is defined as ASYNCHRONOUS to guarantee passing by reference, provided that the actual argument is also defined as ASYNCHRONOUS.

Calling MPI_F_SYNC_REG

The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. The MPI library provides the MPI_F_SYNC_REG routine for this purpose; see Section 17.1.8.

• The problems illustrated by the Examples 17.1 and 17.2 can be solved by calling MPI_F_SYNC_REG(buf) once immediately after MPI_WAIT.

Example 17.1	Example 17.2	26
*	*	27
can be solved with	can be solved with	28
call MPI_IRECV(buf,req)	buf = val	29
	<pre>call MPI_ISEND(buf,req)</pre>	
	copy = buf	30
	10	31
<pre>call MPI_WAIT(req,)</pre>	call MPI_WAIT(req,)	32
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)	33
b1 = buf	<pre>buf = val_overwrite</pre>	33
or our	Sur Vur_SVCIWIIDC	34

The call to MPI_F_SYNC_REG(buf) prevents moving the last line before the MPI_WAIT call. Further calls to MPI_F_SYNC_REG(buf) are not needed because it is still correct if the additional read access copy=buf is moved below MPI_WAIT and before buf=val_overwrite.

• The problems illustrated by the Examples 17.3 and 17.4 can be solved with two additional MPI_F_SYNC_REG(buf) statements; one directly before MPI_RECV/ MPI_SEND, and one directly after this communication operation.

Example 17.3	Example 17.4	44
can be solved with	can be solved with	45
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)	46
<pre>call MPI_RECV(MPI_BOTTOM,)</pre>	<pre>call MPI_SEND(MPI_BOTTOM,)</pre>	47
call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)	48

1

 $\mathbf{2}$

3

4

5

6 $\overline{7}$

8

9

10 11

1213

14

15

1617

18

19

20

21

22 23

24

35

36

37

38

39

40 41

42

	662		CHAPTER 17. LANGUAGE BINDINGS
1 2 3 4			(buf) is needed to finish all load and store refer- PI_SEND; the second call is needed to assure that moved before MPI_RECV/SEND.
5 6 7 8 9 10 11 12 13 14		Process 1, the access to bbbb must b MPI_F_SYNC_REG(bbbb) is needed that further accesses to bbbb are not Process 2, both calls to MPI_WIN_F MPI_BOTTOM as the buffer. That is a call to MPI_F_SYNC_REG(buff) is	wo asynchronous accesses must be protected: in e protected similar to Example 17.1, i.e., a call to after the second MPI_WIN_FENCE to guarantee moved ahead of the call to MPI_WIN_FENCE. In ENCE together act as a communication call with , before the first fence and after the second fence, needed to guarantee that accesses to buff are not o MPI_WIN_FENCE. Using MPI_GET instead of _SYNC_REG are necessary.
15		Source of Process 1	Source of Process 2
16		bbbb = 777	buff = 999
17			call MPI_F_SYNC_REG(buff)
18		call MPI_WIN_FENCE	call MPI_WIN_FENCE
19		call MPI_PUT(bbbb	
20 21		into buff of process 2)	
21		COLL MDT WIN FENCE	COLL MDT LIIN FENCE
23		call MPI_WIN_FENCE call MPI_F_SYNC_REG(bbbb)	call MPI_WIN_FENCE call MPI_F_SYNC_REG(buff)
24			ccc = buff
25			
26 27 28		The temporary memory modificatio with this method.	n problem, i.e., Example 17.6, can not be solved
29 30	A Use	r Defined Routine Instead of MPI_F_1	SYNC_REG
31 32		ad of MPI_F_SYNC_REG, one can all arately compiled:	so use a user defined external subroutine, which
33 34		subroutine DD(buf)	
35		integer buf	
36		end	
37 38 39 40 41	$ it mu \\ does 1 $	st be OUT or INOUT. The subroutine	an explicit interface for the external subroutine, itself may have an empty body, but the compiler at the buffer may be altered. For example, a call er might be replaced by
41			
43		call DD(buf)	、 、
44		call MPI_RECV(MPI_BOTTOM,	.)
45		call DD(buf)	
46	Such	a user-defined routine was introduced	l in MPI-2.0 and is still included here to document
47 48			ns although new applications should prefer possibilities. In an existing application, calls to

CHAPTER 17. LANGUAGE BINDINGS

such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because the user-written routine may not be implemented in accordance with the rules specified in Section 17.1.7.

Module Variables and COMMON Blocks

An alternative to the previously mentioned methods is to put the buffer or variable into a module or a common block and access it through a USE or COMMON statement in each scope where it is referenced, defined or appears as an actual argument in a call to an MPI routine. The compiler will then have to assume that the MPI procedure may alter the buffer or variable, provided that the compiler cannot infer that the MPI procedure does not reference the module or common block.

- This method solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI_BOTTOM and derived datatype handles.
- Unfortunately, this method does **not** solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.

The (Poorly Performing) Fortran VOLATILE Attribute

The VOLATILE attribute gives the buffer or variable the properties needed to avoid register optimization or code movement problems, but it may inhibit optimization of any code containing references or definitions of the buffer or variable. On many modern systems, the performance impact will be large because not only register, but also cache optimizations will not be applied. Therefore, use of the VOLATILE attribute to enforce correct execution of MPI programs is discouraged.

The Fortran TARGET Attribute

The TARGET attribute does not solve the code movement problem because it is not specified for the choice buffer dummy arguments of nonblocking routines. If the compiler detects that the application program specifies the TARGET attribute for an actual buffer argument used in the call to a nonblocking routine, the compiler may ignore this attribute if no pointer reference to this buffer exists.

Rationale. The Fortran standardization body decided to extend the ASYNCHRONOUS attribute within the TS 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the TARGET attribute. (*End of rationale.*)

17.1.18 Temporary Data Movement and Temporary Memory Modification

The compiler is allowed to temporarily modify data in memory. Normally, this problem may occur only when overlapping communication and computation, as in Example 17.5, Case (b) on page 659. Example 17.6 also shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 17.7, buf(100,100) from Example 17.6 is equivalenced with the 1-dimensional array buf_1dim(10000). The nonblock-ing receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused

Unofficial Draft for Comment Only

 $\overline{7}$

```
1
      Example 17.6 Overlapping Communication and Computation.
\mathbf{2}
3
     USE mpi_f08
4
     REAL :: buf(100,100)
\mathbf{5}
      CALL MPI_Irecv(buf(1,1:100),..., req,...)
6
     DO j=1,100
7
        DO i=2,100
8
          buf(i,j)=...
9
        END DO
10
      END DO
11
      CALL MPI_Wait(req,...)
12
13
14
      Example 17.7 The compiler may substitute the nested loops through loop fusion.
15
16
     REAL :: buf(100,100), buf_1dim(10000)
17
      EQUIVALENCE (buf(1,1), buf_1dim(1))
18
     CALL MPI_Irecv(buf(1,1:100),..., req,...)
19
      tmp(1:100) = buf(1,1:100)
20
      DO j=1,10000
21
        buf_1dim(h)=...
22
     END DO
     buf(1,1:100) = tmp(1:100)
23
^{24}
     CALL MPI_Wait(req,...)
25
26
27
      loop is temporarily using this part of the buffer. When the tmp data is written back to buf,
28
      the previous data of buf(1,1:100) is restored and the received data is lost. The principle
29
      behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved
30
      to tmp.
^{31}
          Example 17.8 shows a second possible optimization. The whole array is temporarily
32
     moved to local_buf.
33
          When storing local_buf back to the original location buf, then this implies overwriting
34
      the section of buf that serves as a receive buffer in the nonblocking MPI call, i.e., this
35
      storing back of local_buf is therefore likely to interfere with asynchronously received data
36
      in buf(1,1:100).
37
          Note that this problem may also occur:
38
         • With the local buffer at the origin process, between an RMA communication call and
39
           the ensuing synchronization call; see Chapter 11.
40
41
         • With the window buffer at the target process between two ensuing RMA synchroniza-
42
           tion calls.
43
44
         • With the local buffer in MPI parallel file I/O split collective operations between the
45
           ..._BEGIN and ..._END calls; see Section 13.4.5.
46
          As already mentioned in subsection The Fortran ASYNCHRONOUS attribute on
47
      page 660 of Section 17.1.17, the ASYNCHRONOUS attribute can prevent compiler optimization
48
```

Example 17.8 Another optimization is based on the usage of a separate memory storage area, e.g., in a GPU.

with temporary data movement, but only if the receive buffer and the local references are separated into different variables, as shown in Example 17.9 and in Example 17.10.

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 17.6 and 17.7.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring **buf** as **VOLATILE** because the **VOLATILE** implies that all accesses to any storage unit (word) of **buf** must be directly done in the main memory exactly in the sequence defined by the application program. The **VOLATILE** attribute prevents all register and cache optimizations. Therefore, **VOLATILE** may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the ASYNCHRONOUS attribute. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

Rationale. This is a strong restriction for application programs. To weaken this restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 17.9 (which is a solution for the problem shown in Example 17.5 and ⁴⁶ in Example 17.10 (which is a solution for the problem shown in Example 17.8), the array is split into inner and halo part and both disjoint parts are passed to a subroutine ⁴⁸

Unofficial Draft for Comment Only

1 separated_sections. This routine overlaps the receiving of the halo data and the calcu- $\mathbf{2}$ lations on the inner part of the array. In a second step, the whole array is used to do the 3 calculation on the elements where inner+halo is needed. Note that the halo and the inner 4 area are strided arrays. Those can be used in non-blocking communication only with a TS $\mathbf{5}$ 29113 based MPI library.

6 $\overline{7}$

8

9

17.1.19 Permanent Data Movement

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. 10 An implementation with automatic garbage collection is one use case. Such permanent data 11 movement is in conflict with MPI in several areas: 12

13 14

15

16

17

18

- 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. 19This MPI standard requires that the problems with permanent data movement do not 20occur by imposing suitable restrictions on the MPI library together with the compiler used; 21see Section 17.1.7. 22

23 24

17.1.20 Comparison with C

25In C, subroutines which modify variables that are not in the argument list will not cause 26register optimization problems. This is because taking pointers to storage objects by using 27the & operator and later referencing the objects by indirection on the pointer is an integral 28part of the language. A C compiler understands the implications, so that the problem should 29not occur, in general. However, some compilers do offer optional aggressive optimization 30 levels which may not be safe. Problems due to temporary memory modifications can also 31 occur in C. As above, the best advice is to avoid the problem: use different variables for 32 buffers in nonblocking MPI operations and computation that is executed while a nonblocking 33 operation is pending. 34

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

- 44
- 45
- 4647
- 48

Example 17.9 Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

```
10
USE mpi_f08
                                                                                  11
REAL :: b(0:101) ! elements 0 and 101 are halo cells
                                                                                  12
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                  13
INTEGER :: i
                                                                                  14
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
                                                                                  15
i=1 ! compute leftmost element
                                                                                  16
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                  17
i=100 ! compute rightmost element
                                                                                  18
  bnew(i) = function(b(i-1), b(i), b(i+1))
                                                                                  19
END
                                                                                  20
                                                                                  21
SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
                                                                                  22
USE mpi_f08
                                                                                  23
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
                                                                                  24
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
                                                                                  25
TYPE(MPI_Request) :: req(4)
                                                                                  26
INTEGER :: left, right, i
                                                                                  27
CALL MPI_Cart_shift(...,left, right,...)
                                                                                  28
CALL MPI_Irecv(b_lefthalo ( 0), ..., left, ..., req(1), ...)
                                                                                  29
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
                                                                                  30
! b_lefthalo and b_righthalo is written asynchronously.
                                                                                  31
! There is no other concurrent access to b_lefthalo and b_righthalo.
                                                                                  32
CALL MPI_Isend(b_inner( 1), ..., left, ..., req(3), ...)
                                                                                  33
CALL MPI_Isend(b_inner(100), ..., right, ..., req(4), ...)
                                                                                  34
                                                                                  35
DO i=2,99 ! compute only elements for which halo data is not needed
                                                                                  36
  bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
                                                                                  37
  ! b_inner is read and 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
```

Unofficial Draft for Comment Only

1 2

4 5 6

8

9

```
1
\mathbf{2}
3
4
5
6
7
8
9
10
11
12
13
14
     Example 17.10 Protecting GPU optimizations with the ASYNCHRONOUS attribute.
15
     USE mpi_f08
16
     REAL :: buf(100,100)
17
     CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
18
     END
19
20
     SUBROUTINE separated_sections(buf_halo, buf_inner)
21
     REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
22
     REAL :: buf_inner(2:100,1:100)
23
     REAL :: local_buf(2:100,100)
^{24}
25
     CALL MPI_Irecv(buf_halo(1,1:100),..., req,...)
26
     local_buf = buf_inner
27
     DO j=1,100
28
       DO i=2,100
29
          local_buf(i,j)=...
30
       END DO
31
     END DO
32
     buf_inner = local_buf ! buf_halo is not touched!!!
33
34
     CALL MPI_Wait(req,...)
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

17.2 Language Interoperability

17.2.1 Introduction

It is not uncommon for library developers to use one language to develop an application library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication used between a parallel client and a parallel server. It should be possible to code the server in one language and the clients in another language. To do so, communications should be possible between applications written in different languages.

There are several issues that need to be addressed in order to achieve interoperability.

Initialization We need to specify how the MPI environment is initialized for all languages.

- Interlanguage passing of MPI opaque objects We need to specify how MPI object handles are passed between languages. We also need to specify what happens when an MPI object is accessed in one language, to retrieve information (e.g., attributes) set in another language.
- **Interlanguage communication** We need to specify how messages sent in one language can be received in another language.

It is highly desirable that the solution for interlanguage interoperability be extensible to new languages, should MPI bindings be defined for such languages.

17.2.2 Assumptions

We assume that conventions exist for programs written in one language to call routines written in another language. These conventions specify how to link routines in different languages into one program, how to call functions in a different language, how to pass arguments between languages, and the correspondence between basic data types in different languages. In general, these conventions will be implementation dependent. Furthermore, not every basic datatype may have a matching type in other languages. For example, C character strings may not be compatible with Fortran CHARACTER variables. However, we assume that a Fortran INTEGER, as well as a (sequence associated) Fortran array of INTEGERs, can be passed to a C program. We also assume that Fortran and C have addresssized integers. This does not mean that the default-size integers are the same size as default-sized pointers, but only that there is some way to hold (and pass) a C address in a Fortran integer. It is also assumed that INTEGER(KIND=MPI_OFFSET_KIND) can be passed from Fortran to C as MPI_Offset.

17.2.3 Initialization

A call to MPI_INIT or MPI_INIT_THREAD, from any language, initializes MPI for execution in all languages.

Advice to users. Certain implementations use the (inout) argc, argv arguments of the C version of MPI_INIT in order to propagate values for argc and argv to all

Unofficial Draft for Comment Only

1 2

3 4

5

6

 $\overline{7}$

8

9

10

11

12

13

14

15 16

17

18

19

20 21

22

23

 24

25 26

27 28

29

30

 31

32

33

34

35

36

37

38

39

40

41 42

43 44

45

 $46 \\ 47$

1 executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may 2 result in a loss of this ability. (End of advice to users.) 3 The function MPI_INITIALIZED returns the same answer in all languages. 4 The function MPI_FINALIZE finalizes the MPI environments for all languages. 5The function MPI_FINALIZED returns the same answer in all languages. 6 The function MPI_ABORT kills processes, irrespective of the language used by the 7 caller or by the processes killed. 8 9 The MPI environment is initialized in the same manner for all languages by 10 MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: 11same processes, same environmental attributes, same error handlers. Information can be added to info objects in one language and retrieved in another. 1213 Advice to users. The use of several languages in one MPI program may require the 14use of special options at compile and/or link time. (End of advice to users.) 1516Advice to implementors. Implementations may selectively link language specific MPI 17 libraries only to codes that need them, so as not to increase the size of binaries for codes 18 that use only one language. The MPI initialization code need perform initialization for 19 a language only if that language library is loaded. (End of advice to implementors.) 202117.2.4 Transfer of Handles 2223Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran 24 handles to C handles. There is no direct access to C handles in Fortran. 25The type definition MPI_Fint is provided in C for an integer of the size that matches a 26Fortran INTEGER; usually, MPI_Fint will be equivalent to int. With the Fortran mpi module 27or the mpif.h include file, a Fortran handle is a Fortran INTEGER value that can be used in 28the following conversion functions. With the Fortran mpi_f08 module, a Fortran handle is a 29 BIND(C) derived type that contains an INTEGER component named MPI_VAL. This INTEGER 30 value can be used in the following conversion functions. 31 The following functions are provided in C to convert from a Fortran communicator 32 handle (which is an integer) to a C communicator handle, and vice versa. See also Sec-33 tion 2.6.4. 34 C binding 35 MPI_Comm MPI_Comm_f2c(MPI_Fint comm) 36 If comm is a valid Fortran handle to a communicator, then MPI_Comm_f2c returns a 37 valid C handle to that same communicator; if $comm = MPI_COMM_NULL$ (Fortran value), 38 then MPI_Comm_f2c returns a null C handle; if comm is an invalid Fortran handle, then 39 MPI_Comm_f2c returns an invalid C handle. 40 MPI_Fint MPI_Comm_c2f(MPI_Comm comm) 41 42The function MPI_Comm_c2f translates a C communicator handle into a Fortran handle 43 to the same communicator; it maps a null handle into a null handle and an invalid handle 44 into an invalid handle. 45 Similar functions are provided for the other types of opaque objects. 46

- MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
- ⁴⁸ MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)

MPI_Group MPI_Group_f2c(MPI_Fint group)	1
MPI_Fint MPI_Group_c2f(MPI_Group group)	2 3
MPI_Request MPI_Request_f2c(MPI_Fint request)	4
MPI_Fint MPI_Request_c2f(MPI_Request request)	5 6
MPI_File MPI_File_f2c(MPI_Fint file)	7
MPI_Fint MPI_File_c2f(MPI_File file)	8
MPI_Win MPI_Win_f2c(MPI_Fint win)	9 10
	11
MPI_Fint MPI_Win_c2f(MPI_Win win)	12 13
MPI_Op MPI_Op_f2c(MPI_Fint op)	14
MPI_Fint MPI_Op_c2f(MPI_Op op)	15
MPI_Info MPI_Info_f2c(MPI_Fint info)	16 17
MPI_Fint MPI_Info_c2f(MPI_Info info)	18
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	19 20
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	20
MPI_Message MPI_Message_f2c(MPI_Fint message)	22
MPI_Fint MPI_Message_c2f(MPI_Message message)	23 24
In 1_1 int in 1_nobbage_ezi (in 1_nobbage mebbage)	25
Example 17.11 The example below illustrates how the Fortran MPI function	26 27
MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function	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
! FORTRAN PROCEDURE	32
SUBROUTINE MPI_TYPE_COMMIT(DATATYPE, IERR)	33
INTEGER :: DATATYPE, IERR	34 35
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)	
	36
RETURN END	36 37
RETURN END	37 38
RETURN	37
RETURN END	37 38 39
RETURN END /* C wrapper */ void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr) {	 37 38 39 40 41 42
RETURN END /* C wrapper */ void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr)	37 38 39 40 41
RETURN END /* C wrapper */ void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr) {	37 38 39 40 41 42 43
<pre>RETURN END /* C wrapper */ void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr) { MPI_Datatype datatype; datatype = MPI_Type_f2c(*f_handle); *ierr = (MPI_Fint)MPI_Type_commit(&datatype);</pre>	37 38 39 40 41 42 43 44 45 46
<pre>RETURN END /* C wrapper */ void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr) { MPI_Datatype datatype; datatype = MPI_Type_f2c(*f_handle);</pre>	37 38 39 40 41 42 43 44 45

1

 $\mathbf{2}$

3

4

56

7

8

9

10

11

12

13

14

15

1617

18

21

The same approach can be used for all other MPI functions. The call to MPI_XXX_f2c (resp. MPI_XXX_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

Rationale. The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.

Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple (e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (End of rationale.)

17.2.5 Status

19 The following two procedures are provided in C to convert from a Fortran (with the mpi 20module or mpif.h) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which 22is hidden. That is, no status information is lost in the conversion. 23

 24 25

26

27

28

29

30

 31

32

33

37

38

40

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 cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code. 39 To do the conversion in the other direction, we have the following:

int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status) 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.

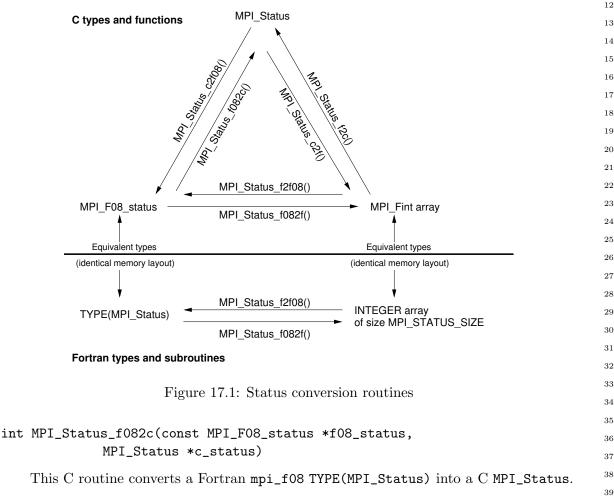
46 Advice to users. There exists no separate conversion function for arrays of statuses, 47 since one can simply loop through the array, converting each status with the routines 48 in Figure 17.1. (End of advice to users.)

Unofficial Draft for Comment Only

}

Rationale. The handling of MPI_STATUS_IGNORE is required in order to layer libraries with only a C wrapper: if the Fortran call has passed MPI_STATUS_IGNORE, then the C wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and C. If MPI_Status_f2c were to handle MPI_STATUS_IGNORE, then the type of its result would have to be MPI_Status**, which was considered an inferior solution. (*End of rationale.*)

Using the mpi_f08 Fortran module, a status is declared as TYPE(MPI_Status). The C type MPI_F08_status can be used to pass a Fortran TYPE(MPI_Status) argument into a C routine. Figure 17.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.



This C routine converts a C MPI_Status into a Fortran mpi_f08 TYPE(MPI_Status). ⁴² Two global variables of type MPI_F08_status*, MPI_F08_STATUS_IGNORE and ⁴³ MPI_F08_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether ⁴⁴ f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in ⁴⁵ the mpi_f08 module. These are global variables, not C constant expressions and cannot be ⁴⁶ used in places where C requires constant expressions. Their value is defined only between ⁴⁷ the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code. ⁴⁸

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

3

4

5 6

7

8

9

10

11

40

```
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
     C binding
9
     int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)
10
11
     F08 binding
12
     MPI_Status_f2f08(f_status, f08_status, ierror)
13
         INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
14
         TYPE(MPI_Status), INTENT(OUT) :: f08_status
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     F binding
17
     MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
18
         INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
19
         TYPE(MPI_Status) :: F08_STATUS
20
21
         This routine converts a Fortran INTEGER, DIMENSION (MPI_STATUS_SIZE) status array
22
     into a Fortran mpi_f08 TYPE(MPI_Status).
23
24
     MPI_STATUS_F082F(f08_status, f_status)
25
26
       IN
                f08_status
                                            status object declared as named type
27
       OUT
                f_status
                                            status object declared as array
28
29
     C binding
30
     int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)
31
32
     F08 binding
33
     MPI_Status_f082f(f08_status, f_status, ierror)
34
         TYPE(MPI_Status), INTENT(IN) :: f08_status
35
         INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     F binding
38
     MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
39
         TYPE(MPI_Status) :: F08_STATUS
40
         INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
41
42
         This routine converts a Fortran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER,
43
     DIMENSION(MPI_STATUS_SIZE) status array.
44
45
            MPI Opaque Objects
     17.2.6
46
47
     Unless said otherwise, opaque objects are "the same" in all languages: they carry the same
```

Unofficial Draft for Comment Only

information, and have the same meaning in both languages. The mechanism described

in the previous section can be used to pass references to MPI objects from language to language. An object created in one language can be accessed, modified or freed in another language.

We examine below in more detail issues that arise for each type of MPI object.

Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see Section 17.2.9).

Example 17.12

```
! FORTRAN CODE
REAL :: R(5)
INTEGER :: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
CALL C_ROUTINE(TYPE)
/* C code */
void C_ROUTINE(MPI_Fint *ftype)
ſ
   int count = 5;
   int lens[2] = \{1, 1\};
  MPI_Aint displs[2];
  MPI_Datatype types[2], newtype;
   /* create an absolute datatype for buffer that consists
                                                               */
   /* of count, followed by R(5)
                                                               */
  MPI_Get_address(&count, &displs[0]);
  displs[1] = 0;
  types[0] = MPI_INT;
  types[1] = MPI_Type_f2c(*ftype);
  MPI_Type_create_struct(2, lens, displs, types, &newtype);
```

1

2

3

4

5 6

7

8

9

10

11

12

13

14

15

16

17 18

19 20

21

22

23

24 25

26

27

28

29

30

31

32 33 34

35

36

37

38

39

40 41

42

43 44

45

46

47

8

9

10

11

12

13

14

15

16

17

18

19

20

21

25

27

29

36

37

38

39

40

41

1 2		<pre>MPI_Type_commit(&newtype);</pre>
3		MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
4		/* the message sent contains an int count of 5, followed */
5		<pre>/* by the 5 REAL entries of the Fortran array R. */</pre>
6	}	
7		

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.

22It may be desirable to use a value other than zero for MPI_BOTTOM even in C, so as 23to distinguish it from a NULL pointer. If $MPI_BOTTOM = c$ then one can still avoid 24 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 26in absolute datatypes. (End of advice to implementors.)

28Callback Functions

MPI calls may associate callback functions with MPI objects: error handlers are associ-30 ated with communicators and files, attribute copy and delete functions are associated with 31 attribute keys, reduce operations are associated with operation objects, etc. In a multilan-32 guage environment, a function passed in an MPI call in one language may be invoked by an 33 MPI call in another language. MPI implementations must make sure that such invocation 34 will use the calling convention of the language the function is bound to. 35

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.)

42Advice to users. If a subroutine written in one language or Fortran support method 43 wants to pass a callback routine including the predefined Fortran functions (e.g., 44MPI_COMM_NULL_COPY_FN) to another application routine written in another lan-45guage or Fortran support method, then it must be guaranteed that both routines use 46the callback interface definition that is defined for the argument when passing the 47 callback to an MPI routine (e.g., MPI_COMM_CREATE_KEYVAL); see also the advice 48 to users on page 302. (End of advice to users.)

Unofficial Draft for Comment Only

Error Handlers

Advice to implementors. Error handlers, have, in C, a variable length argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (*End of advice to implementors.*)

Reduce Operations

All predefined named and unnamed datatypes as listed in Section 5.9.2 can be used in the listed predefined operations independent of the programming language from which the MPI routine is called.

Advice to users. Reduce operations receive as one of their arguments the datatype of the operands. Thus, one can define "polymorphic" reduce operations that work for C and Fortran datatypes. (*End of advice to users.*)

17.2.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI_TAG_UB, MPI_WTIME_IS_GLOBAL, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

Advice to implementors. This requires that attributes be tagged either as "C" or "Fortran" and that the language tag be checked in order to use the right calling convention for the callback function. (*End of advice to implementors.*)

The attribute manipulation functions described in Section 6.7 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 integervalued attributes. C attribute functions put and get address-valued attributes. Fortran attribute functions put and get integer-valued attributes. When an integer-valued attribute is accessed from C, then MPI_XXX_get_attr will return the address of (a pointer to) the

Unofficial Draft for Comment Only

 24

1 integer-valued attribute, which is a pointer to MPI_Aint if the attribute was stored with $\mathbf{2}$ Fortran MPI_XXX_SET_ATTR, and a pointer to int if it was stored with the deprecated 3 Fortran MPI_ATTR_PUT. When an address-valued attribute is accessed from Fortran, then 4 MPI_XXX_GET_ATTR will convert the address into an integer and return the result of this $\mathbf{5}$ conversion. This conversion is lossless if new style attribute functions are used, and an 6 integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if 7deprecated attribute functions are used. In C, the deprecated routines MPI_Attr_put and 8 MPI_Attr_get behave identical to MPI_Comm_set_attr and MPI_Comm_get_attr. 9 Example 17.13 10 A. Setting an attribute value in C 11 12int set_val = 3; 13 struct foo set_struct; 1415/* Set a value that is a pointer to an int */ 1617MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val); 18 /* Set a value that is a pointer to a struct */ 19 MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct); 20/* Set an integer value */ 21MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17); 22 23B. Reading the attribute value in C 24int flag, *get_val; 25struct foo *get_struct; 2627/* Upon successful return, get_val == &set_val 28(and therefore *get_val == 3) */ 29 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag); 30 /* Upon successful return, get_struct == &set_struct */ 31MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag); 32 /* Upon successful return, get_val == (void*) 17 */ 33 i.e., (MPI_Aint) get_val == 17 */ /* 34 MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag); 35 36 C. Reading the attribute value with (deprecated) Fortran MPI-1 calls 37 38 LOGICAL FLAG 39 INTEGER IERR, GET_VAL, GET_STRUCT 4041 ! Upon successful return, GET_VAL == &set_val, possibly truncated 42CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR) 43 ! Upon successful return, GET_STRUCT == &set_struct, possibly truncated 44CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR) 45! Upon successful return, GET_VAL == 17 46CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR) 4748 D. Reading the attribute value with Fortran MPI-2 calls

```
1
LOGICAL FLAG
                                                                                       \mathbf{2}
INTEGER IERR
                                                                                       3
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT
                                                                                       4
! Upon successful return, GET_VAL == &set_val
                                                                                       5
                                                                                       6
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
                                                                                       7
! Upon successful return, GET_STRUCT == &set_struct
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
                                                                                       8
! Upon successful return, GET_VAL == 17
                                                                                       9
                                                                                       10
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
                                                                                       11
                                                                                       12
Example 17.14 A. Setting an attribute value with the (deprecated) Fortran MPI-1 call
                                                                                      13
                                                                                      14
INTEGER IERR, VAL
                                                                                       15
VAL = 7
                                                                                       16
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
                                                                                       17
                                                                                       18
    B. Reading the attribute value in C
                                                                                       19
                                                                                      20
int flag;
                                                                                      21
int *value;
                                                                                      22
                                                                                      23
/* Upon successful return, value points to internal MPI storage and
                                                                                       ^{24}
   *value == (int) 7 */
                                                                                      25
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
                                                                                       26
                                                                                      27
    C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
                                                                                      28
                                                                                      29
LOGICAL FLAG
                                                                                       30
INTEGER IERR, VALUE
                                                                                       31
                                                                                       32
! Upon successful return, VALUE == 7
                                                                                      33
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                      34
    D. Reading the attribute value with Fortran MPI-2 calls
                                                                                      35
                                                                                      36
LOGICAL FLAG
                                                                                      37
INTEGER IERR
                                                                                       38
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE
                                                                                       39
                                                                                       40
! Upon successful return, VALUE == 7 (sign extended)
                                                                                      41
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
                                                                                      42
                                                                                      43
                                                                                      44
Example 17.15 A. Setting an attribute value via a Fortran MPI-2 call
                                                                                       45
                                                                                       46
                                                                                       47
```

```
1
     INTEGER IERR
\mathbf{2}
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
3
     INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
4
     VALUE1 = 42
\mathbf{5}
     VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40
6
\overline{7}
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
8
     CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
9
         B. Reading the attribute value in C
10
11
     int flag;
12
     MPI_Aint *value1, *value2;
13
14
     /* Upon successful return, value1 points to internal MPI storage and
15
        *value1 == 42 */
16
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
17
     /* Upon successful return, value2 points to internal MPI storage and
18
        *value2 == 2^40 */
19
     MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
20
21
         C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
22
23
     LOGICAL FLAG
24
     INTEGER IERR, VALUE1, VALUE2
25
26
     ! Upon successful return, VALUE1 == 42
27
     CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
28
     ! Upon successful return, VALUE2 == 2<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,
```

680

MPI_TAG_UB, &p, &flag) will return in p a pointer to an int containing the upper bound for tag value.

Address-valued predefined attributes, such as MPI_WIN_BASE behave as if they were put by a C call, i.e., in Fortran, MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag, ierror) will return in val the base address of the window, converted to an integer. In C, MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window base, cast to (void *).

Rationale. The design is consistent with the behavior specified for predefined attributes, and ensures that no information is lost when attributes are passed from language to language. Because the language interoperability for predefined attributes was defined based on MPI_ATTR_PUT, this definition is kept for compatibility reasons although the routine itself is now deprecated. (*End of rationale.*)

Advice to implementors. Implementations should tag attributes either as (1) address attributes, (2) as INTEGER(KIND=MPI_ADDRESS_KIND) attributes or (3) as INTEGER attributes, according to whether they were set in (1) C (with MPI_Attr_put or MPI_XXX_set_attr), (2) in Fortran with MPI_XXX_SET_ATTR or (3) with the deprecated Fortran routine MPI_ATTR_PUT. Thus, the right choice can be made when the attribute is retrieved. (End of advice to implementors.)

17.2.8 Extra-State

Extra-state should not be modified by the copy or delete callback functions. (This is obvious from the C binding, but not obvious from the Fortran binding). However, these functions may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be a pointer to a data structure that is modified by the copy or callback functions; in Fortran, extra-state can be an index into an entry in a COMMON array that is modified by the copy or callback functions. In a multithreaded environment, users should be aware that distinct threads may invoke the same callback function concurrently: if this function modifies state associated with extra-state, then mutual exclusion code must be used to protect updates and accesses to the shared state.

17.2.9 Constants

MPI constants have the same value in all languages, unless specified otherwise. This does not apply to constant handles (MPI_INT, MPI_COMM_WORLD, MPI_ERRORS_RETURN, MPI_SUM, etc.) These handles need to be converted, as explained in Section 17.2.4. Constants that specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in Fortran than C since in C the length includes the null terminating character. Thus, these constants represent the amount of space which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

Advice to users. This definition means that it is safe in C to allocate a buffer to receive a string using a declaration like

char name [MPI_MAX_OBJECT_NAME];

Unofficial Draft for Comment Only

 $\mathbf{2}$

 $\mathbf{5}$

 24

 31

(End of advice to users.)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

Rationale. The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM in Fortran must be the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better. See the advice to implementors in the *Datatypes* subsection in Section 17.2.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (*End of rationale.*)

14 15 16

17

1

2

3

4

5 6

7

8

9

10

11

12

13

17.2.10 Interlanguage Communication

The type matching rules for communication in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it complies with these rules.

Example 17.16 In the example below, a Fortran array is sent from Fortran and received in C.

```
27
     ! FORTRAN CODE
28
     SUBROUTINE MYEXAMPLE()
29
     USE mpi_f08
30
     REAL :: R(5)
^{31}
     INTEGER :: IERR, MYRANK, AOBLEN(1)
32
     TYPE(MPI_Datatype) :: TYPE, AOTYPE(1)
33
     INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)
34
35
     ! create an absolute datatype for array R
36
     AOBLEN(1) = 5
37
     CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
38
     AOTYPE(1) = MPI_REAL
39
     CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
40
     CALL MPI_TYPE_COMMIT(TYPE, IERR)
41
42
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
43
     IF (MYRANK.EQ.O) THEN
44
        CALL MPI_SEND(MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
45
     ELSE
46
        CALL C_ROUTINE(TYPE%MPI_VAL)
47
     END IF
48
     END SUBROUTINE
```

```
/* C code */
void C_ROUTINE(MPI_Fint *fhandle)
{
    MPI_Datatype type;
    MPI_Status status;
    type = MPI_Type_f2c(*fhandle);
    MPI_Recv(MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
}
```

MPI implementors may weaken these type matching rules, and allow messages to be sent with Fortran types and received with C types, and vice versa, when those types match. I.e., if the Fortran type INTEGER is identical to the C type int, then an MPI implementation may allow data to be sent with datatype MPI_INTEGER and be received with datatype MPI_INT. However, such code is not portable.

Annex A

Language Bindings Summary

In this section we summarize the specific bindings for C and Fortran. First we present the constants, type definitions, info values and keys. Then we present the routine prototypes separately for each binding. Listings are alphabetical within chapter.

A.1 Defined Values and Handles

A.1.1 Defined Constants

The C and Fortran names are listed below. Constants with the type const int may also be implemented as literal integer constants substituted by the preprocessor.

	24
Error classes	25
C type: const int (or unnamed enum)	26
Fortran type: INTEGER	27
MPI_SUCCESS	28
MPI_ERR_BUFFER	29
MPI_ERR_COUNT	30
MPI_ERR_TYPE	31
MPI_ERR_TAG	32
MPI_ERR_COMM	33
MPI_ERR_RANK	34
MPI_ERR_REQUEST	35
MPI_ERR_ROOT	36
MPI_ERR_GROUP	37
MPI_ERR_OP	38
MPI_ERR_TOPOLOGY	39
MPI_ERR_DIMS	40
MPI_ERR_ARG	41
MPI_ERR_UNKNOWN	42
MPI_ERR_TRUNCATE	43
MPI_ERR_OTHER	44
MPI_ERR_INTERN	45
MPI_ERR_PENDING	46
(Continued on next page)	47
	48

Error classes (continued)
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_ERR_IN_STATUS
MPI_ERR_ACCESS
MPI_ERR_AMODE
MPI_ERR_ASSERT
MPI_ERR_BAD_FILE
MPI_ERR_BASE
MPI_ERR_CONVERSION
MPI_ERR_DISP
MPI_ERR_DUP_DATAREP
MPI_ERR_FILE_EXISTS
MPI_ERR_FILE_IN_USE
MPI_ERR_FILE
MPI_ERR_INFO_KEY
MPI_ERR_INFO_NOKEY
MPI_ERR_INFO_NOKE1
MPI_ERR_INFO_VALUE MPI_ERR_INFO
MPI_ERR_IO
MPI_ERR_KEYVAL
MPI_ERR_LOCKTYPE
MPI_ERR_LOCKTTPE MPI_ERR_NAME
MPI_ERR_NO_MEM
MPI_ERR_NOT_SAME
MPI_ERR_NO_SPACE
MPI_ERR_NO_SPACE MPI_ERR_NO_SUCH_FILE
MPI_ERR_PORT MPI_ERR_QUOTA
MPI_ERR_READ_ONLY
MPI_ERR_RMA_ATTACH
MPI_ERR_RMA_CONFLICT MPI_ERR_RMA_RANGE
MPI_ERR_RMA_SHARED MPI_ERR_RMA_SYNC
MPI_ERR_RMA_SYNC MPI_ERR_RMA_FLAVOR
MPI_ERR_RMA_FLAVOR MPI_ERR_SERVICE
MPI_ERR_SIZE
MPI_ERR_SPAWN MPI_ERR_UNSUPPORTED_DATAREP
MPI_ERR_UNSUPPORTED_OPERATION
MPI_ERR_WIN
(Continued on next page)

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) ^{1}	24
MPI_BOTTOM	25
MPI_IN_PLACE	26
1 Note that in Fortran these constants are not usable for initialized	zation 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 type: LOGICAL
-	MPI_SUBARRAYS_SUPPORTED (Fortran only)
	MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran of
-	
St	tatus size and reserved index values (Fortrar
Fo	ortran type: INTEGER
Μ	PI_STATUS_SIZE
Μ	PI_SOURCE
Μ	PI_TAG
Μ	PI_ERROR
	Variable Address Size (Fortran only)
	Fortran type: INTEGER
	MPI_ADDRESS_KIND
	MPI_ADDRESS_KIND MPI_COUNT_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_VAL
	MPI_MAX_LIBRARY_VERSION_STRING
	MPI_MAX_OBJECT_NAME
	MPI_MAX_PORT_NAME
	MPI_MAX_PROCESSOR_NAME

Named Predefined Datatypes	C types	
C type: MPI_Datatype		
Fortran type: INTEGER		
or TYPE(MPI_Datatype)		
MPI_CHAR	char	
	(treated as printable character)	
MPI_SHORT	signed short int	
MPI_INT	signed int	
MPI_LONG	signed long	
MPI_LONG_LONG_INT	signed long long	
MPI_LONG_LONG (as a synonym)	signed long long	
MPI_SIGNED_CHAR	signed char	
	(treated as integral value)	
MPI_UNSIGNED_CHAR	unsigned char	
	(treated as integral value)	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long	
MPI_UNSIGNED_LONG_LONG	unsigned long long	
MPI_FLOAT	float	
MPI_DOUBLE	double	
MPI_LONG_DOUBLE	long double	
MPI_WCHAR	wchar_t	
	(defined in <stddef.h>)</stddef.h>	
	(treated as printable character)	
MPI_C_BOOL	_Bool	
MPI_INT8_T	int8_t	
MPI_INT16_T	int16_t	
MPI_INT32_T	int32_t	
MPI_INT64_T	int64_t	
MPI_UINT8_T	uint8_t	
MPI_UINT16_T	uint16_t	
MPI_UINT32_T	uint32_t	
MPI_UINT64_T	uint64_t	
MPI_AINT	MPI_Aint	
MPI_COUNT	MPI_Count	
MPI_OFFSET	MPI_Offset	
MPI_C_COMPLEX	float _Complex	
MPI_C_FLOAT_COMPLEX	float _Complex	
MPI_C_DOUBLE_COMPLEX	double _Complex	
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex	
MPI_BYTE	(any C type)	
MPI_PACKED	(any C type)	

1	Named Predefined Datatypes	For	rtran types
2	C type: MPI_Datatype		
3	Fortran type: INTEGER		
4	or TYPE(MPI_Datatype)		
5	MPI_INTEGER	INT	ΓEGER
6	MPI_REAL	REA	AL
7	MPI_DOUBLE_PRECISION	DOU	JBLE PRECISION
8	MPI_COMPLEX	COM	MPLEX
9	MPI_LOGICAL	LOG	GICAL
10	MPI_CHARACTER	CHA	ARACTER(1)
11	MPI_AINT	INT	TEGER (KIND=MPI_ADDRESS_KIND)
12	MPI_COUNT	INT	FEGER (KIND=MPI_COUNT_KIND)
13	MPI_OFFSET		<pre>FEGER (KIND=MPI_OFFSET_KIND)</pre>
14	MPI_BYTE	(an	y Fortran type)
15	MPI_PACKED		y Fortran type)
16		,	
17	Named Predefined Datatype	\mathbf{es}^1	C++ types
18	C type: MPI_Datatype		
19	Fortran type: INTEGER		
20	or TYPE(MPI_Datatype)		
21	MPI_CXX_BOOL		bool
22	MPI_CXX_FLOAT_COMPLEX		<pre>std::complex<float></float></pre>
23	MPI_CXX_DOUBLE_COMPLEX		std::complex <double></double>
24	MPI_CXX_LONG_DOUBLE_COMPI	LEX	std::complex <long double=""></long>
25	$^{-1}$ If an accompanying C++ comp	piler	is missing, then the
26	MPI datatypes in this table are	e not	defined.
27			
28	Optional datatypes (I	Forti	ran) Fortran types
29	$\mathrm{C} \ \mathrm{type}$: MPI_Datatype		
30	Fortran type: INTEGER		
31	or TYPE(MPI_Datatype)		
32	MPI_DOUBLE_COMPLEX		DOUBLE COMPLEX
33	MPI_INTEGER1		INTEGER*1
34	MPI_INTEGER2		INTEGER*2
35	MPI_INTEGER4		INTEGER*4
36	MPI_INTEGER8		INTEGER*8
37	MPI_INTEGER16		INTEGER*16
38	MPI_REAL2		REAL*2
39	MPI_REAL4		REAL*4
40	MPI_REAL8		REAL*8
41	MPI_REAL16		REAL*16
42	MPI_COMPLEX4		COMPLEX*4
43	MPI_COMPLEX8		COMPLEX*8
44	MPI_COMPLEX16		COMPLEX*16
45	MPI_COMPLEX32		COMPLEX*32
46			
47			
48			

	Datatypes for reduction functions (C)	1 2
	type: MPI_Datatype	3
	ortran type: INTEGER or TYPE(MPI_Datatype)	4
		* 5
		6
	1PI_LONG_INT	7
	1PI_2INT	8
	1PI_SHORT_INT 1PI_LONG_DOUBLE_INT	9
N		10
Dat	atypes for reduction functions (Fortran)	11
C ty	pe: MPI_Datatype	12
Forti	an type: INTEGER or TYPE(MPI_Datatype)	13
MPI	2REAL	14
MPI_	2DOUBLE_PRECISION	15
MPI_	2INTEGER	16
		17
	Reserved communicators	18
	C type: MPI_Comm	19
	Fortran type: INTEGER or TYPE(MPI_Comm)	20
	MPI_COMM_WORLD	21
	MPI_COMM_SELF	22
		23
-	Communicator split type constants	24
	C type: const int (or unnamed enum)	25
_	Fortran type: INTEGER	26
_	MPI_COMM_TYPE_SHARED	27
		28
	s of communicator and group comparisons	29
v -	const int (or unnamed enum)	30
	type: INTEGER	31
MPI_ID		32
	NGRUENT	33
MPI_SIN		34
MPI_UN	IEQUAL	35
	Environmental inquiry info key	37
	C type: MPI_Info	38
	Fortran type: INTEGER or TYPE(MPI_Info)	39
	MPI_INFO_ENV	40
		41
	Environmental inquiry keys	42
	C type: const int (or unnamed enum)	43
	Fortran type: INTEGER	44
	MPI_TAG_UB	45
	MPI_IO	46
	MPI_HOST	47
	MPI_WTIME_IS_GLOBAL	48

Collective Operations
C type: MPI_Op
Fortran type: INTEGER or TYPE(MPI_0p)
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_MAXLOC
MPI_MIALOC MPI_MINLOC
_
MPI_BAND
MPI_BOR
MPI_BXOR
MPI_LAND
MPI_LOR
MPI_LXOR
MPI_REPLACE
MPI_NO_OP
NT 11 TT 11
Null Handles
C/Fortran name
C type / Fortran type
MPI_GROUP_NULL
MPI_Group / INTEGER or TYPE(MPI_Group)
MPI_COMM_NULL
MPI_Comm / INTEGER or TYPE(MPI_Comm)
MPI_DATATYPE_NULL
$ t MPI_Datatype \ / \ ext{INTEGER} \ ext{or} \ ext{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 TYPE(MPI_WIN) MPI_MESSAGE_NULL
<pre>MPI_Message / INTEGER or TYPE(MPI_Message)</pre>
Empty group
C type: MPI_Group
·
Fortran type: INTEGER or TYPE(MPI_Group) MPI_GROUP_EMPTY

	Topologies
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_GRAPH
	MPI_CART
	MPI_DIST_GRAPH
	Predefined functions
/Fortran name	
C type	
/ Fortran type with mpi	i module / Fortran type with mpi_f08 module
PI_COMM_NULL_COF	PY_FN
MPI_Comm_copy_attr_f	function
/ COMM_COPY_ATTR_FUN IPI_COMM_DUP_FN	$[CTION / PROCEDURE(MPI_Comm_copy_attr_function)]$
MPI_Comm_copy_attr_f	function
/ COMM_COPY_ATTR_FUN	
PI_COMM_NULL_DEL	_ETE_FN
MPI_Comm_delete_attr	r_function
/ COMM_DELETE_ATTR_F	· · · · · · · · · · · · · · · · · · ·
PI_WIN_NULL_COPY_	
MPI_Win_copy_attr_fu	
/ WIN_COPY_ATTR_FUNC	$TION / PROCEDURE(MPI_Win_copy_attr_function)$
PI_WIN_DUP_FN	
MPI_Win_copy_attr_fu	
/ WIN_COPY_ATTR_FUNC	
IPI_WIN_NULL_DELET	
MPI_Win_delete_attr_	
/ WIN_DELETE_ATTR_FU	· · · · · · · · · · · · · · · · · · ·
IPI_TYPE_NULL_COP	—
MPI_Type_copy_attr_f	
/ TYPE_COPY_ATTR_FUN IPI_TYPE_DUP_FN	<pre>ICTION / PROCEDURE(MPI_Type_copy_attr_function) 1)</pre>
MPI_TYPE_DOP_FN MPI_Type_copy_attr_f	function
/ TYPE_COPY_ATTR_FUN	
PI_TYPE_NULL_DELE	, , ,
MPI_Type_delete_attr	
/ TYPE_DELETE_ATTR_F	
PI_CONVERSION_FN_	
MPI_Datarep_conversi	
/ DATAREP_CONVERSION	
1 –	ementors (on page 301) and advice to users (on page 302)
-	ctran functions MPI_COMM_NULL_COPY_FN, in
Section $6.7.2$.	
Section $6.7.2$.	

2 C/Fortran name 3 C type / Fortran type with mpi module 4 MPI_NULL_COPY_FN MPI_Copy_function / COPY_FUNCTION 6 MPI_OUL_DELETE_FN 7 MPI_Delete_function / DELETE_FUNCTION 8 MPI_NULL_DELETE_FN 9 MPI_Delete_function / DELETE_FUNCTION 11 Predefined Attribute Keys 12 C type: const int (or unnamed enum) 13 Fortran type: INTEGER 14 MPI_APPNUM 15 MPI_WIN_DISP_UNIT 16 MPI_WIN_DISP_UNIT 17 MPI_WIN_DISP_UNIT 18 MPI_WIN_CREATE_FLAVOR 19 MPI_WIN_CREATE_FLAVOR 19 MPI_WIN_CREATE_FLAVOR 19 MPI_WIN_FLAVOR_CREATE 20 MPI_WIN_FLAVOR_ALLOCATE 21 MPI_WIN_FLAVOR_OPYNAMIC 22 MPI_WIN_FLAVOR_CREATE 23 MPI_WIN_SEPARATE 24 C type: const int (or unnamed enum) 25 Fortran type: INTEGER 26 MPI_WIN_FLAVOR_SHARED 27 MPI_WIN_SEPARATE <t< th=""><th>1</th><th>Deprecated predefined functions</th></t<>	1	Deprecated predefined functions
C type / Fortran type with mpi module MPL_NULL_COPY_FN MPI_Copy_function / COPY_FUNCTION MPI_Copy_function / COPY_FUNCTION MPI_DUP_FN MPI_Delete_function / DELETE_FUNCTION MPI_VIN_DELETE_FUNCTION MPI_VIN_FUNCERE_SIZE MPI_WIN_BASE MPI_WIN_SIZE MPI_WIN_SIZE MPI_WIN_SIZE MPI_WIN_SIZE MPI_WIN_FLAVOR_CREATE MPI_WIN_FLAVOR_ALLOCATE MPI_WIN_FLAVOR_ALLOCATE MPI_WIN_FLAVOR_CREATE MPI_WIN_SEPARATE MPI_WIN_SEPARATE MPI_WIN_UNIFIED MPI_WIN_UNIFIED	2	
4 MPI_NULL_COPY_FN 6 MPI_Opy_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 Predefined Attribute Keys 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_DISP_UNIT 18 MPI_WIN_DISP_UNIT 19 MPI_WIN_DISP_UNIT 19 MPI_WIN_DISP_UNIT 19 MPI_WIN_DISP_UNIT 19 MPI_WIN_DISP_UNIT 19 MPI_WIN_DISP_UNIT 19 MPI_WIN_DOEL 20 MPI_WIN_MODEL 21 MPI_WIN_FLAVOR_CREATE 22 MPI_WIN_FLAVOR_CREATE 23 MPI_WIN_FLAVOR_CREATE 24 C type: const int (or unnamed enum) 25 Fortran type: INTEGER 26 MPI_WIN_SEPARA	3	
6 MPI_DUP_FN 7 MPI_Copy_function / COPY_FUNCTION 8 MPI_NULL_DELETE_FN 9 MPI_Delete_function / DELETE_FUNCTION 11 Predefined Attribute Keys 12 C type: const int (or unnamed enum) 13 Fortran type: INTEGER 14 MPI_ASTUSEDCODE 15 MPI_WIN_BASE 18 MPI_WIN_DISP_UNIT 19 MPI_WIN_SIZE 20 MPI_WIN_MODEL 21 C type: const int (or unnamed enum) 22 C type: const int (or unnamed enum) 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_DYNAMIC 28 MPI_WIN_FLAVOR_DYNAMIC 29 MPI_WIN_SEPARATE 30 Fortran type: INTEGER 31 G type: const int (or unnamed enum) 32 Fortran type: INTEGER 33 MPI_WIN_SEPARATE 34 MPI_WIN_UNIFIED		MPI_NULL_COPY_FN
MPI_Copy_function / COPY_FUNCTION MPI_NULL_DELETE_FN MPI_Delete_function / DELETE_FUNCTION MPI_Delete_function / DELETE_FUNCTION MPI_Collete_function / DELETE_FUNCTION MPI_Collete_function / DELETE_FUNCTION MPI_Delete_function / DELETE_FUNCTION MPI_Collete_function / DELETE_FUNCTION MPI_VinCER MPI_APPNUM MPI_WIN_LASTUSEDCODE MPI_WIN_BASE MPI_WIN_BASE MPI_WIN_DISP_UNIT MPI_WIN_DISP_UNIT MPI_WIN_SIZE MPI_WIN_MODEL MPI_WIN_MODEL MPI_WIN_FLAVOR MPI_WIN_FLAVOR_CREATE MPI_WIN_FLAVOR_CREATE MPI_WIN_FLAVOR_SHARED MPI_WIN_SEPARATE MPI_WIN_UNIFIED MPI_WIN_UNIFIED		
MPI_NULL_DELETE_FN MPI_Delete_function / DELETE_FUNCTION Predefined Attribute Keys C type: const int (or unnamed enum) Fortran type: INTEGER MPI_APPNUM MPI_LASTUSEDCODE MPI_UNI_CRESE_SIZE MPI_WIN_BASE MPI_WIN_DISP_UNIT MPI_WIN_SIZE MPI_WIN_MODEL MPI_WIN_MODEL MPI_WIN_MODEL MPI_WIN_MODEL MPI_WIN_FLAVOR_CREATE Flavors C type: const int (or unnamed enum) Fortran type: INTEGER MPI_WIN_FLAVOR_ALLOCATE MPI_WIN_FLAVOR_SHARED MPI_WIN_SEPARATE MPI_WIN_UNIFIED MPI_WIN_UNIFIED MPI_WIN_UNIFIED		
MPI_Delete_function / DELETE_FUNCTION Predefined Attribute Keys C type: const int (or unnamed enum) Fortran type: INTEGER MPI_APPNUM MPI_APPNUM MPI_ASTUSEDCODE MPI_WIN_BASE MPI_WIN_DISP_UNIT MPI_WIN_SIZE MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL MPI_WIN_FLAVOR_CREATE MPI_WIN_FLAVOR_CREATE MPI_WIN_FLAVOR_CREATE MPI_WIN_FLAVOR_ALLOCATE MPI_WIN_FLAVOR_SHARED MPI_WIN_FLAVOR_SHARED MPI_WIN_SEPARATE MPI_WIN_UNIFIED		,
Information Predefined Attribute Keys C type: const int (or unnamed enum) Fortran type: INTEGER MPI_APPNUM MPI_LASTUSEDCODE MPI_UNIVERSE_SIZE MPI_WIN_BASE MPI_WIN_DISP_UNIT MPI_WIN_SIZE MPI_WIN_MODEL MPI_WIN_MODEL MPI_WIN_CREATE_FLAVOR MPI_WIN_MODEL MPI_WIN_MODEL C type: const int (or unnamed enum) Fortran type: INTEGER MPI_WIN_FLAVOR_CREATE MPI_WIN_FLAVOR_CREATE MPI_WIN_FLAVOR_ALLOCATE MPI_WIN_FLAVOR_SHARED MPI_WIN_FLAVOR_SHARED MPI_WIN_SEPARATE MPI_WIN_SEPARATE MPI_WIN_UNIFIED MPI_WIN_SEPARATE MPI_WIN_UNIFIED MPI_WIN_SEPARATE	9	
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_DASE 18 MPI_WIN_DISP_UNIT 19 MPI_WIN_SIZE 20 MPI_WIN_MODEL 21 MPI_WIN_MODEL 22 C type: const int (or unnamed enum) 23 Fortran type: INTEGER 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_OPYNAMIC 29 MPI_WIN_FLAVOR_SHARED 30 MPI_WIN_SEPARATE 31 MPI_WIN_SEPARATE 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED	10	
13 Fortran type: INTEGER 14 MPI_APPNUM 15 MPI_LASTUSEDCODE 16 MPI_UNIVERSE_SIZE 17 MPI_WIN_BASE 18 MPI_WIN_SIZE 20 MPI_WIN_SIZE 21 MPI_WIN_MODEL 22 MPI_WIN_MODEL 23 MPI_WIN_FLAVOR 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_SHARED 30 MPI_WIN_FLAVOR_SHARED 31 MPI_WIN_SEPARATE 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED	11	Predefined Attribute Keys
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_MODEL 21 MPI_WIN_MODEL 22 E type: const int (or unnamed enum) 25 Fortran type: INTEGER 26 MPI_WIN_FLAVOR_CREATE 27 MPI_WIN_FLAVOR_ALLOCATE 28 MPI_WIN_FLAVOR_OPYNAMIC 29 MPI_WIN_FLAVOR_SHARED 30 String through the second term 31 MPI_WIN_SEPARATE 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 99 40 44 41 44 42 44 43 44 44 45 44 45 45 47	12	C type: const int (or unnamed enum)
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 MPI_WIN_MODEL 23 MPI_WIN_MODEL 24 C type: const int (or unnamed enum) 25 Fortran type: INTEGER 26 MPI_WIN_FLAVOR_CREATE 27 MPI_WIN_FLAVOR_DYNAMIC 28 MPI_WIN_FLAVOR_SHARED 30 MPI_WIN_STEGER 31 MPI_WIN_STEAVOR_SHARED 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED	13	Fortran type: INTEGER
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 MPI_WIN_MODEL 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 MPI_WIN_SEPARATE 31 MPI_WIN_SEPARATE 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_UNIFIED 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED 37 Sa 38 Sa 39 Sa 34 Sa 35 MPI_WIN_UNIFIED 36 Sa 37 Sa 38 Sa 39 Sa <tr< td=""><td>14</td><td>MPI_APPNUM</td></tr<>	14	MPI_APPNUM
17 MPI_WIN_BASE 18 MPI_WIN_DISP_UNIT 19 MPI_WIN_SIZE 20 MPI_WIN_CREATE_FLAVOR 21 MPI_WIN_MODEL 22 C type: const int (or unnamed enum) 25 Fortran type: INTEGER 26 MPI_WIN_FLAVOR_CREATE 27 MPI_WIN_FLAVOR_CREATE 28 MPI_WIN_FLAVOR_DYNAMIC 29 MPI_WIN_FLAVOR_SHARED 30 MPI_WIN_FLAVOR_SHARED 31 MPI Window Models 32 C type: const int (or unnamed enum) 34 MPI_WIN_SEPARATE 35 MPI_WIN_SEPARATE 36 MPI_WIN_UNIFIED 37 MPI_WIN_UNIFIED 38 MPI_WIN_UNIFIED	15	MPI_LASTUSEDCODE
18 MPI_WIN_DISP_UNIT 19 MPI_WIN_SIZE 20 MPI_WIN_CREATE_FLAVOR 21 MPI_WIN_MODEL 22 C type: const int (or unnamed enum) 25 Fortran type: INTEGER 26 MPI_WIN_FLAVOR_CREATE 27 MPI_WIN_FLAVOR_CREATE 28 MPI_WIN_FLAVOR_ALLOCATE 29 MPI_WIN_FLAVOR_SHARED 30 MPI_WIN_FLAVOR_SHARED 31 MPI Window Models 32 C type: const int (or unnamed enum) 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED 37 API_WIN_UNIFIED 38 API_WIN_UNIFIED	16	
18 MPI_WIN_DISP_UNIT 19 MPI_WIN_SIZE 20 MPI_WIN_CREATE_FLAVOR 21 MPI_WIN_MODEL 22 C type: const int (or unnamed enum) 25 Fortran type: INTEGER 26 MPI_WIN_FLAVOR_CREATE 27 MPI_WIN_FLAVOR_CREATE 28 MPI_WIN_FLAVOR_ALLOCATE 29 MPI_WIN_FLAVOR_SHARED 30 MPI_WIN_FLAVOR_SHARED 31 MPI Window Models 32 C type: const int (or unnamed enum) 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED 37 API_WIN_UNIFIED 38 API_WIN_UNIFIED	17	
19 MPI_WIN_SIZE 20 MPI_WIN_CREATE_FLAVOR 21 MPI_WIN_MODEL 22 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 C type: const int (or unnamed enum) 31 Fortran type: INTEGER 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED 37 MPI_WIN_UNIFIED	18	
20 MPI_WIN_CREATE_FLAVOR 21 MPI_WIN_MODEL 22 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 MPI Window Models 31 C type: const int (or unnamed enum) 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED 37 MPI_WIN_UNIFIED	19	
MPI_WIN_MODEL MPI_Window Create Flavors C type: const int (or unnamed enum) Fortran type: INTEGER MPI_WIN_FLAVOR_CREATE MPI_WIN_FLAVOR_ALLOCATE MPI_WIN_FLAVOR_DYNAMIC MPI_WIN_FLAVOR_SHARED MPI_WIN_FLAVOR_SHARED C type: const int (or unnamed enum) Fortran type: INTEGER MPI_WIN_SEPARATE MPI_WIN_UNIFIED MPI_WIN_UNIFIED	20	
22 23 24 25 26 7 28 9 10 11 12 13 14 15 16 17 18 19 10 11 12 13 14 15 16 17 18 19 10	21	
24C type: const int (or unnamed enum)25Fortran type: INTEGER26MPI_WIN_FLAVOR_CREATE27MPI_WIN_FLAVOR_ALLOCATE28MPI_WIN_FLAVOR_DYNAMIC29MPI_WIN_FLAVOR_SHARED30C type: const int (or unnamed enum)33Fortran type: INTEGER34MPI_WIN_SEPARATE35MPI_WIN_UNIFIED3637383940414243444545464747	22	
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 MPI_WIN_FLAVOR_SHARED 31 MPI Window Models 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED 37 S 38 MPI_WIN_HIPHICAL 39 MPI_WIN_UNIFIED 40 MI 41 MI 42 MI 43 MI 44 MI 45 MI 46 MI 47 MI	23	MPI Window Create Flavors
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 MPI_WIN_FLAVOR_SHARED 31 MPI Window Models 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED 37 S 38 MPI_WIN_HIPHICAL 39 MPI_WIN_UNIFIED 40 MI 41 MI 42 MI 43 MI 44 MI 45 MI 46 MI 47 MI	24	C type: const int (or unnamed enum)
26MPI_WIN_FLAVOR_CREATE27MPI_WIN_FLAVOR_ALLOCATE28MPI_WIN_FLAVOR_DYNAMIC29MPI_WIN_FLAVOR_SHARED30MPI Window Models32C type: const int (or unnamed enum)33Fortran type: INTEGER34MPI_WIN_SEPARATE35MPI_WIN_UNIFIED3637383940414243444545464747	25	
27MPI_WIN_FLAVOR_ALLOCATE28MPI_WIN_FLAVOR_DYNAMIC29MPI_WIN_FLAVOR_SHARED30Impl_WIN_FLAVOR_SHARED31MPI Window Models32C type: const int (or unnamed enum)33Fortran type: INTEGER34MPI_WIN_SEPARATE35MPI_WIN_UNIFIED3637383940414243444445464747	26	
28MPI_WIN_FLAVOR_DYNAMIC MPI_WIN_FLAVOR_SHARED30MPI_WIN_FLAVOR_SHARED31MPI Window Models32C type: const int (or unnamed enum) Fortran type: INTEGER34MPI_WIN_SEPARATE MPI_WIN_UNIFIED36	27	
29MPI_WIN_FLAVOR_SHARED30MPI_Window Models31MPI Window Models32C type: const int (or unnamed enum)33Fortran type: INTEGER34MPI_WIN_SEPARATE35MPI_WIN_UNIFIED369404142434344444545464741	28	
30 MPI Window Models 32 C type: const int (or unnamed enum) 33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 MPI_WIN_UNIFIED 37 MPI_WIN_UNIFIED 38 MPI_WIN_UNIFIED 39 MPI_WIN_UNIFIED 40 MPI_WIN_UNIFIED 41 MPI_WIN_UNIFIED 42 MPI_WIN_UNIFIED 43 MPI_WIN_UNIFIED 44 MPI_WIN_UNIFIED 45 MPI_WIN_UNIFIED 46 MPI_WIN_UNIFIED	29	
32C type: const int (or unnamed enum)33Fortran type: INTEGER34MPI_WIN_SEPARATE35MPI_WIN_UNIFIED3637383940414243434445464747	30	
33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 37 38 39 40 40 41 42 43 44 43 44 45 46 47	31	MPI Window Models
33 Fortran type: INTEGER 34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36 37 38 39 40 41 42 43 44 45 46 47	32	C type: const int (or unnamed enum)
34 MPI_WIN_SEPARATE 35 MPI_WIN_UNIFIED 36	33	· - / /
35 MPI_WIN_UNIFIED 36	34	
36 37 38 39 40 41 42 43 44 45 46 47	35	
38 39 40 41 42 43 44 45 46 47	36	
39 40 41 42 43 44 45 46 47	37	
40 41 42 43 44 45 46 47	38	
41 42 43 44 45 46 47	39	
42 43 44 45 46 47	40	
43 44 45 46 47	41	
44 45 46 47	42	
45 46 47	43	
46 47	44	
46 47	45	
47		
	48	

Mode Constants	1
C type: const int (or unnamed enum)	2
Fortran type: INTEGER	3
MPI_MODE_APPEND	4
MPI_MODE_CREATE	5
MPI_MODE_DELETE_ON_CLOSE	6
MPI_MODE_EXCL	7
MPI_MODE_NOCHECK	8
MPI_MODE_NOPRECEDE	9
MPI_MODE_NOPUT	10
MPI_MODE_NOSTORE	11
MPI_MODE_NOSUCCEED	12
MPI_MODE_RDONLY	13
MPI_MODE_RDWR	14
MPI_MODE_SEQUENTIAL	15
MPI_MODE_UNIQUE_OPEN	16
MPI_MODE_WRONLY	17
	18
Datatype Decoding Constants	19
C type: const int (or unnamed enum)	20
Fortran type: INTEGER	21
MPI_COMBINER_CONTIGUOUS	22
MPI_COMBINER_DARRAY	23
MPI_COMBINER_DUP	24
MPI_COMBINER_F90_COMPLEX	25
MPI_COMBINER_F90_INTEGER	26
MPI_COMBINER_F90_REAL	27
MPI_COMBINER_HINDEXED	28
MPI_COMBINER_HVECTOR	29
MPI_COMBINER_INDEXED_BLOCK	30
MPI_COMBINER_HINDEXED_BLOCK	31
MPI_COMBINER_INDEXED	32
MPI_COMBINER_NAMED	33
MPI_COMBINER_RESIZED	34
MPI_COMBINER_STRUCT	35
MPI_COMBINER_SUBARRAY	36
MPI_COMBINER_VECTOR	37
Threads Constants	38
	40
C type: const int (or unnamed enum) Fortran type: INTEGER	41
MPI_THREAD_FUNNELED	42
MPI_THREAD_FONNELED MPI_THREAD_MULTIPLE	43
MPI_THREAD_MOLTIPLE MPI_THREAD_SERIALIZED	44
MPI_THREAD_SERIALIZED MPI_THREAD_SINGLE	45
	46
	47

1	File Operation Constants, Part 1
2	C type: const MPI_Offset (or unnamed enum)
3	Fortran type: INTEGER (KIND=MPI_OFFSET_KIND)
4	MPI_DISPLACEMENT_CURRENT
5	
6	File Operation Constants, Part 2
7	C type: const int (or unnamed enum)
8	Fortran type: INTEGER
9	MPI_DISTRIBUTE_BLOCK
10	MPI_DISTRIBUTE_CYCLIC
11	MPI_DISTRIBUTE_DFLT_DARG
12	MPI_DISTRIBUTE_NONE
13	MPI_ORDER_C
14	MPI_ORDER_FORTRAN
15	MPI_SEEK_CUR
16	MPI_SEEK_END
17	MPI_SEEK_SET
18	
19	F90 Datatype Matching Constants
20	C type: const int (or unnamed enum)
21	Fortran type: INTEGER
22	MPI_TYPECLASS_COMPLEX
23	MPI_TYPECLASS_INTEGER
24	MPI_TYPECLASS_REAL
25	
26	Constants Specifying Empty or Ignored Input
27	C/Fortran name
28	C type / Fortran type ¹
29	MPI_ARGVS_NULL
30	char*** / 2-dim. array of CHARACTER*(*)
31	MPI_ARGV_NULL
32	<pre>char** / array of CHARACTER*(*)</pre>
33	MPI_ERRCODES_IGNORE
34	int* / INTEGER array
35	MPI_STATUSES_IGNORE
36	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)
37	or TYPE(MPI_Status), DIMENSION(*)
37 38	,
	or TYPE(MPI_Status), DIMENSION(*)
38	or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE
38 39	or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)
38 39 40	or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status)
38 39 40 41	or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED
38 39 40 41 42	or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array
38 39 40 41 42 43	or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY
38 39 40 41 42 43 44	or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY int* / INTEGER array
38 39 40 41 42 43 44 45	or TYPE(MPI_Status), DIMENSION(*) MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE) or TYPE(MPI_Status) MPI_UNWEIGHTED int* / INTEGER array MPI_WEIGHTS_EMPTY int* / INTEGER array ¹ Note that in Fortran these constants are not usable for initialization

C Constants Specify	ing Ignored Input (no Fortran)
ype: MPI_Fint*	equivalent to Fortran
PI_F_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi / mpif.
PI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h
type: MPI_F08_status*	equivalent to Fortran
PI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi_f08
PI_F08_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi_f08
C preprocessor Con	stants and Fortran Parameters
	nacro that expands to an int value
Fortran type: INTEGER	r in the second s
MPI_SUBVERSION	
MPI_VERSION	
Null handles used in t	he MPI tool information interface
MPI_T_ENUM_NULL	
MPI_T_enum	
MPI_T_CVAR_HANDLE_NU	LL
MPI_T_cvar_handle	
MPI_T_PVAR_HANDLE_NU	LL
MPI_T_pvar_handle	
MPI_T_PVAR_SESSION_NU	ILL
MPI_T_pvar_session	
	e MPI tool information interface
C type: const int (or unn	,
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_USER	
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_TUNE	
MPI_T_VERBOSITY_MPID MPI_T_VERBOSITY_MPID	
MPI_T_VERBOSITY_MPID MPI_T_VERBOSITY_MPID	

Constants to identify associations of variables
in the MPI tool information interface
C type: const int (or unnamed enum)
MPI_T_BIND_NO_OBJECT
MPI_T_BIND_MPI_COMM
MPI_T_BIND_MPI_DATATYPE
MPI_T_BIND_MPI_ERRHANDLER
MPI_T_BIND_MPI_FILE
MPI_T_BIND_MPI_GROUP
MPI_T_BIND_MPI_OP
MPI_T_BIND_MPI_REQUEST
MPI_T_BIND_MPI_WIN
MPI_T_BIND_MPI_MESSAGE
MPI_T_BIND_MPI_INFO
Constants describing the scope of a control variab
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
C type: MP1_1_pvar_nandle
MPI_T_PVAR_ALL_HANDLES
MPI_T_PVAR_ALL_HANDLES
MPI_T_PVAR_ALL_HANDLES Performance variables classes used by the
MPI_T_PVAR_ALL_HANDLES Performance variables classes used by the MPI tool information interface
MPI_T_PVAR_ALL_HANDLES Performance variables classes used by the MPI tool information interface C type: const int (or unnamed enum)
MPI_T_PVAR_ALL_HANDLES 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_ALL_HANDLES 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_ALL_HANDLES 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_ALL_HANDLES 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_ALL_HANDLES 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_ALL_HANDLES 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_ALL_HANDLES 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_ALL_HANDLES 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_TIMER
MPI_T_PVAR_ALL_HANDLES 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

A.1.2 Types

47

 $_{48}$ $\,$ The following are defined C type definitions, included in the file mpi.h.

/* C opaque types */	1
MPI_Aint	2
MPI_Count	3
MPI_Fint	4
MPI_Offset	5
MPI_Status	6
MPI_F08_status	7
	8
/* C handles to assorted structures */	9
MPI_Comm	10
MPI_Datatype	11
MPI_Errhandler	12
MPI_File	13
MPI_Group	14
MPI_Info	15
	16
MPI_Message	17
MPI_Op	18
MPI_Request	19
MPI_Win	20
	20
<pre>/* Types for the MPI_T interface */</pre>	21
MPI_T_enum	22
MPI_T_cvar_handle	
MPI_T_pvar_handle	24
MPI_T_pvar_session	25
	26
	27
The following are defined Fortran type definitions, included in the mpi_f08 and mpi	28
modules.	29
! Fortran opaque types in the mpi_f08 and mpi modules	30
TYPE(MPI_Status)	31
	32
! Fortran handles in the mpi_f08 and mpi modules	33
TYPE(MPI_Comm)	34
TYPE(MPI_Datatype)	35
TYPE(MPI_Errhandler)	36
TYPE(MPI_File)	37
TYPE(MPI_Group)	38
TYPE(MPI_Info)	39
TYPE(MPI_III0) TYPE(MPI_Message)	40
-	41
TYPE(MPI_Op)	42
TYPE(MPI_Request)	43
TYPE(MPI_Win)	44
	45
	46
	47
	48

1 A.1.3 Prototype Definitions $\mathbf{2}$ C Bindings 3 4 The following are defined C typedefs for user-defined functions, also included in the file 5mpi.h. 6 $\overline{7}$ /* prototypes for user-defined functions */ typedef void MPI_User_function(void *invec, void *inoutvec, int *len, 8 9 MPI_Datatype *datatype); 10 11 typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, 12void *attribute_val_out, int *flag); 13 14typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval, void *attribute_val, void *extra_state); 151617typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval, 18 void *extra_state, void *attribute_val_in, 19 void *attribute_val_out, int *flag); typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval, 2021void *attribute_val, void *extra_state); 2223typedef 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
```

1	
2	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
3	ATTRIBUTE_VAL_OUT LOGICAL FLAG
4	LOGICAL FLAG
5	SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
6	EXTRA_STATE, IERROR)
7	INTEGER WIN, WIN_KEYVAL, IERROR
8	INTEGER (KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
9	INIBOLI((IND IN I_NDDALDO_KIND) AIII(IDOIL_VAL, DAINA_DIAIL
10	The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be
11	declared like these:
12	
13	SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
14	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
15	INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
16	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
17	ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
18	LOGICAL FLAG
19	
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	The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de-
25	clared like this:
26	
27	SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
28 29	INTEGER COMM, ERROR_CODE
29 30	
31	The handler-function argument to $MPI_WIN_CREATE_ERRHANDLER$ should be de-
32	clared like this:
33	
34	SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
35	INTEGER WIN, ERROR_CODE
36	The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-
37	clared like this:
38	
39	SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
40	INTEGER FILE, ERROR_CODE
41	
42	The query, free, and cancel function arguments to $MPI_GREQUEST_START$ should be
43	declared like these:
44	
45	SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
46	INTEGER STATUS(MPI_STATUS_SIZE), IERROR
47	INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
48	

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR) INTEGER IERROR	1 2
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	$\frac{3}{4}$
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR) INTEGER IERROR	5 6
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE LOGICAL COMPLETE	7 8
The extent and conversion function arguments to MPI_REGISTER_DATAREP should be declared like these:	9 10 11
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR) INTEGER DATATYPE, IERROR	12 13 14
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE	15
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION, EXTRA_STATE, IERROR)	16 17 18
<type> USERBUF(*), FILEBUF(*) INTEGER COUNT, DATATYPE, IERROR</type>	19 20
INTEGER(KIND=MPI_OFFSET_KIND) POSITION INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	21 22
A.1.4 Deprecated Prototype Definitions	23 24
The following are defined C typedefs for deprecated user-defined functions, also included in the file mpi.h.	25 26 27
<pre>/* prototypes for user-defined functions */ typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,</pre>	28 29
<pre>void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag);</pre>	30 31 32
<pre>typedef int MPI_Delete_function(MPI_Comm comm, int keyval,</pre>	32 33 34
The following are deprecated Fortran user-defined callback subroutine prototypes. The deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be de-	35 36 37
clared like these:	38
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)	39 40
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR	41 42 43
LOGICAL FLAG	44
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR	45 46 47
	48

¹ A.1.5 Info Keys

- $_{3}^{2}$ The following info keys are reserved. They are strings.
- 4 access_style
- 5 accumulate_ops
- 6 accumulate_ordering
- 7 alloc_shared_noncontig
- 8 appnum
- 9 arch
- 10 cb_block_size
- 11 cb_buffer_size
- 12 cb_nodes
- 13 chunked_item
- 14 chunked_size
- 15 chunked
- ¹⁶ collective_buffering
- 17 file_perm
- 18 filename
- 19 file
- 20 host
- 21 io_node_list
- 22 ip_address
- 23 ip_port
- ²⁴ mpi_assert_allow_overtaking
- 25 mpi_assert_exact_length
- 26 mpi_assert_no_any_source
- 27 mpi_assert_no_any_tag
- 28 mpi_assert_strict_start_ordering
- ²⁹ mpi_optimization_goal
- 30 mpi_reuse_count
- 31 nb_proc
- 32 no_locks
- 33 num_io_nodes
- 34 path
- 35 same_disp_unit
- 36 same_size
- 37 soft
- 38 striping_factor
- 39 striping_unit
- 40 wdir
- 41
- 42

⁴³ A.1.6 Info Values

- $\frac{44}{45}$ The following info values are reserved. They are strings.
- 46 false
- 47 random
- 48 rar

raw	1
read_mostly	2
read_once	3
reverse_sequential	4
same_op	5
same_op_no_op	6
sequential	7
true	8
war	9
waw	10
write_mostly	11
write_once	12
	13
	14
	15
	16
	17
	18
	19
	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30
	31
	32
	33
	34
	35
	36
	37
	38

1	A.2 C Bindings
2 3	A.2.1 Point-to-Point Communication C Bindings
4 5 6 7	C binding int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
8 9	<pre>int MPI_Bsend_init(const void *buf, int count, MPI_Datatype datatype,</pre>
10 11	<pre>int MPI_Buffer_attach(void *buffer, int size)</pre>
12	<pre>int MPI_Buffer_detach(void *buffer_addr, int *size)</pre>
13 14	int MPI_Cancel(MPI_Request *request)
15 16	<pre>int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,</pre>
17 18 19	<pre>int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
20 21 22	int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message, MPI_Status *status)
22 23 24	int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Request *request)
25 26 27	int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
28 29	<pre>int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,</pre>
30 31 32	<pre>int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
33 34	<pre>int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
35 36 37	<pre>int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
38 39	int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)
40 41 42	int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype, MPI_Message *message, MPI_Status *status)
43	int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
44 45 46	<pre>int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,</pre>
47 48	<pre>int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source,</pre>

int	MPI_Request_free(MPI_Request *request)	1
int	<pre>MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status)</pre>	2 3 4
int	<pre>MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	5 6 7
int	<pre>MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype,</pre>	7 8 9
int	<pre>MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	10 11 12
int	<pre>MPI_Send_init(const void *buf, int count, MPI_Datatype datatype,</pre>	13 14
int	<pre>MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,</pre>	15 16 17 18 19
int	<pre>MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,</pre>	20 21 22
int	<pre>MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	23 24 25
int	<pre>MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype,</pre>	26 27
int	MPI_Start(MPI_Request *request)	28 29
int	<pre>MPI_Startall(int count, MPI_Request array_of_requests[])</pre>	30
int	MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)	31 32
int	MPI_Test_cancelled(const MPI_Status *status, int *flag)	33
int	<pre>MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status array_of_statuses[])</pre>	34 35 36
int	<pre>MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag, MPI_Status *status)</pre>	37 38
int	<pre>MPI_Testsome(int incount, MPI_Request array_of_requests[],</pre>	39 40 41 42
int	MPI_Wait(MPI_Request *request, MPI_Status *status)	43
int	<pre>MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])</pre>	44 45 46
int	<pre>MPI_Waitany(int count, MPI_Request array_of_requests[], int *index,</pre>	47 48

1	MPI_Status *status)
2 3 4 5 6	<pre>int MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>
7 8	A.2.2 Datatypes C Bindings
9	MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
10 11	MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)
12	<pre>int MPI_Get_address(const void *location, MPI_Aint *address)</pre>
13 14 15	<pre>int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)</pre>
16 17 18	<pre>int MPI_Pack_external(const char *datarep, const void *inbuf, int incount,</pre>
19 20 21	<pre>int MPI_Pack_external_size(const char *datarep, int incount, MPI_Datatype datatype, MPI_Aint *size)</pre>
22 23	<pre>int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,</pre>
24 25	<pre>int MPI_Type_commit(MPI_Datatype *datatype)</pre>
26 27	<pre>int MPI_Type_contiguous(int count, MPI_Datatype oldtype,</pre>
28 29 30 31 32	<pre>int MPI_Type_create_darray(int size, int rank, int ndims,</pre>
33 34 35	<pre>int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],</pre>
36 37 38 39	<pre>int MPI_Type_create_hindexed_block(int count, int blocklength,</pre>
40 41 42	<pre>int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>
42 43 44 45	<pre>int MPI_Type_create_indexed_block(int count, int blocklength,</pre>
46 47 48	<pre>int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,</pre>

int	<pre>MPI_Type_create_struct(int count, const int array_of_blocklengths[],</pre>	1 2 3
int	<pre>MPI_Type_create_subarray(int ndims, const int array_of_sizes[],</pre>	4 5 6 7
int	MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)	8
int	MPI_Type_free(MPI_Datatype *datatype)	9 10
int	<pre>MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,</pre>	11 12 13 14 15
int	<pre>MPI_Type_get_elements(MPI_Status *status, MPI_Datatype datatype,</pre>	16 17
int	<pre>MPI_Type_get_elements_x(MPI_Status *status, MPI_Datatype datatype, MPI_Count *count)</pre>	18 19 20
int	<pre>MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,</pre>	21 22
int	<pre>MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)</pre>	23 24 25
int	<pre>MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb, MPI_Count *extent)</pre>	26 26 27
int	<pre>MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent)</pre>	28 29 30
int	<pre>MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb, MPI_Count *true_extent)</pre>	30 31 32
int	<pre>MPI_Type_indexed(int count, const int array_of_blocklengths[],</pre>	33 34 35 36
int	MPI_Type_size(MPI_Datatype datatype, int *size)	37
int	MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)	38 39
int	<pre>MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>	40 41
int	<pre>MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf,</pre>	42 43 44
int	<pre>MPI_Unpack_external(const char *datarep, const void *inbuf, MPI_Aint insize, MPI_Aint *position, void *outbuf, int outsize, MPI_Datatype datatype)</pre>	45 46 47 48

1A.2.3 Collective Communication C Bindings $\mathbf{2}$ int MPI_Allgather(const void* sendbuf, int sendcount, 3 MPI_Datatype sendtype, void* recvbuf, int recvcount, 4 MPI_Datatype recvtype, MPI_Comm comm) 56 int MPI_Allgather_init(const void* sendbuf, int sendcount, 7 MPI_Datatype sendtype, void* recvbuf, int recvcount, 8 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 9 MPI_Request *request) 10 int MPI_Allgatherv(const void* sendbuf, int sendcount, 11 MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], 12const int displs[], MPI_Datatype recvtype, MPI_Comm comm) 13 14int MPI_Allgatherv_init(const void* sendbuf, int sendcount, 15MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], 16 const int displs[], MPI_Datatype recvtype, MPI_Comm comm, 17MPI_Info info, MPI_Request* request) 18 int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, 19 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) 2021int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count, 22MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 23MPI_Info info, MPI_Request *request) 24int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, 25void* recvbuf, int recvcount, MPI_Datatype recvtype, 26MPI_Comm comm) 2728int MPI_Alltoall_init(const void* sendbuf, int sendcount, 29 MPI_Datatype sendtype, void* recvbuf, int recvcount, 30 MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, 31MPI_Request *request) 32 int MPI_Alltoallv(const void* sendbuf, const int sendcounts[], 33 const int sdispls[], MPI_Datatype sendtype, void* recvbuf, 34 const int recvcounts[], const int rdispls[], 35 MPI_Datatype recvtype, MPI_Comm comm) 36 37 int MPI_Alltoallv_init(const void* sendbuf, const int sendcounts[], 38 const int sdispls[], MPI_Datatype sendtype, void* recvbuf, 39 const int recvcounts[], const int rdispls[], 40 MPI_Datatype recvtype, MPI_Comm comm, MPI_info info, 41 MPI_Request *request) 42int MPI_Alltoallw(const void* sendbuf, const int sendcounts[], 43 const int sdispls[], const MPI_Datatype sendtypes[], 44 void* recvbuf, const int recvcounts[], const int rdispls[], 45 const MPI_Datatype recvtypes[], MPI_Comm comm) 46 47int MPI_Alltoallw_init(const void* sendbuf, const int sendcounts[], 48

	<pre>const int sdispls[], const MPI_Datatype sendtypes[],</pre>	1
	<pre>void* recvbuf, const int recvcounts[], const int rdispls[],</pre>	2
	<pre>const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,</pre>	3
	MPI_Request *request)	4
int MDT Down	ion(MDI Comm comm)	5
IIIC MPI_Darr	ier(MPI_Comm comm)	6
int MPI_Barr	ier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)	7
int MPT Bcas	t(void* buffer, int count, MPI_Datatype datatype, int root,	8
Int in i_beas	MPI_Comm comm)	9 10
int MPI_Bcas	t_init(void* buffer, int count, MPI_Datatype datatype,	11
	int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)	12
		13
int MPI_Exsc	an(const void* sendbuf, void* recvbuf, int count,	14
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	15
int MPI_Exsc	an_init(const void* sendbuf, void* recvbuf, int count,	16
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	17
	MPI_Info info, MPI_Request *request)	18
int MDT Cath	er(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	19
IIIC MII_Gach	void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,	20
	MPI_Comm comm)	21
		22 23
int MPI_Gath	er_init(const void* sendbuf, int sendcount,	23 24
	<pre>MPI_Datatype sendtype, void* recvbuf, int recvcount,</pre>	25
	MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	26
	MPI_Request *request)	27
int MPI_Gath	erv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	28
_	<pre>void* recvbuf, const int recvcounts[], const int displs[],</pre>	29
	MPI_Datatype recvtype, int root, MPI_Comm comm)	30
int MDT Cath		31
int MPI_Gath	erv_init(const void* sendbuf, int sendcount,	32
	<pre>MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root,</pre>	33
	MPI_Comm comm, MPI_Info info, MPI_Request *request)	34
	Mr1_Comm comm, Mr1_INIO INIO, Mr1_Request *request)	35
int MPI_Iall	gather(const void* sendbuf, int sendcount,	36
	MPI_Datatype sendtype, void* recvbuf, int recvcount,	37
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	38
int MPT Tall	gatherv(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_Request* request)	42
· . WDT T		43
int MPI_Iall	reduce(const void* sendbuf, void* recvbuf, int count,	44
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	45
	MPI_Request *request)	46 47
int MPI_Iall	<pre>toall(const void* sendbuf, int sendcount,</pre>	47
		-10

1MPI_Datatype sendtype, void* recvbuf, int recvcount, $\mathbf{2}$ MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 3 int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[], 4 const int sdispls[], MPI_Datatype sendtype, void* recvbuf, 5const int recvcounts[], const int rdispls[], 6 MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request) 7 8 int MPI_Ialltoallw(const void* sendbuf, const int sendcounts[], 9 const int sdispls[], const MPI_Datatype sendtypes[], 10void* recvbuf, const int recvcounts[], const int rdispls[], 11 const MPI_Datatype recvtypes[], MPI_Comm comm, 12MPI_Request *request) 13 int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request) 1415int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root, 16MPI_Comm comm, MPI_Request *request) 17int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count, 18 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 19 MPI_Request *request) 2021int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, 22 void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, 23MPI_Comm comm, MPI_Request *request) 24int MPI_Igatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, 25void* recvbuf, const int recvcounts[], const int displs[], 26MPI_Datatype recvtype, int root, MPI_Comm comm, 27MPI_Request *request) 2829 int MPI_Ireduce(const void* sendbuf, void* recvbuf, int count, 30 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, 31MPI_Request *request) 32 int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf, 33 const int recvcounts[], MPI_Datatype datatype, MPI_Op op, 34 MPI_Comm comm, MPI_Request *request) 35 36 int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf, 37 int recvcount, MPI_Datatype datatype, MPI_Op op, 38 MPI_Comm comm, MPI_Request *request) 39 int MPI_Iscan(const void* sendbuf, void* recvbuf, int count, 40MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, 41 42MPI_Request *request) 43 int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, 44 void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, 45MPI_Comm comm, MPI_Request *request) 4647int MPI_Iscatterv(const void* sendbuf, const int sendcounts[], 48 const int displs[], MPI_Datatype sendtype, void* recvbuf,

	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)	1 2
int	MPI_Op_commutative(MPI_Op op, int *commute)	3 4
int	MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)	5
int	MPI_Op_free(MPI_Op *op)	6 7
	MPI_Reduce(const void* sendbuf, void* recvbuf, int count,	8
THE	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)	9
int	MPI_Reduce_init(const void* sendbuf, void* recvbuf, int count,	10 11
	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)	12 13
int	MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count, MPI_Datatype datatype, MPI_Op op)	14 15
		16
int	<pre>MPI_Reduce_scatter(const void* sendbuf, void* recvbuf,</pre>	17 18 19
in+		20
IIIC	<pre>MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,</pre>	21 22
int	MPI_Reduce_scatter_block_init(const void* sendbuf, void* recvbuf,	23 24
	<pre>int recvcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	25 26
int	MPI_Reduce_scatter_init(const void* sendbuf, void* recvbuf,	27 28
	const int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)	29 30
int	MPI_Scan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	31 32
int	MPI_Scan_init(const void* sendbuf, void* recvbuf, int count,	33
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)	34 35
int	MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,	36 37
1110	<pre>void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,</pre>	38
	MPI_Comm comm)	39 40
int	MPI_Scatter_init(const void* sendbuf, int sendcount,	41
	<pre>MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,</pre>	42
	MPI_Datatype recytype, int root, MPI_comm comm, MPI_Info info, MPI_Request *request)	43 44
int	MPI_Scatterv(const void* sendbuf, const int sendcounts[],	45
	const int displs[], MPI_Datatype sendtype, void* recvbuf,	46
	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	47
		48

	716	ANNEX A. LANGUAGE BINDINGS SUMMARY
1 2 3 4 5	int MPI	<pre>[_Scatterv_init(const void* sendbuf, const int sendcounts[],</pre>
6 7	A.2.4	Groups, Contexts, Communicators, and Caching C Bindings
8 9 10	int MPI	_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)
11 12 13	int MPI	_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)
14 15 16	int MPI	I_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval, void *attribute_val, void *extra_state)
17	int MPI	<pre>[_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)</pre>
18 19	int MPI	_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
20 21	int MPI	I_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm)
22 23 24 25	int MPI	<pre>[_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn, MPI_Comm_delete_attr_function *comm_delete_attr_fn, int *comm_keyval, void *extra_state)</pre>
26	int MPI	_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
27 28	int MPI	_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
29	int MPI	Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
30 31	int MPI	<pre>[_Comm_free(MPI_Comm *comm)</pre>
32	int MPI	_Comm_free_keyval(int *comm_keyval)
33 34 35	int MPI	<pre>[_Comm_get_attr(MPI_Comm comm, int comm_keyval, void* attribute_val, int *flag)</pre>
36 37	int MPI	_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
38	int MPI	_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
39 40	int MPI	_Comm_group(MPI_Comm comm, MPI_Group *group)
41	int MPI	_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
42 43 44	int MPI	I_Comm_idup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm, MPI_Request *request)
45	int MPI	[_Comm_rank(MPI_Comm comm, int *rank)
46 47	int MPI	_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
48	int MPI	<pre>L_Comm_remote_size(MPI_Comm comm, int *size)</pre>

int	<pre>MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void* attribute_val)</pre>	1
int	MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)	2 3
int	MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)	4
	MPI_Comm_size(MPI_Comm comm, int *size)	5
		6 7
int	<pre>MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)</pre>	8
int	MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,	9
	MPI_Info info, MPI_Comm *newcomm)	10 11
int	<pre>MPI_Comm_test_inter(MPI_Comm comm, int *flag)</pre>	12
int	<pre>MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)</pre>	13
int	MPI_Group_difference(MPI_Group group1, MPI_Group group2,	14
	MPI_Group *newgroup)	15 16
int	<pre>MPI_Group_excl(MPI_Group group, int n, const int ranks[],</pre>	17
	MPI_Group *newgroup)	18
int	MPI_Group_free(MPI_Group *group)	19 20
int	<pre>MPI_Group_incl(MPI_Group group, int n, const int ranks[],</pre>	21
	MPI_Group *newgroup)	22
int	MPI_Group_intersection(MPI_Group group1, MPI_Group group2,	23 24
	MPI_Group *newgroup)	25
int	<pre>MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],</pre>	26
	MPI_Group *newgroup)	27 28
int	<pre>MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],</pre>	28 29
	MPI_Group *newgroup)	30
int	MPI_Group_rank(MPI_Group group, int *rank)	31 32
int	MPI_Group_size(MPI_Group group, int *size)	33
int	<pre>MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[],</pre>	34
	MPI_Group group2, int ranks2[])	35
int	<pre>MPI_Group_union(MPI_Group group1, MPI_Group group2,</pre>	36 37
	MPI_Group *newgroup)	38
int	MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,	39
	MPI_Comm peer_comm, int remote_leader, int tag,	40 41
	MPI_Comm *newintercomm)	42
int	<pre>MPI_Intercomm_merge(MPI_Comm intercomm, int high,</pre>	43
•		44 45
int	<pre>MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval,</pre>	46
	void *attribute_val_out, int *flag)	47
		48

```
1
     int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval,
\mathbf{2}
                   void *extra_state, void *attribute_val_in,
3
                   void *attribute_val_out, int *flag)
4
     int MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval,
5
                   void *attribute_val, void *extra_state)
6
\overline{7}
     int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
8
                   MPI_Type_delete_attr_function *type_delete_attr_fn,
9
                   int *type_keyval, void *extra_state)
10
     int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)
11
12
     int MPI_Type_free_keyval(int *type_keyval)
13
     int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
14
                   void* attribute_val, int *flag)
15
16
     int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,
17
                   int *resultlen)
18
     int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
19
                   void* attribute_val)
20
21
     int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)
22
     int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
23
                   void *attribute_val_in, void *attribute_val_out, int *flag)
^{24}
25
     int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
26
                   void *attribute_val_in, void *attribute_val_out, int *flag)
27
     int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval,
28
                   void *attribute_val, void *extra_state)
29
30
     int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
^{31}
                   MPI_Win_delete_attr_function *win_delete_attr_fn,
32
                   int *win_keyval, void *extra_state)
33
34
     int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
35
     int MPI_Win_free_keyval(int *win_keyval)
36
37
     int MPI_Win_get_attr(MPI_Win win, int win_keyval, void* attribute_val,
38
                   int *flag)
39
     int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
40
41
     int MPI_Win_set_attr(MPI_Win win, int win_keyval, void* attribute_val)
42
     int MPI_Win_set_name(MPI_Win win, const char *win_name)
43
44
45
     A.2.5 Process Topologies C Bindings
46
     int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
47
48
```

int	<pre>MPI_Cart_create(MPI_Comm comm_old, const int ndims, const int dims[],</pre>	1 2
int	<pre>MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[], int coords[])</pre>	3 4 5
int	<pre>MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],</pre>	6 7 8
int	<pre>MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)</pre>	9
int	<pre>MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>	10 11 12
int	<pre>MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)</pre>	12
int	MPI_Cartdim_get(MPI_Comm comm, int *ndims)	14
int	MPI_Dims_create(int nnodes, int ndims, int dims[])	15 16
int	<pre>MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>	17 18 19 20 21
int	<pre>MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,</pre>	22 23 24 25
int	<pre>MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>	26 27 28 29
int	<pre>MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,</pre>	30 31
int	<pre>MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],</pre>	32 33 34
int	<pre>MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[], int edges[])</pre>	35 36
int	<pre>MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],</pre>	37 38 39
int	<pre>MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,</pre>	40 41
int	MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)	42 43
int	MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)	44
int	<pre>MPI_Ineighbor_allgather(const void* sendbuf, int sendcount,</pre>	45 46 47 48

```
1
     int MPI_Ineighbor_allgatherv(const void* sendbuf, int sendcount,
\mathbf{2}
                   MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
3
                   const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
4
                   MPI_Request *request)
5
     int MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount,
6
                   MPI_Datatype sendtype, void* recvbuf, int recvcount,
7
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
8
9
     int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[],
10
                   const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
11
                   const int recvcounts[], const int rdispls[],
12
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
13
     int MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[],
14
                   const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
15
                   void* recvbuf, const int recvcounts[],
16
                   const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
17
                   MPI_Comm comm, MPI_Request *request)
18
19
     int MPI_Neighbor_allgather(const void *sendbuf, int sendcount,
20
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
21
                   MPI_Datatype recvtype, MPI_Comm comm)
22
     int MPI_Neighbor_allgather_init(const void* sendbuf, int sendcount,
23
                   MPI_Datatype sendtype, void* recvbuf, int recvcount,
24
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
25
                   MPI_Request *request)
26
27
     int MPI_Neighbor_allgatherv(const void *sendbuf, int sendcount,
28
                   MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
29
                   const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
30
     int MPI_Neighbor_allgatherv_init(const void* sendbuf, int sendcount,
^{31}
                   MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
32
                   const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
33
                   MPI_Info info, MPI_Request *request)
34
35
     int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,
36
                   MPI_Datatype sendtype, void *recvbuf, int recvcount,
37
                   MPI_Datatype recvtype, MPI_Comm comm)
38
     int MPI_Neighbor_alltoall_init(const void* sendbuf, int sendcount,
39
                   MPI_Datatype sendtype, void* recvbuf, int recvcount,
40
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
41
                   MPI_Request *request)
42
43
     int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[],
44
                   const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
45
                   const int recvcounts[], const int rdispls[],
46
                   MPI_Datatype recvtype, MPI_Comm comm)
47
     int MPI_Neighbor_alltoallv_init(const void* sendbuf,
48
```

<pre>const int sendcounts[], const int sdispls[],</pre>	1
<pre>MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],</pre>	2 3
const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)	4
MFI_INIO INIO, MFI_Request *request)	5
<pre>int MPI_Neighbor_alltoallw(const void* sendbuf, const int sendcounts[],</pre>	6
<pre>const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],</pre>	7
<pre>void* recvbuf, const int recvcounts[],</pre>	8
const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm)	9
	10
<pre>int MPI_Neighbor_alltoallw_init(const void* sendbuf,</pre>	11 12
<pre>const int sendcounts[], const MPI_Aint sdispls[],</pre>	12
<pre>const MPI_Datatype sendtypes[], void* recvbuf, const int recvcounts[], const MPI_Aint rdispls[],</pre>	14
const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,	15
MPI_Request *request)	16
	17
<pre>int MPI_Topo_test(MPI_Comm comm, int *status)</pre>	18
	19
A.2.6 MPI Environmental Management C Bindings	20
double MPI_Wtick(void)	21 22
	23
double MPI_Wtime(void)	24
<pre>int MPI_Abort(MPI_Comm comm, int errorcode)</pre>	25
<pre>int MPI_Add_error_class(int *errorclass)</pre>	26 27
<pre>int MPI_Add_error_code(int errorclass, int *errorcode)</pre>	28
<pre>int MPI_Add_error_string(int errorcode, const char *string)</pre>	29 30
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)	31
<pre>int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)</pre>	32
	33 34
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function *comm_	
MPI_Comm_create_errhandler(MFI_comm_errhandler_runction *comm_ MPI_Errhandler *errhandler)	36 36
	37
<pre>int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)</pre>	38
<pre>int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)</pre>	39 40
int MPI_Errhandler_free(MPI_Errhandler *errhandler)	41
<pre>int MPI_Error_class(int errorcode, int *errorclass)</pre>	42
<pre>int MPI_Error_string(int errorcode, char *string, int *resultlen)</pre>	43 44
int MPI_File_call_errhandler(MPI_File fh, int errorcode)	45
	46
int	47
MPI_File_create_errhandler(MPI_File_errhandler_function *file_	errnandier_in,

1	MPI_Errhandler *errhandler)
2 3	int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
4	int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
5 6	int MPI_Finalize(void)
7	int MPI_Finalized(int *flag)
8 9	<pre>int MPI_Free_mem(void *base)</pre>
10 11	<pre>int MPI_Get_library_version(char *version, int *resultlen)</pre>
11	<pre>int MPI_Get_processor_name(char *name, int *resultlen)</pre>
13 14	<pre>int MPI_Get_version(int *version, int *subversion)</pre>
15	<pre>int MPI_Init(int *argc, char ***argv)</pre>
16 17	<pre>int MPI_Initialized(int *flag)</pre>
18	<pre>int MPI_Win_call_errhandler(MPI_Win win, int errorcode)</pre>
19 20 21 22	int MPI_Win_create_errhandler(MPI_Win_errhandler_function *win_errhandler_fn, MPI_Errhandler *errhandler)
23	int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
24 25	<pre>int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)</pre>
26	
27 28	A.2.7 The Info Object C Bindings
29	<pre>int MPI_Info_create(MPI_Info *info)</pre>
30 31	<pre>int MPI_Info_delete(MPI_Info info, const char *key)</pre>
32 33	int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
34	<pre>int MPI_Info_free(MPI_Info *info)</pre>
35 36	<pre>int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,</pre>
37 38	<pre>int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)</pre>
39	<pre>int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)</pre>
40 41 42	<pre>int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,</pre>
43 44	<pre>int MPI_Info_set(MPI_Info info, const char *key, const char *value)</pre>
45 46	A.2.8 Process Creation and Management C Bindings
47 48	<pre>int MPI_Close_port(const char *port_name)</pre>

int	<pre>MPI_Comm_accept(const char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)</pre>	1 2
int	<pre>MPI_Comm_connect(const char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)</pre>	3 4 5
int	MPI_Comm_disconnect(MPI_Comm *comm)	6
int	MPI_Comm_get_parent(MPI_Comm *parent)	7
	MPI_Comm_join(int fd, MPI_Comm *intercomm)	8 9
		10
int	MPI_Comm_join(int fd, MPI_Comm *intercomm)	11
int	<pre>MPI_Comm_spawn(const char *command, char* argv[], int maxprocs,</pre>	12 13 14
int	<pre>MPI_Comm_spawn_multiple(int count, char* array_of_commands[],</pre>	15 16
1110	char** array_of_argv[], const int array_of_maxprocs[],	10
	<pre>const MPI_Info array_of_info[], int root, MPI_Comm comm,</pre>	18
	<pre>MPI_Comm *intercomm, int array_of_errcodes[])</pre>	19
int	<pre>MPI_Lookup_name(const char *service_name, MPI_Info info,</pre>	20 21
	char *port_name)	21
int	<pre>MPI_Open_port(MPI_Info info, char *port_name)</pre>	23
int	MPI_Publish_name(const char *service_name, MPI_Info info,	24
	<pre>const char *port_name)</pre>	25 26
int	MPI_Unpublish_name(const char *service_name, MPI_Info info,	20
	const char *port_name)	28
		29
A.2.	9 One-Sided Communications C Bindings	30 31
	MPI_Accumulate(const void *origin_addr, int origin_count,	31
THC	MPI_Datatype origin_datatype, int target_rank,	33
	MPI_Aint target_disp, int target_count,	34
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)	35
int	MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,	36 37
	void *result_addr, MPI_Datatype datatype, int target_rank,	37
	MPI_Aint target_disp, MPI_Win win)	39
int	MPI_Fetch_and_op(const void *origin_addr, void *result_addr,	40
	MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,	41
	MPI_Op op, MPI_Win win)	42 43
int	MPI_Get(void *origin_addr, int origin_count,	43 44
	MPI_Datatype origin_datatype, int target_rank,	45
	MPI_Aint target_disp, int target_count,	46
	MPI_Datatype target_datatype, MPI_Win win)	47
		48

1int MPI_Get_accumulate(const void *origin_addr, int origin_count, $\mathbf{2}$ MPI_Datatype origin_datatype, void *result_addr, 3 int result_count, MPI_Datatype result_datatype, 4 int target_rank, MPI_Aint target_disp, int target_count, 5MPI_Datatype target_datatype, MPI_Op op, MPI_Win win) 6 int MPI_Put(const void *origin_addr, int origin_count, 7 MPI_Datatype origin_datatype, int target_rank, 8 MPI_Aint target_disp, int target_count, 9 MPI_Datatype target_datatype, MPI_Win win) 10 11int MPI_Raccumulate(const void *origin_addr, int origin_count, 12MPI_Datatype origin_datatype, int target_rank, 13 MPI_Aint target_disp, int target_count, 14MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, 15MPI_Request *request) 16int MPI_Rget(void *origin_addr, int origin_count, 17MPI_Datatype origin_datatype, int target_rank, 18 MPI_Aint target_disp, int target_count, 19 MPI_Datatype target_datatype, MPI_Win win, 20MPI_Request *request) 2122int MPI_Rget_accumulate(const void *origin_addr, int origin_count, 23MPI_Datatype origin_datatype, void *result_addr, 24int result_count, MPI_Datatype result_datatype, 25int target_rank, MPI_Aint target_disp, int target_count, 26MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, 27MPI_Request *request) 28int MPI_Rput(const void *origin_addr, int origin_count, 29 MPI_Datatype origin_datatype, int target_rank, 30 MPI_Aint target_disp, int target_count, 31MPI_Datatype target_datatype, MPI_Win win, 32 MPI_Request *request) 33 34int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info, 35 MPI_Comm comm, void *baseptr, MPI_Win *win) 36 int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info, 37 MPI_Comm comm, void *baseptr, MPI_Win *win) 38 39 int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size) 40int MPI_Win_complete(MPI_Win win) 41 42int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, 43 MPI_Comm comm, MPI_Win *win) 44 int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win) 4546int MPI_Win_detach(MPI_Win win, const void *base) 4748 int MPI_Win_fence(int assert, MPI_Win win)

int MPI_V	Win_flush(int rank, MPI_Win win)	1
int MPI_V	Win_flush_all(MPI_Win win)	2 3
int MPI_N	Win_flush_local(int rank, MPI_Win win)	4
int MPI_V	Win_flush_local_all(MPI_Win win)	5 6
	Win_free(MPI_Win *win)	7
		8
	Win_get_group(MPI_Win win, MPI_Group *group)	9 10
int MPI_V	Win_get_info(MPI_Win win, MPI_Info *info_used)	11
int MPI_V	Win_lock(int lock_type, int rank, int assert, MPI_Win win)	12
int MPI_V	Win_lock_all(int assert, MPI_Win win)	13 14
int MPI_V	Win_post(MPI_Group group, int assert, MPI_Win win)	15
int MPI_N	Win_set_info(MPI_Win win, MPI_Info info)	16
int MPI_V	Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,	17 18
	int *disp_unit, void *baseptr)	19
int MPI_V	Win_start(MPI_Group group, int assert, MPI_Win win)	20 21
int MPI_V	Win_sync(MPI_Win win)	21
int MPI V	Win_test(MPI_Win win, int *flag)	23
	Win_unlock(int rank, MPI_Win win)	24 25
		26
	Win_unlock_all(MPI_Win win)	27
int MPI_N	Win_wait(MPI_Win win)	28 29
		30
A.2.10 E	External Interfaces C Bindings	31
int MPI_(Grequest_complete(MPI_Request request)	32 33
int MPI_(Grequest_start(MPI_Grequest_query_function *query_fn,	34
	<pre>MPI_Grequest_free_function *free_fn, MPI_Grequest_cancel_function *cancel_fn, void *extra_state,</pre>	35 36
	MPI_Request *request)	37
int MPI_	<pre>Init_thread(int *argc, char ***argv, int required, int *provided)</pre>	38
int MPI	Is_thread_main(int *flag)	39 40
	Query_thread(int *provided)	41
		42
	Status_set_cancelled(MPI_Status *status, int flag)	43 44
int MPI_	<pre>Status_set_elements(MPI_Status *status, MPI_Datatype datatype, int count)</pre>	45
int MDT (46 47
int MPI_)	Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype, MPI_Count count)	47

1	A.2.11 I/O C Bindings
2 3 4	<pre>int MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int count, void *filebuf, MPI_Offset position, void *extra_state)</pre>
5	<pre>int MPI_File_close(MPI_File *fh)</pre>
6 7	<pre>int MPI_File_delete(const char *filename, MPI_Info info)</pre>
8	<pre>int MPI_File_get_amode(MPI_File fh, int *amode)</pre>
9 10	<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>
11 12 13	<pre>int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,</pre>
14	<pre>int MPI_File_get_group(MPI_File fh, MPI_Group *group)</pre>
15 16	<pre>int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)</pre>
17	<pre>int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)</pre>
18 19	<pre>int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)</pre>
20	int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
21 22 23	<pre>int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype, MPI_Aint *extent)</pre>
24 25	int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep)
26 27 28	int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
29 30	int MPI_File_iread_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
31 32 33	<pre>int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>
34 35	<pre>int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>
36 37 38	int MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
39 40 41	int MPI_File_iwrite(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
41 42 43	int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
44 45 46	<pre>int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
40 47 48	<pre>int MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>

int	<pre>MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Request *request)</pre>	1 2
int	<pre>MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)</pre>	3 4 5
int	MPI_File_preallocate(MPI_File fh, MPI_Offset size)	6
int	<pre>MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	7 8 9
int	<pre>MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	10 11
int	<pre>MPI_File_read_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	12 13 14
int	MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)	15
int	<pre>MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	16 17 18
int	<pre>MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	19 20 21
int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>	22 23
int	MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)	24 25
int	<pre>MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	26 27
int	<pre>MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	28 29 30
int	<pre>MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	31
int	MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	32 33 34
int	MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)	35
int	MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)	36 37
int	MPI_File_set_atomicity(MPI_File fh, int flag)	38
int	MPI_File_set_info(MPI_File fh, MPI_Info info)	39 40
int	MPI_File_set_size(MPI_File fh, MPI_Offset size)	41
int	<pre>MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, const char *datarep, MPI_Info info)</pre>	42 43 44
int	MPI_File_sync(MPI_File fh)	45
int	MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	46 47 48

1 int MPI_File_write_all(MPI_File fh, const void *buf, int count, $\mathbf{2}$ MPI_Datatype datatype, MPI_Status *status) 3 int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count, 4 MPI_Datatype datatype) 56 int MPI_File_write_all_end(MPI_File fh, const void *buf, 7 MPI_Status *status) 8 int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf, 9 int count, MPI_Datatype datatype, MPI_Status *status) 10 11 int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf, 12int count, MPI_Datatype datatype, MPI_Status *status) 13int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, 14const void *buf, int count, MPI_Datatype datatype) 1516int MPI_File_write_at_all_end(MPI_File fh, const void *buf, 17MPI_Status *status) 18 int MPI_File_write_ordered(MPI_File fh, const void *buf, int count, 19MPI_Datatype datatype, MPI_Status *status) 2021int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count, 22MPI_Datatype datatype) 23int MPI_File_write_ordered_end(MPI_File fh, const void *buf, 24 MPI_Status *status) 2526int MPI_File_write_shared(MPI_File fh, const void *buf, int count, 27MPI_Datatype datatype, MPI_Status *status) 28int MPI_Register_datarep(const char *datarep, 29MPI_Datarep_conversion_function *read_conversion_fn, 30 MPI_Datarep_conversion_function *write_conversion_fn, 31MPI_Datarep_extent_function *dtype_file_extent_fn, 32 void *extra_state) 33 3435 A.2.12 Language Bindings C Bindings 36 37 int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status) 38 int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status) 39 40int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype) 41 int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype) 4243int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype) 44int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype) 4546MPI_Fint MPI_Comm_c2f(MPI_Comm comm) 47MPI_Comm MPI_Comm_f2c(MPI_Fint comm) 48

MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	1 2
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	3
MPI_Fint MPI_File_c2f(MPI_File file)	4
MPI_File MPI_File_f2c(MPI_Fint file)	5 6
MPI_Fint MPI_Group_c2f(MPI_Group group)	7
MPI_Group MPI_Group_f2c(MPI_Fint group)	8 9
MPI_Fint MPI_Info_c2f(MPI_Info info)	10
MPI_Info MPI_Info_f2c(MPI_Fint info)	11 12
MPI_Fint MPI_Message_c2f(MPI_Message message)	13
MPI_Message MPI_Message_f2c(MPI_Fint message)	14 15
MPI_Fint MPI_Op_c2f(MPI_Op op)	16
	17
MPI_Op MPI_Op_f2c(MPI_Fint op)	18 19
MPI_Fint MPI_Request_c2f(MPI_Request request)	20
MPI_Request MPI_Request_f2c(MPI_Fint request)	21
int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)	22 23
int MPI_Status_c2f08(const MPI_Status *c_status,	24
MPI_F08_status *f08_status)	25
int MPI_Status_f082c(const MPI_F08_status *f08_status, MPI_Status *c_status)	26 27 28
int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)	29
MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)	30
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)	31 32
	33
MPI_Fint MPI_Win_c2f(MPI_Win win)	34
MPI_Win MPI_Win_f2c(MPI_Fint win)	35 36
	37
A.2.13 Tools / Profiling Interface C Bindings	38
<pre>int MPI_Pcontrol(const int level,)</pre>	39 40
	41
A.2.14 Tools / MPI Tool Information Interface C Bindings	42
<pre>int MPI_T_category_changed(int *stamp)</pre>	43
<pre>int MPI_T_category_get_categories(int cat_index, int len, int indices[])</pre>	45
	46
<pre>int MPI_T_category_get_cvars(int cat_index, int len, int indices[])</pre>	47 48
	40

1	int	<pre>MPI_T_category_get_index(const char *name, int *cat_index)</pre>
2 3 4 5	int	<pre>MPI_T_category_get_info(int cat_index, char *name, int *name_len,</pre>
6 7	int	MPI_T_category_get_num(int *num_cat)
8	int	<pre>MPI_T_category_get_pvars(int cat_index, int len, int indices[])</pre>
9 10	int	<pre>MPI_T_cvar_get_index(const char *name, int *cvar_index)</pre>
11 12 13	int	<pre>MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len,</pre>
14 15	int	MPI_T_cvar_get_num(int *num_cvar)
16 17	int	<pre>MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle, MPI_T_cvar_handle *handle, int *count)</pre>
18 19	int	MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
20	int	<pre>MPI_T_cvar_read(MPI_T_cvar_handle handle, void* buf)</pre>
21 22	int	<pre>MPI_T_cvar_write(MPI_T_cvar_handle handle, const void* buf)</pre>
23 24	int	<pre>MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name,</pre>
25 26 27	int	<pre>MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value,</pre>
28	int	MPI_T_finalize(void)
29 30	int	MPI_T_init_thread(int required, int *provided)
31	int	<pre>MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)</pre>
32 33 34 35 36	int	<pre>MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,</pre>
37	int	MPI_T_pvar_get_num(int *num_pvar)
38 39 40	int	<pre>MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index,</pre>
41 42	int	<pre>MPI_T_pvar_handle_free(MPI_T_pvar_session session,</pre>
43 44 45	int	<pre>MPI_T_pvar_read(MPI_T_pvar_session session, MPI_T_pvar_handle handle, void* buf)</pre>
46 47 48	int	<pre>MPI_T_pvar_readreset(MPI_T_pvar_session session,</pre>

int	<pre>MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)</pre>	1
int	<pre>MPI_T_pvar_session_create(MPI_T_pvar_session *session)</pre>	2 3
int	MPI_T_pvar_session_free(MPI_T_pvar_session *session)	4
int	MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)	5 6
int	MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)	7
int	<pre>MPI_T_pvar_write(MPI_T_pvar_session session, MPI_T_pvar_handle handle,</pre>	8 9 10
A.2.	15 Deprecated C Bindings	11 12
int	MPI_Attr_delete(MPI_Comm comm, int keyval)	13 14
int	MPI_Attr_get(MPI_Comm comm, int keyval, void* attribute_val, int *flag)	15
int	MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)	16 17
	MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,	18
	void *attribute_val_in, void *attribute_val_out, int *flag)	19 20
int	MPI_Keyval_create(MPI_Copy_function *copy_fn,	21
	<pre>MPI_Delete_function *delete_fn, int *keyval, woid *outro state)</pre>	22 23
	void *extra_state)	24
int	MPI_Keyval_free(int *keyval)	25
int	MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,	26 27
	<pre>void *attribute_val_in, void *attribute_val_out, int *flag)</pre>	28
int	MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val,	29
	void *extra_state)	30
		31 32
		33
		34
		35
		36
		37
		38 39
		40
		41
		42
		43
		44
		45 46
		40
		48

```
A.3
          Fortran 2008 Bindings with the mpi_f08 Module
1
\mathbf{2}
     A.3.1 Point-to-Point Communication Fortran 2008 Bindings
3
4
     F08 binding
5
     MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
6
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
7
         INTEGER, INTENT(IN) :: count, dest, tag
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
12
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
13
         INTEGER, INTENT(IN) :: count, dest, tag
14
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         TYPE(MPI_Request), INTENT(OUT) :: request
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Buffer_attach(buffer, size, ierror)
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
21
         INTEGER, INTENT(IN) :: size
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_Buffer_detach(buffer_addr, size, ierror)
25
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
26
         TYPE(C_PTR), INTENT(OUT) :: buffer_addr
27
         INTEGER, INTENT(OUT) :: size
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_Cancel(request, ierror)
30
         TYPE(MPI_Request), INTENT(IN) :: request
^{31}
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
32
33
     MPI_Get_count(status, datatype, count, ierror)
34
         TYPE(MPI_Status), INTENT(IN) :: status
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
         INTEGER, INTENT(OUT) :: count
36
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
39
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
40
         INTEGER, INTENT(IN) :: count, dest, tag
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
43
         TYPE(MPI_Request), INTENT(OUT) :: request
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     MPI_Improbe(source, tag, comm, flag, message, status, ierror)
47
         INTEGER, INTENT(IN) :: source, tag
48
```

TYPE(MPI_Comm), INTENT(IN) :: comm LOGICAL, INTENT(OUT) :: flag	1 2
TYPE(MPI_Message), INTENT(OUT) :: message TYPE(MPI_Status) :: status	3 4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5 6
<pre>MPI_Imrecv(buf, count, datatype, message, request, ierror) TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf</pre>	7
INTEGER, INTENT(IN) :: count	8 9
TYPE(MPI_Datatype), INTENT(IN) :: datatype	10
TYPE(MPI_Message), INTENT(INOUT) :: message	11
TYPE(MPI_Request), INTENT(OUT) :: request	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
MPI_Iprobe(source, tag, comm, flag, status, ierror)	14
INTEGER, INTENT(IN) :: source, tag	15
TYPE(MPI_Comm), INTENT(IN) :: comm	16
LOGICAL, INTENT(OUT) :: flag	17
TYPE(MPI_Status) :: status	18 19
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)	20
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	22
INTEGER, INTENT(IN) :: count, source, tag	23
TYPE(MPI_Datatype), INTENT(IN) :: datatype	24
TYPE(MPI_Comm), INTENT(IN) :: comm	25
TYPE(MPI_Request), INTENT(OUT) :: request	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)	28
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	29
INTEGER, INTENT(IN) :: count, dest, tag	30
TYPE(MPI_Datatype), INTENT(IN) :: datatype	31
TYPE(MPI_Comm), INTENT(IN) :: comm	32
TYPE(MPI_Request), INTENT(OUT) :: request	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)	35
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	36 37
INTEGER, INTENT(IN) :: count, dest, tag	37
TYPE(MPI_Datatype), INTENT(IN) :: datatype	39
TYPE(MPI_Comm), INTENT(IN) :: comm	40
TYPE(MPI_Request), INTENT(OUT) :: request	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
MDT Tagond (buf count determs deat top comm request ionnon)	43
<pre>MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf</pre>	44
INTEGER, INTENT(IN) :: count, dest, tag	45
TYPE(MPI_Datatype), INTENT(IN) :: datatype	46
TYPE(MPI_Comm), INTENT(IN) :: comm	47
·,	48

```
1
         TYPE(MPI_Request), INTENT(OUT) :: request
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Mprobe(source, tag, comm, message, status, ierror)
4
         INTEGER, INTENT(IN) :: source, tag
5
         TYPE(MPI_Comm), INTENT(IN) :: comm
6
         TYPE(MPI_Message), INTENT(OUT) :: message
7
         TYPE(MPI_Status) :: status
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
    MPI_Mrecv(buf, count, datatype, message, status, ierror)
11
         TYPE(*), DIMENSION(..) :: buf
12
         INTEGER, INTENT(IN) :: count
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         TYPE(MPI_Message), INTENT(INOUT) :: message
15
         TYPE(MPI_Status) :: status
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Probe(source, tag, comm, status, ierror)
18
         INTEGER, INTENT(IN) :: source, tag
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Status) :: status
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
^{24}
         TYPE(*), DIMENSION(..) :: buf
25
         INTEGER, INTENT(IN) :: count, source, tag
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(MPI_Status) :: status
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
     MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
31
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
32
         INTEGER, INTENT(IN) :: count, source, tag
33
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(MPI_Request), INTENT(OUT) :: request
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Request_free(request, ierror)
39
         TYPE(MPI_Request), INTENT(INOUT) :: request
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     MPI_Request_get_status(request, flag, status, ierror)
42
         TYPE(MPI_Request), INTENT(IN) :: request
43
         LOGICAL, INTENT(OUT) :: flag
44
         TYPE(MPI_Status) :: status
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
48
```

```
TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                 1
                                                                                 2
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                 3
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 5
                                                                                 6
MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                 7
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 8
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                 9
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 11
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 13
                                                                                 14
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
                                                                                 15
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                 16
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                 17
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 18
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 20
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                 21
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 22
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                 23
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 27
                                                                                 28
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
                                                                                 29
             recvcount, recvtype, source, recvtag, comm, status, ierror)
                                                                                 30
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 31
    INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
                                                                                 32
    recvtag
                                                                                 33
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 34
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 36
    TYPE(MPI_Status) :: status
                                                                                 37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 38
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
                                                                                 39
             comm, status, ierror)
                                                                                 40
    TYPE(*), DIMENSION(..) :: buf
                                                                                 41
    INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
                                                                                 42
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 43
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 44
    TYPE(MPI_Status) :: status
                                                                                 45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 46
                                                                                 47
MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
                                                                                 48
```

```
1
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
2
         INTEGER, INTENT(IN) :: count, dest, tag
3
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
    MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
7
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
8
         INTEGER, INTENT(IN) :: count, dest, tag
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
    MPI_Start(request, ierror)
15
         TYPE(MPI_Request), INTENT(INOUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
    MPI_Startall(count, array_of_requests, ierror)
18
         INTEGER, INTENT(IN) :: count
19
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
    MPI_Test(request, flag, status, ierror)
23
         TYPE(MPI_Request), INTENT(INOUT) :: request
24
         LOGICAL, INTENT(OUT) :: flag
25
         TYPE(MPI_Status) :: status
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
    MPI_Test_cancelled(status, flag, ierror)
28
         TYPE(MPI_Status), INTENT(IN) :: status
29
         LOGICAL, INTENT(OUT) :: flag
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
33
         INTEGER, INTENT(IN) :: count
34
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
35
         LOGICAL, INTENT(OUT) :: flag
36
         TYPE(MPI_Status) :: array_of_statuses(*)
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
    MPI_Testany(count, array_of_requests, index, flag, status, ierror)
39
         INTEGER, INTENT(IN) :: count
40
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
41
         INTEGER, INTENT(OUT) :: index
42
         LOGICAL, INTENT(OUT) ::
                                  flag
43
         TYPE(MPI_Status) :: status
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
    MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
47
                  array_of_statuses, ierror)
48
```

```
1
    INTEGER, INTENT(IN) :: incount
                                                                                 2
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                 3
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
                                                                                 4
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 5
                                                                                 6
MPI_Wait(request, status, ierror)
                                                                                 7
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                 8
    TYPE(MPI_Status) :: status
                                                                                 9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 10
                                                                                 11
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: count
                                                                                 12
                                                                                13
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                14
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                 15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 16
MPI_Waitany(count, array_of_requests, index, status, ierror)
                                                                                 17
    INTEGER, INTENT(IN) :: count
                                                                                 18
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                 19
    INTEGER, INTENT(OUT) :: index
                                                                                 20
    TYPE(MPI_Status) :: status
                                                                                 21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 22
                                                                                23
MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
                                                                                 24
             array_of_statuses, ierror)
                                                                                 25
    INTEGER, INTENT(IN) :: incount
                                                                                 26
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                 27
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
                                                                                 28
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                 29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 30
                                                                                 31
A.3.2 Datatypes Fortran 2008 Bindings
                                                                                 32
                                                                                 33
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)
                                                                                 34
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp
                                                                                 35
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2)
                                                                                 36
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2
                                                                                37
                                                                                 38
MPI_Get_address(location, address, ierror)
                                                                                 39
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: location
                                                                                 40
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
                                                                                 41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 42
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
                                                                                 43
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                 44
    INTEGER, INTENT(IN) :: incount, outsize
                                                                                 45
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 46
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                 47
    INTEGER, INTENT(INOUT) :: position
                                                                                 48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
4
                  position, ierror)
5
         CHARACTER(LEN=*), INTENT(IN) :: datarep
6
         TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
7
         INTEGER, INTENT(IN) :: incount
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         TYPE(*), DIMENSION(..) :: outbuf
10
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
11
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
15
         CHARACTER(LEN=*), INTENT(IN) :: datarep
16
         INTEGER, INTENT(IN) :: incount
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
    MPI_Pack_size(incount, datatype, comm, size, ierror)
21
         INTEGER, INTENT(IN) :: incount
22
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         INTEGER, INTENT(OUT) :: size
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
    MPI_Type_commit(datatype, ierror)
28
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
     MPI_Type_contiguous(count, oldtype, newtype, ierror)
31
         INTEGER, INTENT(IN) :: count
32
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
33
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
    MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
37
                  array_of_distribs, array_of_dargs, array_of_psizes, order,
38
                  oldtype, newtype, ierror)
39
         INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
40
         array_of_distribs(ndims), array_of_dargs(ndims),
41
         array_of_psizes(ndims), order
42
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
43
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
    MPI_Type_create_hindexed(count, array_of_blocklengths,
46
                  array_of_displacements, oldtype, newtype, ierror)
47
         INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
48
```

```
1
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                                                 2
    array_of_displacements(count)
                                                                                 3
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                 4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 5
                                                                                 6
MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
                                                                                 7
             oldtype, newtype, ierror)
                                                                                  8
    INTEGER, INTENT(IN) :: count, blocklength
                                                                                 9
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                                                 10
    array_of_displacements(count)
                                                                                 11
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                 12
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                 13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 14
                                                                                 15
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
                                                                                 16
              ierror)
                                                                                 17
    INTEGER, INTENT(IN) :: count, blocklength
                                                                                 18
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
                                                                                 19
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                 20
                                                                                 21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 22
MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
                                                                                 23
             oldtype, newtype, ierror)
                                                                                 24
    INTEGER, INTENT(IN) :: count, blocklength,
                                                                                 25
    array_of_displacements(count)
                                                                                 26
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                 27
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                 28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 29
                                                                                 30
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
                                                                                 31
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                 32
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
                                                                                 33
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                 34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 35
MPI_Type_create_struct(count, array_of_blocklengths,
                                                                                 36
             array_of_displacements, array_of_types, newtype, ierror)
                                                                                 37
    INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
                                                                                 38
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                                                 39
    array_of_displacements(count)
                                                                                 40
    TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
                                                                                 41
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                 42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 43
                                                                                 44
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
                                                                                 45
             array_of_starts, order, oldtype, newtype, ierror)
                                                                                 46
    INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
                                                                                 47
    array_of_subsizes(ndims), array_of_starts(ndims), order
                                                                                 48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
2
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Type_dup(oldtype, newtype, ierror)
5
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
6
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
    MPI_Type_free(datatype, ierror)
10
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
    MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
13
                  array_of_integers, array_of_addresses, array_of_datatypes,
14
                  ierror)
15
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
16
         INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
17
         INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
18
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
19
         array_of_addresses(max_addresses)
20
         TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Type_get_elements(status, datatype, count, ierror)
24
         TYPE(MPI_Status), INTENT(IN) :: status
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         INTEGER, INTENT(OUT) :: count
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Type_get_elements_x(status, datatype, count, ierror)
29
         TYPE(MPI_Status), INTENT(IN) :: status
30
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
35
                  combiner, ierror)
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,
38
         combiner
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Type_get_extent(datatype, lb, extent, ierror)
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Type_get_extent_x(datatype, lb, extent, ierror)
46
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)	2 3
TYPE(MPI_Datatype), INTENT(IN) :: datatype	3 4
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent	5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	8
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,	12
oldtype, newtype, ierror) INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),	13
array_of_displacements(count)	14
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	15
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	16 17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
	19
<pre>MPI_Type_size(datatype, size, ierror) TYPE(MPI_Datatype), INTENT(IN) :: datatype</pre>	20
INTEGER, INTENT(OUT) :: size	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
	23
MPI_Type_size_x(datatype, size, ierror)	24
TYPE(MPI_Datatype), INTENT(IN) :: datatype	25
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size	26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)	28
INTEGER, INTENT(IN) :: count, blocklength, stride	29 30
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	31
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,	34
ierror)	35
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf	36
INTEGER, INTENT(IN) :: insize, outcount	37
INTEGER, INTENT(INOUT) :: position	38
TYPE(*), DIMENSION() :: outbuf	39
TYPE(MPI_Datatype), INTENT(IN) :: datatype	40
TYPE(MPI_Comm), INTENT(IN) :: comm	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outsize,	43 44
datatype, ierror)	44 45
CHARACTER(LEN=*), INTENT(IN) :: datarep	46
TYPE(*), DIMENSION(), INTENT(IN) :: inbuf	47
<pre>INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize</pre>	48

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
2
         TYPE(*), DIMENSION(..) :: outbuf
3
         INTEGER, INTENT(IN) :: outsize
4
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
7
     A.3.3 Collective Communication Fortran 2008 Bindings
8
9
    MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
10
                  comm, ierror)
11
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
12
         TYPE(*), DIMENSION(..) :: recvbuf
13
         INTEGER, INTENT(IN) :: sendcount, recvcount
14
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
18
                  recvtype, comm, info, request, ierror)
19
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
21
         INTEGER, INTENT(IN) :: sendcount, recvcount
22
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         TYPE(MPI_Info), INTENT(IN) :: info
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
29
                  recvtype, comm, ierror)
30
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
31
         TYPE(*), DIMENSION(..) :: recvbuf
32
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
33
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
37
                  displs, recvtype, comm, info, request, ierror)
38
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
39
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
40
         INTEGER, INTENT(IN) :: sendcount
41
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
42
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         TYPE(MPI_Info), INTENT(IN) :: info
45
         TYPE(MPI_Request), INTENT(OUT) :: request
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
                                                                                 2
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 3
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: count
                                                                                 4
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 5
                                                                                 6
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 7
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 8
                                                                                 9
MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
                                                                                10
             request, ierror)
                                                                                11
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                12
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                13
    INTEGER, INTENT(IN) :: count
                                                                                14
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                15
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 16
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 17
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                20
                                                                                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
MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                30
             recvtype, comm, info, request, ierror)
                                                                                31
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                32
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                33
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                34
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                36
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                37
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                39
                                                                                40
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                41
             rdispls, recvtype, comm, ierror)
                                                                                42
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                43
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                44
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                45
    rdispls(*)
                                                                                46
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
3
                  recvcounts, rdispls, recvtype, comm, info, request, ierror)
4
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
7
                   recvcounts(*), rdispls(*)
8
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Info), INTENT(IN) :: info
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
15
                  rdispls, recvtypes, comm, ierror)
16
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
17
         TYPE(*), DIMENSION(..) :: recvbuf
18
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
19
                   rdispls(*)
20
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*)
21
         TYPE(MPI_Datatype), INTENT(IN) :: recvtypes(*)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
25
                  recvcounts, rdispls, recvtypes, comm, info, request, ierror)
26
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
27
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
28
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
29
                   recvcounts(*), rdispls(*)
30
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
31
                   recvtypes(*)
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Info), INTENT(IN) :: info
34
         TYPE(MPI_Request), INTENT(OUT) :: request
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Barrier(comm, ierror)
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Barrier_init(comm, info, request, ierror)
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Info), INTENT(IN) :: info
43
         TYPE(MPI_Request), INTENT(OUT) :: request
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     MPI_Bcast(buffer, count, datatype, root, comm, ierror)
47
         TYPE(*), DIMENSION(..) :: buffer
48
```

```
1
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 2
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 4
                                                                                 5
MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
                                                                                 6
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                 7
    INTEGER, INTENT(IN) :: count, root
                                                                                 8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 9
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 10
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 11
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 13
                                                                                 14
MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
                                                                                 15
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 16
                                                                                 17
    INTEGER, INTENT(IN) :: count
                                                                                 18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 19
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 20
                                                                                 21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 22
MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                                                                                 23
             ierror)
                                                                                 24
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 25
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 26
    INTEGER, INTENT(IN) :: count
                                                                                 27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 28
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 30
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 31
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 33
                                                                                 34
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 35
             root, comm, ierror)
                                                                                 36
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 37
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 38
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                 39
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 42
MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                43
             root, comm, info, request, ierror)
                                                                                 44
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 45
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 46
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                 47
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         TYPE(MPI_Info), INTENT(IN) :: info
3
         TYPE(MPI_Request), INTENT(OUT) :: request
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
6
                  recvtype, root, comm, ierror)
7
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
8
         TYPE(*), DIMENSION(..) :: recvbuf
9
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
10
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
    MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
15
                  recvtype, root, comm, info, request, ierror)
16
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
17
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
18
         INTEGER, INTENT(IN) :: sendcount, root
19
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
20
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         TYPE(MPI_Info), INTENT(IN) :: info
23
         TYPE(MPI_Request), INTENT(OUT) :: request
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
26
                  comm, request, ierror)
27
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
28
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
29
         INTEGER, INTENT(IN) :: sendcount, recvcount
30
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
31
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         TYPE(MPI_Request), INTENT(OUT) :: request
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
36
                  recvtype, comm, request, ierror)
37
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
38
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
39
         INTEGER, INTENT(IN) :: sendcount
40
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
41
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
43
         TYPE(MPI_Request), INTENT(OUT) :: request
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
46
                  ierror)
47
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
48
```

```
TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 1
                                                                                 2
    INTEGER, INTENT(IN) :: count
                                                                                 3
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 4
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 5
                                                                                 6
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 9
             comm, request, ierror)
                                                                                 10
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 11
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 12
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 13
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 14
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 15
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 17
                                                                                 18
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                 19
             rdispls, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 20
                                                                                 21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 22
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                 23
              recvcounts(*), rdispls(*)
                                                                                 24
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 26
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 28
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                 29
             recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                 30
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 31
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 32
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                 33
              recvcounts(*), rdispls(*)
                                                                                 34
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                 35
              recvtypes(*)
                                                                                 36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 37
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 39
                                                                                 40
MPI_Ibarrier(comm, request, ierror)
                                                                                 41
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 42
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 44
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
                                                                                 45
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                 46
    INTEGER, INTENT(IN) :: count, root
                                                                                 47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         TYPE(MPI_Request), INTENT(OUT) :: request
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
6
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
7
         INTEGER, INTENT(IN) :: count
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         TYPE(MPI_Op), INTENT(IN) :: op
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
    MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
15
                  root, comm, request, ierror)
16
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
17
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
18
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
19
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
20
         TYPE(MPI_Comm), INTENT(IN) :: comm
21
         TYPE(MPI_Request), INTENT(OUT) :: request
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
    MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
24
                  recvtype, root, comm, request, ierror)
25
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
         INTEGER, INTENT(IN) :: sendcount, root
28
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
29
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         TYPE(MPI_Request), INTENT(OUT) :: request
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
35
                  ierror)
36
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
38
         INTEGER, INTENT(IN) :: count, root
39
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
         TYPE(MPI_Op), INTENT(IN) :: op
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Request), INTENT(OUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
45
                  request, ierror)
46
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
47
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
48
```

```
1
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                 2
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 3
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 4
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 5
                                                                                 6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 7
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                 8
             request, ierror)
                                                                                 9
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 10
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                 11
    INTEGER, INTENT(IN) :: recvcount
                                                                                 12
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 13
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 14
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 15
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 17
                                                                                 18
MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                                                                                 19
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 20
                                                                                 21
    INTEGER, INTENT(IN) :: count
                                                                                 22
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 23
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 27
MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                 28
             root, comm, request, ierror)
                                                                                 29
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 30
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 31
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                 32
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 33
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 34
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 36
                                                                                 37
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                 38
             recvtype, root, comm, request, ierror)
                                                                                 39
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 40
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 41
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
                                                                                 42
    INTEGER, INTENT(IN) :: recvcount, root
                                                                                 43
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 45
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 47
MPI_Op_commutative(op, commute, ierror)
                                                                                 48
```

```
1
         TYPE(MPI_Op), INTENT(IN) :: op
2
         LOGICAL, INTENT(OUT) :: commute
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Op_create(user_fn, commute, op, ierror)
5
         PROCEDURE(MPI_User_function) :: user_fn
6
         LOGICAL, INTENT(IN) :: commute
7
         TYPE(MPI_Op), INTENT(OUT) :: op
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10
    MPI_Op_free(op, ierror)
11
         TYPE(MPI_Op), INTENT(INOUT) :: op
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
14
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
15
         TYPE(*), DIMENSION(..) :: recvbuf
16
         INTEGER, INTENT(IN) :: count, root
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Op), INTENT(IN) :: op
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
23
                  request, ierror)
^{24}
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
25
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
26
         INTEGER, INTENT(IN) :: count, root
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         TYPE(MPI_Op), INTENT(IN) :: op
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         TYPE(MPI_Info), INTENT(IN) :: info
31
         TYPE(MPI_Request), INTENT(OUT) :: request
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
34
         TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
35
         TYPE(*), DIMENSION(..) :: inoutbuf
36
         INTEGER, INTENT(IN) :: count
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         TYPE(MPI_Op), INTENT(IN) :: op
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
42
                  ierror)
43
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
44
         TYPE(*), DIMENSION(..) :: recvbuf
45
         INTEGER, INTENT(IN) :: recvcounts(*)
46
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
         TYPE(MPI_Op), INTENT(IN) :: op
48
```

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
             ierror)
                                                                                 5
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 6
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 7
    INTEGER, INTENT(IN) :: recvcount
                                                                                 8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 9
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 12
                                                                                 13
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
                                                                                 14
             comm, info, request, ierror)
                                                                                 15
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 16
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 17
    INTEGER, INTENT(IN) :: recvcount
                                                                                 18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 19
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 20
                                                                                 21
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 24
MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                 25
             info, request, ierror)
                                                                                 26
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 27
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 28
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                 29
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 30
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 32
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 35
                                                                                 36
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
                                                                                 37
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 38
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 39
    INTEGER, INTENT(IN) :: count
                                                                                 40
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 41
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 44
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                                                                                 45
             ierror)
                                                                                 46
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 48
```

```
1
         INTEGER, INTENT(IN) :: count
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Op), INTENT(IN) :: op
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Info), INTENT(IN) :: info
6
         TYPE(MPI_Request), INTENT(OUT) :: request
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
    MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
9
                  root, comm, ierror)
10
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
11
         TYPE(*), DIMENSION(..) :: recvbuf
12
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
13
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
18
                  recvtype, root, comm, info, request, ierror)
19
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
21
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
22
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
^{24}
         TYPE(MPI_Info), INTENT(IN) :: info
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
28
                  recvtype, root, comm, ierror)
29
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
30
         TYPE(*), DIMENSION(..) :: recvbuf
31
         INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
32
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
37
                  recvcount, recvtype, root, comm, info, request, ierror)
38
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
39
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
40
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
41
         INTEGER, INTENT(IN) :: recvcount, root
42
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         TYPE(MPI_Info), INTENT(IN) :: info
45
         TYPE(MPI_Request), INTENT(OUT) :: request
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
A.3.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings
                                                                                 1
                                                                                 2
MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
                                                                                 3
             attribute_val_out, flag, ierror)
                                                                                 4
    TYPE(MPI_Comm) :: oldcomm
                                                                                 5
    INTEGER :: comm_keyval
                                                                                 6
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
                                                                                 7
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
                                                                                 8
    LOGICAL :: flag
                                                                                 9
    INTEGER :: ierror
                                                                                 10
                                                                                 11
MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
                                                                                 12
                                                                                 13
    TYPE(MPI_Comm) :: oldcomm
    INTEGER :: comm_keyval
                                                                                 14
                                                                                 15
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
                                                                                 16
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
                                                                                 17
    LOGICAL :: flag
                                                                                 18
    INTEGER :: ierror
                                                                                 19
MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
                                                                                 20
             ierror)
                                                                                 21
    TYPE(MPI_Comm) :: comm
                                                                                 22
    INTEGER :: comm_keyval
                                                                                 23
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                 24
    INTEGER :: ierror
                                                                                 25
                                                                                 26
MPI_Comm_compare(comm1, comm2, result, ierror)
                                                                                 27
    TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
                                                                                 28
    INTEGER, INTENT(OUT) :: result
                                                                                 29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 30
MPI_Comm_create(comm, group, newcomm, ierror)
                                                                                 31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 32
    TYPE(MPI_Group), INTENT(IN) :: group
                                                                                 33
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                 34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 35
                                                                                 36
MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
                                                                                 37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 38
    TYPE(MPI_Group), INTENT(IN) :: group
                                                                                 39
    INTEGER, INTENT(IN) :: tag
                                                                                 40
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                 41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 42
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
                                                                                 43
             extra_state, ierror)
                                                                                 44
    PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn
                                                                                 45
    PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn
                                                                                 46
    INTEGER, INTENT(OUT) :: comm_keyval
                                                                                 47
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                 48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Comm_delete_attr(comm, comm_keyval, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, INTENT(IN) :: comm_keyval
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
     MPI_Comm_dup(comm, newcomm, ierror)
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Info), INTENT(IN) :: info
14
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Comm_free(comm, ierror)
18
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_Comm_free_keyval(comm_keyval, ierror)
21
         INTEGER, INTENT(INOUT) :: comm_keyval
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         INTEGER, INTENT(IN) :: comm_keyval
27
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
28
         LOGICAL, INTENT(OUT) :: flag
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
    MPI_Comm_get_info(comm, info_used, ierror)
^{31}
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         TYPE(MPI_Info), INTENT(OUT) :: info_used
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
38
         INTEGER, INTENT(OUT) :: resultlen
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Comm_group(comm, group, ierror)
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Group), INTENT(OUT) :: group
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Comm_idup(comm, newcomm, request, ierror)
46
         TYPE(MPI_Comm), INTENT(IN) :: comm
47
         TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
48
```

TYPE(MPI_Request), INTENT(OUT) :: request	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2
MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)	4
TYPE(MPI_Comm), INTENT(IN) :: comm	5
TYPE(MPI_Info), INTENT(IN) :: info	6
TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm	7
TYPE(MPI_Request), INTENT(OUT) :: request	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
MPI_Comm_rank(comm, rank, ierror)	10
TYPE(MPI_Comm), INTENT(IN) :: comm	11
INTEGER, INTENT(OUT) :: rank	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
MPI_Comm_remote_group(comm, group, ierror)	14 15
TYPE(MPI_Comm), INTENT(IN) :: comm	15
TYPE(MPI_Group), INTENT(OUT) :: group	10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18
MPI_Comm_remote_size(comm, size, ierror)	19
TYPE(MPI_Comm), INTENT(IN) :: comm	20
INTEGER, INTENT(OUT) :: size	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
MDT (amm act atta (acmm acmm barned) atta ibuta well i amaan)	23
<pre>MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm</pre>	24
INTEGER, INTENT(IN) :: comm_keyval	25
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val	26 27
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
	29
<pre>MPI_Comm_set_info(comm, info, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm</pre>	30
TYPE(MPI_COMMI), INTENT(IN) COMM TYPE(MPI_Info), INTENT(IN) :: info	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
	33
MPI_Comm_set_name(comm, comm_name, ierror)	34
TYPE(MPI_Comm), INTENT(IN) :: comm CHARACTER(LEN=*), INTENT(IN) :: comm_name	35
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	36
INTEGER, OFFICIARE, INTENT(001) TETTOT	37 38
MPI_Comm_size(comm, size, ierror)	38 39
TYPE(MPI_Comm), INTENT(IN) :: comm	40
INTEGER, INTENT(OUT) :: size	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
MPI_Comm_split(comm, color, key, newcomm, ierror)	43
TYPE(MPI_Comm), INTENT(IN) :: comm	44
INTEGER, INTENT(IN) :: color, key	45
TYPE(MPI_Comm), INTENT(OUT) :: newcomm	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
	48

```
1
     MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
\mathbf{2}
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         INTEGER, INTENT(IN) :: split_type, key
4
         TYPE(MPI_Info), INTENT(IN) :: info
5
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Comm_test_inter(comm, flag, ierror)
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         LOGICAL, INTENT(OUT) :: flag
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Group_compare(group1, group2, result, ierror)
13
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
14
         INTEGER, INTENT(OUT) :: result
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
     MPI_Group_difference(group1, group2, newgroup, ierror)
17
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
18
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Group_excl(group, n, ranks, newgroup, ierror)
22
         TYPE(MPI_Group), INTENT(IN) :: group
23
         INTEGER, INTENT(IN) :: n, ranks(n)
24
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_Group_free(group, ierror)
27
         TYPE(MPI_Group), INTENT(INOUT) :: group
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Group_incl(group, n, ranks, newgroup, ierror)
^{31}
         TYPE(MPI_Group), INTENT(IN) :: group
32
         INTEGER, INTENT(IN) :: n, ranks(n)
33
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Group_intersection(group1, group2, newgroup, ierror)
36
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
37
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
41
         TYPE(MPI_Group), INTENT(IN) :: group
42
         INTEGER, INTENT(IN) :: n, ranges(3,n)
43
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
    MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
46
         TYPE(MPI_Group), INTENT(IN) :: group
47
         INTEGER, INTENT(IN) :: n, ranges(3,n)
48
```

```
1
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 2
                                                                                 3
MPI_Group_rank(group, rank, ierror)
                                                                                 4
    TYPE(MPI_Group), INTENT(IN) :: group
                                                                                 5
    INTEGER, INTENT(OUT) :: rank
                                                                                 6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 7
                                                                                 8
MPI_Group_size(group, size, ierror)
                                                                                 9
    TYPE(MPI_Group), INTENT(IN) :: group
                                                                                 10
    INTEGER, INTENT(OUT) :: size
                                                                                 11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 12
MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
                                                                                 13
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
                                                                                 14
    INTEGER, INTENT(IN) :: n, ranks1(n)
                                                                                 15
    INTEGER, INTENT(OUT) :: ranks2(n)
                                                                                 16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 17
                                                                                 18
MPI_Group_union(group1, group2, newgroup, ierror)
                                                                                 19
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
                                                                                 20
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
                                                                                 21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 22
MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
                                                                                 23
             tag, newintercomm, ierror)
                                                                                 24
    TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
                                                                                 25
    INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
                                                                                 26
    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
                                                                                 27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 28
                                                                                 29
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
                                                                                 30
    TYPE(MPI_Comm), INTENT(IN) :: intercomm
                                                                                 31
    LOGICAL, INTENT(IN) :: high
                                                                                 32
    TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
                                                                                 33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 34
MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
                                                                                 35
             attribute_val_out, flag, ierror)
                                                                                 36
    TYPE(MPI_Datatype) :: oldtype
                                                                                 37
    INTEGER :: type_keyval
                                                                                 38
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
                                                                                 39
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
                                                                                 40
    LOGICAL :: flag
                                                                                 41
    INTEGER :: ierror
                                                                                 42
                                                                                 43
MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
                                                                                 44
             attribute_val_out, flag, ierror)
                                                                                 45
    TYPE(MPI_Datatype) :: oldtype
                                                                                 46
    INTEGER :: type_keyval
                                                                                 47
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
                                                                                 48
```

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
2
         LOGICAL :: flag
3
         INTEGER :: ierror
4
     MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
5
                  ierror)
6
         TYPE(MPI_Datatype) :: datatype
7
         INTEGER :: type_keyval
8
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
9
         INTEGER, INTENT(OUT) :: ierror
10
11
    MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
12
                  extra_state, ierror)
13
         PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
14
         PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
15
         INTEGER, INTENT(OUT) :: type_keyval
16
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
    MPI_Type_delete_attr(datatype, type_keyval, ierror)
19
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
         INTEGER, INTENT(IN) :: type_keyval
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
    MPI_Type_free_keyval(type_keyval, ierror)
24
         INTEGER, INTENT(INOUT) :: type_keyval
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
    MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         INTEGER, INTENT(IN) :: type_keyval
29
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
30
         LOGICAL, INTENT(OUT) :: flag
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Type_get_name(datatype, type_name, resultlen, ierror)
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
36
         INTEGER, INTENT(OUT) :: resultlen
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
    MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
39
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
         INTEGER, INTENT(IN) :: type_keyval
41
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Type_set_name(datatype, type_name, ierror)
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         CHARACTER(LEN=*), INTENT(IN) :: type_name
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
                                                                                2
             attribute_val_out, flag, ierror)
                                                                                 3
    TYPE(MPI_Win) :: oldwin
    INTEGER :: win_keyval
                                                                                4
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
                                                                                5
                                                                                6
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
                                                                                7
    LOGICAL :: flag
    INTEGER :: ierror
                                                                                9
MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
                                                                                10
             attribute_val_out, flag, ierror)
                                                                                11
    TYPE(MPI_Win) :: oldwin
                                                                                12
    INTEGER :: win_keyval
                                                                                13
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
                                                                                14
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
                                                                                15
    LOGICAL :: flag
                                                                                16
    INTEGER :: ierror
                                                                                17
                                                                                18
MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
                                                                                19
    TYPE(MPI_Win) :: win
    INTEGER :: win_keyval
                                                                                20
                                                                                21
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                22
    INTEGER :: ierror
                                                                                23
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
                                                                                24
             extra_state, ierror)
                                                                                25
    PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
                                                                                26
    PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
                                                                                27
    INTEGER, INTENT(OUT) :: win_keyval
                                                                                28
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                30
                                                                                31
MPI_Win_delete_attr(win, win_keyval, ierror)
                                                                                32
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                33
    INTEGER, INTENT(IN) :: win_keyval
                                                                                34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                35
MPI_Win_free_keyval(win_keyval, ierror)
                                                                                36
    INTEGER, INTENT(INOUT) :: win_keyval
                                                                                37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                38
                                                                                39
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
                                                                                40
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                41
    INTEGER, INTENT(IN) :: win_keyval
                                                                                42
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
                                                                                43
    LOGICAL, INTENT(OUT) :: flag
                                                                                44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                45
MPI_Win_get_name(win, win_name, resultlen, ierror)
                                                                                46
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                47
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
                                                                                48
```

```
1
         INTEGER, INTENT(OUT) :: resultlen
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
4
         TYPE(MPI_Win), INTENT(IN) :: win
5
         INTEGER, INTENT(IN) :: win_keyval
6
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
9
     MPI_Win_set_name(win, win_name, ierror)
10
         TYPE(MPI_Win), INTENT(IN) :: win
11
         CHARACTER(LEN=*), INTENT(IN) :: win_name
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     A.3.5 Process Topologies Fortran 2008 Bindings
15
16
    MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
17
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         INTEGER, INTENT(IN) :: rank, maxdims
19
         INTEGER, INTENT(OUT) :: coords(maxdims)
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
23
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
24
         LOGICAL, INTENT(IN) :: periods(ndims), reorder
25
         TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         INTEGER, INTENT(IN) :: maxdims
31
         INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
32
         LOGICAL, INTENT(OUT) :: periods(maxdims)
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
37
         INTEGER, INTENT(IN) :: ndims, dims(ndims)
         LOGICAL, INTENT(IN) :: periods(ndims)
38
         INTEGER, INTENT(OUT) :: newrank
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Cart_rank(comm, coords, rank, ierror)
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
43
         INTEGER, INTENT(IN) :: coords(*)
44
         INTEGER, INTENT(OUT) :: rank
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
48
         TYPE(MPI_Comm), INTENT(IN) :: comm
```

1 INTEGER, INTENT(IN) :: direction, disp 2 INTEGER, INTENT(OUT) :: rank_source, rank_dest INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4 MPI_Cart_sub(comm, remain_dims, newcomm, ierror) 5 TYPE(MPI_Comm), INTENT(IN) :: comm 6 LOGICAL, INTENT(IN) :: remain_dims(*) 7 TYPE(MPI_Comm), INTENT(OUT) :: newcomm 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10 MPI_Cartdim_get(comm, ndims, ierror) 11 TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER, INTENT(OUT) :: ndims 1213 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 14MPI_Dims_create(nnodes, ndims, dims, ierror) 15INTEGER, INTENT(IN) :: nnodes, ndims 16INTEGER, INTENT(INOUT) :: dims(ndims) 17INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 19 MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights, 20info, reorder, comm_dist_graph, ierror) 21TYPE(MPI_Comm), INTENT(IN) :: comm_old 22 INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*), 23weights(*) 24TYPE(MPI_Info), INTENT(IN) :: info 25LOGICAL, INTENT(IN) :: reorder 26TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28 MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights, 29 outdegree, destinations, destweights, info, reorder, 30 comm_dist_graph, ierror) 31TYPE(MPI_Comm), INTENT(IN) :: comm_old 32 INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*), 33 outdegree, destinations(outdegree), destweights(*) 34 TYPE(MPI_Info), INTENT(IN) :: info 35LOGICAL, INTENT(IN) :: reorder 36 TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights, 40 maxoutdegree, destinations, destweights, ierror) 41 TYPE(MPI_Comm), INTENT(IN) :: comm 42INTEGER, INTENT(IN) :: maxindegree, maxoutdegree 43 INTEGER, INTENT(OUT) :: sources(maxindegree), 44 destinations(maxoutdegree) 45INTEGER :: sourceweights(*), destweights(*) 46INTEGER, OPTIONAL, INTENT(OUT) :: ierror 47 MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror) 48

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         INTEGER, INTENT(OUT) :: indegree, outdegree
3
         LOGICAL, INTENT(OUT) :: weighted
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
6
                  ierror)
7
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
8
         INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
9
         LOGICAL, INTENT(IN) :: reorder
10
         TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         INTEGER, INTENT(IN) :: maxindex, maxedges
16
         INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
    MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
21
         INTEGER, INTENT(OUT) :: newrank
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
     MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         INTEGER, INTENT(IN) :: rank, maxneighbors
27
         INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, INTENT(IN) :: rank
32
         INTEGER, INTENT(OUT) :: nneighbors
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Graphdims_get(comm, nnodes, nedges, ierror)
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         INTEGER, INTENT(OUT) :: nnodes, nedges
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
40
                  recvtype, comm, request, ierror)
41
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
42
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
43
         INTEGER, INTENT(IN) :: sendcount, recvcount
44
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
45
         TYPE(MPI_Comm), INTENT(IN) :: comm
46
         TYPE(MPI_Request), INTENT(OUT) :: request
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                 2
             displs, recvtype, comm, request, ierror)
                                                                                 3
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 4
    INTEGER, INTENT(IN) :: sendcount
                                                                                 5
                                                                                 6
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                 7
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 8
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 10
                                                                                 11
MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                 12
             recvtype, comm, request, ierror)
                                                                                 13
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 15
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 16
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 17
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 20
                                                                                 21
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                 22
             recvcounts, rdispls, recvtype, comm, request, ierror)
                                                                                 23
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 24
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 25
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                 26
              recvcounts(*), rdispls(*)
                                                                                 27
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 28
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 29
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 30
                                                                                 31
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                 32
             recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                 33
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                 34
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                 35
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                 36
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                 37
              sdispls(*), rdispls(*)
                                                                                 38
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                 39
              recvtypes(*)
                                                                                 40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 41
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 43
                                                                                 44
MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                 45
             recvtype, comm, ierror)
                                                                                 46
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 47
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2
         TYPE(*), DIMENSION(..) :: recvbuf
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
6
                  recvcount, recvtype, comm, info, request, ierror)
7
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
9
         INTEGER, INTENT(IN) :: sendcount, recvcount
10
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Info), INTENT(IN) :: info
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
17
                  displs, recvtype, comm, ierror)
18
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
19
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
20
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21
         TYPE(*), DIMENSION(..) :: recvbuf
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
25
                  recvcounts, displs, recvtype, comm, info, request, ierror)
26
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
27
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
28
         INTEGER, INTENT(IN) :: sendcount
29
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
30
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
31
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         TYPE(MPI_Info), INTENT(IN) :: info
33
         TYPE(MPI_Request), INTENT(OUT) :: request
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
37
                  recvtype, comm, ierror)
38
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
39
         INTEGER, INTENT(IN) :: sendcount, recvcount
40
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
41
         TYPE(*), DIMENSION(..) :: recvbuf
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
45
                  recvcount, recvtype, comm, info, request, ierror)
46
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
47
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
48
```

```
1
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                 2
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                 3
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 4
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 5
                                                                                 6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 7
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                 8
             recvcounts, rdispls, recvtype, comm, ierror)
                                                                                 9
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 10
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                 11
    rdispls(*)
                                                                                12
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                13
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                14
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 16
                                                                                17
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
                                                                                18
             recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
                                                                                19
             ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                20
                                                                                21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                22
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                23
              recvcounts(*), rdispls(*)
                                                                                24
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 26
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                27
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                28
                                                                                29
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                30
             recvcounts, rdispls, recvtypes, comm, ierror)
                                                                                31
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                 32
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                 33
    INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
                                                                                34
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                35
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                38
                                                                                39
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
                                                                                40
             recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
                                                                                41
             ierror)
                                                                                42
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                43
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                44
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                45
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                46
              sdispls(*), rdispls(*)
                                                                                 47
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                48
```

```
1
                   recvtypes(*)
2
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         TYPE(MPI_Info), INTENT(IN) :: info
4
         TYPE(MPI_Request), INTENT(OUT) :: request
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Topo_test(comm, status, ierror)
7
         TYPE(MPI_Comm), INTENT(IN) :: comm
8
         INTEGER, INTENT(OUT) :: status
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
12
     A.3.6 MPI Environmental Management Fortran 2008 Bindings
13
    DOUBLE PRECISION MPI_Wtick()
14
15
     DOUBLE PRECISION MPI_Wtime()
16
17
     MPI_Abort(comm, errorcode, ierror)
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         INTEGER, INTENT(IN) :: errorcode
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Add_error_class(errorclass, ierror)
22
         INTEGER, INTENT(OUT) :: errorclass
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Add_error_code(errorclass, errorcode, ierror)
25
26
         INTEGER, INTENT(IN) :: errorclass
         INTEGER, INTENT(OUT) :: errorcode
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_Add_error_string(errorcode, string, ierror)
30
         INTEGER, INTENT(IN) :: errorcode
^{31}
         CHARACTER(LEN=*), INTENT(IN) :: string
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Alloc_mem(size, info, baseptr, ierror)
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
35
36
         INTEGER(KIND=MPI_ADDRESS_KIND, INTENT(IN) :: size
37
         TYPE(MPI_Info), INTENT(IN) :: info
38
         TYPE(C_PTR), INTENT(OUT) :: baseptr
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Comm_call_errhandler(comm, errorcode, ierror)
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         INTEGER, INTENT(IN) :: errorcode
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
46
         PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
47
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
48
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI_Comm_get_errhandler(comm, errhandler, ierror)	1
TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4
MPI_Comm_set_errhandler(comm, errhandler, ierror)	5
TYPE(MPI_Comm), INTENT(IN) :: comm	6 7
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	8
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
MPI_Errhandler_free(errhandler, ierror)	10
TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
MPI_Error_class(errorcode, errorclass, ierror)	13 14
INTEGER, INTENT(IN) :: errorcode	15
INTEGER, INTENT(OUT) :: errorclass	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
MPI_Error_string(errorcode, string, resultlen, ierror)	18
INTEGER, INTENT(IN) :: errorcode	19 20
CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string INTEGER, INTENT(OUT) :: resultlen	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
	23
<pre>MPI_File_call_errhandler(fh, errorcode, ierror) TYPE(MPI_File), INTENT(IN) :: fh</pre>	24
INTEGER, INTENT(IN) :: errorcode	25 26
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)	28
PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn	29
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	30
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	31
MPI_File_get_errhandler(file, errhandler, ierror)	33
TYPE(MPI_File), INTENT(IN) :: file	34
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	35
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	36
<pre>MPI_File_set_errhandler(file, errhandler, ierror)</pre>	37 38
TYPE(MPI_File), INTENT(IN) :: file	39
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
	41
MPI_Finalize(ierror)	42
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	43
MPI_Finalized(flag, ierror)	44 45
LOGICAL, INTENT(OUT) :: flag	46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	47
MPI_Free_mem(base, ierror)	48

```
1
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: base
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Get_library_version(version, resultlen, ierror)
4
         CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
5
         INTEGER, INTENT(OUT) :: resultlen
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Get_processor_name(name, resultlen, ierror)
9
         CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
10
         INTEGER, INTENT(OUT) :: resultlen
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Get_version(version, subversion, ierror)
13
         INTEGER, INTENT(OUT) :: version, subversion
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Init(ierror)
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
    MPI_Initialized(flag, ierror)
19
         LOGICAL, INTENT(OUT) :: flag
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     MPI_Win_call_errhandler(win, errorcode, ierror)
23
         TYPE(MPI_Win), INTENT(IN) :: win
^{24}
         INTEGER, INTENT(IN) :: errorcode
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
27
         PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
28
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Win_get_errhandler(win, errhandler, ierror)
32
         TYPE(MPI_Win), INTENT(IN) :: win
33
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Win_set_errhandler(win, errhandler, ierror)
36
         TYPE(MPI_Win), INTENT(IN) :: win
37
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
41
     A.3.7 The Info Object Fortran 2008 Bindings
42
43
     MPI_Info_create(info, ierror)
44
         TYPE(MPI_Info), INTENT(OUT) :: info
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
    MPI_Info_delete(info, key, ierror)
47
         TYPE(MPI_Info), INTENT(IN) :: info
48
```

CHARACTER(LEN=*), INTENT(IN) :: key INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$\frac{1}{2}$
<pre>MPI_Info_dup(info, newinfo, ierror) TYPE(MPI_Info), INTENT(IN) :: info</pre>	$\frac{3}{4}$
TYPE(MPI_INIO), INTENI(IN) :: INIO TYPE(MPI_Info), INTENT(OUT) :: newinfo	5 6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	7
MPI_Info_free(info, ierror)	8
TYPE(MPI_Info), INTENT(INOUT) :: info	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10 11
MPI_Info_get(info, key, valuelen, value, flag, ierror)	12
TYPE(MPI_Info), INTENT(IN) :: info CHARACTER(LEN=*), INTENT(IN) :: key	13
INTEGER, INTENT(IN) :: valuelen	14 15
CHARACTER(LEN=*), INTENT(OUT) :: value	16
LOGICAL, INTENT(OUT) :: flag	17
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	18 19
MPI_Info_get_nkeys(info, nkeys, ierror)	19 20
TYPE(MPI_Info), INTENT(IN) :: info INTEGER, INTENT(OUT) :: nkeys	21
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
MPI_Info_get_nthkey(info, n, key, ierror)	23 24
TYPE(MPI_Info), INTENT(IN) :: info	24
INTEGER, INTENT(IN) :: n	26
CHARACTER(LEN=*), INTENT(OUT) :: key INTEGER, OPTIONAL, INTENT(OUT) :: ierror	27
	28 29
<pre>MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) TYPE(MPI_Info), INTENT(IN) :: info</pre>	30
CHARACTER(LEN=*), INTENT(IN) :: key	31
INTEGER, INTENT(OUT) :: valuelen	32 33
LOGICAL, INTENT(OUT) :: flag	33 34
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	35
MPI_Info_set(info, key, value, ierror)	36
TYPE(MPI_Info), INTENT(IN) :: info CHARACTER(LEN=*), INTENT(IN) :: key, value	37 38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
	40
A.3.8 Process Creation and Management Fortran 2008 Bindings	41
MPI_Close_port(port_name, ierror)	42 43
CHARACTER(LEN=*), INTENT(IN) :: port_name	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)	$46 \\ 47$
CHARACTER(LEN=*), INTENT(IN) :: port_name	47

```
1
         TYPE(MPI_Info), INTENT(IN) :: info
2
         INTEGER, INTENT(IN) :: root
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
7
         CHARACTER(LEN=*), INTENT(IN) :: port_name
8
         TYPE(MPI_Info), INTENT(IN) :: info
9
         INTEGER, INTENT(IN) :: root
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Comm_disconnect(comm, ierror)
15
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Comm_get_parent(parent, ierror)
18
         TYPE(MPI_Comm), INTENT(OUT) :: parent
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Comm_join(fd, intercomm, ierror)
22
         INTEGER, INTENT(IN) :: fd
23
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
^{24}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Comm_join(fd, intercomm, ierror)
26
         INTEGER, INTENT(IN) :: fd
27
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
^{31}
                  array_of_errcodes, ierror)
32
         CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
33
         INTEGER, INTENT(IN) :: maxprocs, root
34
         TYPE(MPI_Info), INTENT(IN) :: info
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
37
         INTEGER :: array_of_errcodes(*)
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
40
                  array_of_maxprocs, array_of_info, root, comm, intercomm,
41
                  array_of_errcodes, ierror)
42
         INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
43
         CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
44
         array_of_argv(count,*)
45
         TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
46
         TYPE(MPI_Comm), INTENT(IN) :: comm
47
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
48
```

1 INTEGER :: array_of_errcodes(*) $\mathbf{2}$ INTEGER, OPTIONAL, INTENT(OUT) :: ierror 3 MPI_Lookup_name(service_name, info, port_name, ierror) 4 CHARACTER(LEN=*), INTENT(IN) :: service_name 5TYPE(MPI_Info), INTENT(IN) :: info 6 CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 9 MPI_Open_port(info, port_name, ierror) 10 TYPE(MPI_Info), INTENT(IN) :: info 11 CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI_Publish_name(service_name, info, port_name, ierror) 14CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name 15TYPE(MPI_Info), INTENT(IN) :: info 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1718 MPI_Unpublish_name(service_name, info, port_name, ierror) 19 CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name 20TYPE(MPI_Info), INTENT(IN) :: info 21INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22 23A.3.9 One-Sided Communications Fortran 2008 Bindings 2425MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank, 26target_disp, target_count, target_datatype, op, win, ierror) 27TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 28 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count 29TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype 30 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 31TYPE(MPI_Op), INTENT(IN) :: op 32 TYPE(MPI_Win), INTENT(IN) :: win 33 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 34 MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype, 35 target_rank, target_disp, win, ierror) 36 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 37 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: compare_addr 38 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr 39 TYPE(MPI_Datatype), INTENT(IN) :: datatype 40INTEGER, INTENT(IN) :: target_rank 41 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp 42TYPE(MPI_Win), INTENT(IN) :: win 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44 45MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank, 46target_disp, op, win, ierror) 47TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr 48

```
1
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         INTEGER, INTENT(IN) :: target_rank
4
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
5
         TYPE(MPI_Op), INTENT(IN) :: op
6
         TYPE(MPI_Win), INTENT(IN) :: win
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
    MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
9
                  target_disp, target_count, target_datatype, win, ierror)
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
11
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
12
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
13
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
14
         TYPE(MPI_Win), INTENT(IN) :: win
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
18
                  result_count, result_datatype, target_rank, target_disp,
19
                  target_count, target_datatype, op, win, ierror)
20
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
21
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
22
         INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
23
                   target_count
^{24}
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
25
                   result_datatype
26
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
27
         TYPE(MPI_Op), INTENT(IN) :: op
28
         TYPE(MPI_Win), INTENT(IN) :: win
29
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                            ierror
30
     MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
31
                  target_disp, target_count, target_datatype, win, ierror)
32
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
33
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
34
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
35
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
36
         TYPE(MPI_Win), INTENT(IN) :: win
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
40
                  target_disp, target_count, target_datatype, op, win, request,
41
                  ierror)
42
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
43
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
44
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
45
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
46
         TYPE(MPI_Op), INTENT(IN) :: op
47
         TYPE(MPI_Win), INTENT(IN) :: win
48
```

```
1
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 3
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                 4
             target_disp, target_count, target_datatype, win, request,
                                                                                 5
             ierror)
                                                                                 6
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                 7
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                 8
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                 9
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                 10
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                 11
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 13
                                                                                 14
MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
                                                                                 15
             result_addr, result_count, result_datatype, target_rank,
                                                                                 16
             target_disp, target_count, target_datatype, op, win, request,
                                                                                 17
             ierror)
                                                                                 18
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                 19
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
                                                                                 20
                                                                                 21
              target_count
                                                                                 22
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
                                                                                 23
              result_datatype
                                                                                 24
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                 25
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                 26
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                 27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 29
MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                 30
             target_disp, target_count, target_datatype, win, request,
                                                                                 31
             ierror)
                                                                                 32
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                 33
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                 34
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                 35
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                 36
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                 37
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 39
                                                                                 40
MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
                                                                                 41
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                 42
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
                                                                                 43
    INTEGER, INTENT(IN) :: disp_unit
                                                                                 44
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 45
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                 46
    TYPE(C_PTR), INTENT(OUT) :: baseptr
                                                                                 47
    TYPE(MPI_Win), INTENT(OUT) :: win
                                                                                 48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
3
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
4
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
5
         INTEGER, INTENT(IN) :: disp_unit
6
         TYPE(MPI_Info), INTENT(IN) :: info
7
         TYPE(MPI_Comm), INTENT(IN) :: comm
8
         TYPE(C_PTR), INTENT(OUT) :: baseptr
9
         TYPE(MPI_Win), INTENT(OUT) :: win
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Win_attach(win, base, size, ierror)
13
         TYPE(MPI_Win), INTENT(IN) :: win
14
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
15
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
     MPI_Win_complete(win, ierror)
18
         TYPE(MPI_Win), INTENT(IN) :: win
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
22
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
23
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
^{24}
         INTEGER, INTENT(IN) :: disp_unit
25
         TYPE(MPI_Info), INTENT(IN) :: info
26
         TYPE(MPI_Comm), INTENT(IN) :: comm
27
         TYPE(MPI_Win), INTENT(OUT) :: win
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_Win_create_dynamic(info, comm, win, ierror)
30
         TYPE(MPI_Info), INTENT(IN) :: info
^{31}
         TYPE(MPI_Comm), INTENT(IN) ::
                                         comm
32
         TYPE(MPI_Win), INTENT(OUT) :: win
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Win_detach(win, base, ierror)
36
         TYPE(MPI_Win), INTENT(IN) :: win
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Win_fence(assert, win, ierror)
40
         INTEGER, INTENT(IN) :: assert
41
         TYPE(MPI_Win), INTENT(IN) :: win
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Win_flush(rank, win, ierror)
45
         INTEGER, INTENT(IN) :: rank
46
         TYPE(MPI_Win), INTENT(IN) :: win
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

<pre>MPI_Win_flush_all(win, ierror) TYPE(MPI_Win), INTENT(IN) :: win</pre>	1 2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	$\frac{3}{4}$
<pre>MPI_Win_flush_local(rank, win, ierror) INTEGER, INTENT(IN) :: rank TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	4 5 6 7 8
<pre>MPI_Win_flush_local_all(win, ierror) TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	9 10 11
<pre>MPI_Win_free(win, ierror) TYPE(MPI_Win), INTENT(INOUT) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	12 13 14 15
<pre>MPI_Win_get_group(win, group, ierror) TYPE(MPI_Win), INTENT(IN) :: win TYPE(MPI_Group), INTENT(OUT) :: group INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	16 17 18 19
<pre>MPI_Win_get_info(win, info_used, ierror) TYPE(MPI_Win), INTENT(IN) :: win TYPE(MPI_Info), INTENT(OUT) :: info_used INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	 20 21 22 23 24
<pre>MPI_Win_lock(lock_type, rank, assert, win, ierror) INTEGER, INTENT(IN) :: lock_type, rank, assert TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	25 26 27 28
<pre>MPI_Win_lock_all(assert, win, ierror) INTEGER, INTENT(IN) :: assert TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	29 30 31 32 33
<pre>MPI_Win_post(group, assert, win, ierror) TYPE(MPI_Group), INTENT(IN) :: group INTEGER, INTENT(IN) :: assert TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	34 35 36 37 38
<pre>MPI_Win_set_info(win, info, ierror) TYPE(MPI_Win), INTENT(IN) :: win TYPE(MPI_Info), INTENT(IN) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	 39 40 41 42 43
<pre>MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(MPI_Win), INTENT(IN) :: win INTEGER, INTENT(IN) :: rank</pre>	44 45 46 47 48

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
2
         INTEGER, INTENT(OUT) :: disp_unit
3
         TYPE(C_PTR), INTENT(OUT) :: baseptr
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Win_start(group, assert, win, ierror)
6
         TYPE(MPI_Group), INTENT(IN) :: group
7
         INTEGER, INTENT(IN) :: assert
8
         TYPE(MPI_Win), INTENT(IN) :: win
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Win_sync(win, ierror)
12
         TYPE(MPI_Win), INTENT(IN) :: win
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     MPI_Win_test(win, flag, ierror)
15
         TYPE(MPI_Win), INTENT(IN) :: win
16
         LOGICAL, INTENT(OUT) :: flag
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
     MPI_Win_unlock(rank, win, ierror)
20
         INTEGER, INTENT(IN) :: rank
21
         TYPE(MPI_Win), INTENT(IN) ::
                                       win
22
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
23
     MPI_Win_unlock_all(win, ierror)
24
         TYPE(MPI_Win), INTENT(IN) :: win
25
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
26
27
     MPI_Win_wait(win, ierror)
28
         TYPE(MPI_Win), INTENT(IN) :: win
29
         INTEGER, OPTIONAL, INTENT(OUT) ::
                                             ierror
30
^{31}
     A.3.10 External Interfaces Fortran 2008 Bindings
32
33
    MPI_Grequest_complete(request, ierror)
34
         TYPE(MPI_Request), INTENT(IN) :: request
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
38
                   ierror)
         PROCEDURE(MPI_Grequest_query_function) :: query_fn
39
         PROCEDURE(MPI_Grequest_free_function) :: free_fn
40
41
         PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
         TYPE(MPI_Request), INTENT(OUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Init_thread(required, provided, ierror)
46
         INTEGER, INTENT(IN) :: required
47
         INTEGER, INTENT(OUT) :: provided
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
<pre>MPI_Is_thread_main(flag, ierror) LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	2 3 4
<pre>MPI_Query_thread(provided, ierror) INTEGER, INTENT(OUT) :: provided INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	5 6 7 8
<pre>MPI_Status_set_cancelled(status, flag, ierror) TYPE(MPI_Status), INTENT(INOUT) :: status LOGICAL, INTENT(IN) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	9 10 11 12 13
<pre>MPI_Status_set_elements(status, datatype, count, ierror) TYPE(MPI_Status), INTENT(INOUT) :: status TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: count INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	14 15 16 17 18 19
<pre>MPI_Status_set_elements_x(status, datatype, count, ierror) TYPE(MPI_Status), INTENT(INOUT) :: status TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	20 21 22 23 24 25
A.3.11 I/O Fortran 2008 Bindings	26
<pre>MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,</pre>	27 28 29 30 31 32 33 34 35 36
<pre>MPI_File_close(fh, ierror) TYPE(MPI_File), INTENT(INOUT) :: fh INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	37 38 39
<pre>MPI_File_delete(filename, info, ierror) CHARACTER(LEN=*), INTENT(IN) :: filename TYPE(MPI_Info), INTENT(IN) :: info INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	40 41 42 43
<pre>MPI_File_get_amode(fh, amode, ierror) TYPE(MPI_File), INTENT(IN) :: fh INTEGER, INTENT(OUT) :: amode INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>	44 45 46 47 48

```
1
     MPI_File_get_atomicity(fh, flag, ierror)
\mathbf{2}
         TYPE(MPI_File), INTENT(IN) :: fh
3
         LOGICAL, INTENT(OUT) :: flag
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_File_get_byte_offset(fh, offset, disp, ierror)
6
         TYPE(MPI_File), INTENT(IN) :: fh
7
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
8
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_File_get_group(fh, group, ierror)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         TYPE(MPI_Group), INTENT(OUT) :: group
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
     MPI_File_get_info(fh, info_used, ierror)
16
         TYPE(MPI_File), INTENT(IN) :: fh
17
         TYPE(MPI_Info), INTENT(OUT) :: info_used
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_File_get_position(fh, offset, ierror)
21
         TYPE(MPI_File), INTENT(IN) :: fh
22
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_File_get_position_shared(fh, offset, ierror)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
    MPI_File_get_size(fh, size, ierror)
30
         TYPE(MPI_File), INTENT(IN) :: fh
31
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
    MPI_File_get_type_extent(fh, datatype, extent, ierror)
34
         TYPE(MPI_File), INTENT(IN) :: fh
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
40
         TYPE(MPI_File), INTENT(IN) :: fh
41
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
42
         TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
43
         CHARACTER(LEN=*), INTENT(OUT) :: datarep
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_File_iread(fh, buf, count, datatype, request, ierror)
46
         TYPE(MPI_File), INTENT(IN) :: fh
47
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
48
```

```
1
    INTEGER, INTENT(IN) :: count
                                                                                 2
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 3
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 4
                                                                                 5
MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
                                                                                 6
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 7
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 8
    INTEGER, INTENT(IN) :: count
                                                                                 9
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 10
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 12
                                                                                 13
MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
                                                                                 14
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 15
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                 16
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 17
    INTEGER, INTENT(IN) :: count
                                                                                 18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 19
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 20
                                                                                 21
MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)
                                                                                 22
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 23
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                 24
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 25
    INTEGER, INTENT(IN) :: count
                                                                                 26
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 29
                                                                                 30
MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
                                                                                 31
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 32
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                 33
    INTEGER, INTENT(IN) :: count
                                                                                 34
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 35
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 37
MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
                                                                                 38
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 39
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 40
    INTEGER, INTENT(IN) :: count
                                                                                 41
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 42
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                 43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 44
                                                                                 45
MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
                                                                                 46
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 47
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 48
```

```
1
         INTEGER, INTENT(IN) :: count
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Request), INTENT(OUT) ::
                                            request
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
6
         TYPE(MPI_File), INTENT(IN) :: fh
7
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
8
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
9
         INTEGER, INTENT(IN) :: count
10
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
    MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror)
15
         TYPE(MPI_File), INTENT(IN) :: fh
16
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
17
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
18
         INTEGER, INTENT(IN) :: count
19
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
    MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
23
         TYPE(MPI_File), INTENT(IN) :: fh
24
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
25
         INTEGER, INTENT(IN) :: count
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_File_open(comm, filename, amode, info, fh, ierror)
31
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         CHARACTER(LEN=*), INTENT(IN) :: filename
33
         INTEGER, INTENT(IN) :: amode
34
         TYPE(MPI_Info), INTENT(IN) :: info
35
         TYPE(MPI_File), INTENT(OUT) :: fh
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
    MPI_File_preallocate(fh, size, ierror)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_File_read(fh, buf, count, datatype, status, ierror)
43
         TYPE(MPI_File), INTENT(IN) :: fh
44
         TYPE(*), DIMENSION(..) :: buf
45
         INTEGER, INTENT(IN) :: count
46
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
         TYPE(MPI_Status) :: status
48
```

1 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2 MPI_File_read_all(fh, buf, count, datatype, status, ierror) 3 TYPE(MPI_File), INTENT(IN) :: fh 4 TYPE(*), DIMENSION(..) :: buf 5INTEGER, INTENT(IN) :: count 6 TYPE(MPI_Datatype), INTENT(IN) :: datatype 7 TYPE(MPI_Status) :: status 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10MPI_File_read_all_begin(fh, buf, count, datatype, ierror) 11 TYPE(MPI_File), INTENT(IN) :: fh TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 1213 INTEGER, INTENT(IN) :: count 14TYPE(MPI_Datatype), INTENT(IN) :: datatype 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 16MPI_File_read_all_end(fh, buf, status, ierror) 17TYPE(MPI_File), INTENT(IN) :: fh 18 TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 19 TYPE(MPI_Status) :: status 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122 MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) 23TYPE(MPI_File), INTENT(IN) :: fh 24INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 25TYPE(*), DIMENSION(..) :: buf 26INTEGER, INTENT(IN) :: count 27TYPE(MPI_Datatype), INTENT(IN) :: datatype 28 TYPE(MPI_Status) :: status 29INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror) 31TYPE(MPI_File), INTENT(IN) :: fh 32 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 33 TYPE(*), DIMENSION(..) :: buf 34 INTEGER, INTENT(IN) :: count 35 TYPE(MPI_Datatype), INTENT(IN) :: datatype 36 TYPE(MPI_Status) :: status 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror) 40TYPE(MPI_File), INTENT(IN) :: fh 41 INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset 42TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf 43 INTEGER, INTENT(IN) :: count 44TYPE(MPI_Datatype), INTENT(IN) :: datatype 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46MPI_File_read_at_all_end(fh, buf, status, ierror) 47TYPE(MPI_File), INTENT(IN) :: fh 48

```
1
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
2
         TYPE(MPI_Status) :: status
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
5
         TYPE(MPI_File), INTENT(IN) :: fh
6
         TYPE(*), DIMENSION(..) :: buf
7
         INTEGER, INTENT(IN) :: count
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         TYPE(MPI_Status) :: status
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_File_read_ordered_begin(fh, buf, count, datatype, 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
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
    MPI_File_read_ordered_end(fh, buf, status, ierror)
19
         TYPE(MPI_File), INTENT(IN) :: fh
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
21
         TYPE(MPI_Status) :: status
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
24
    MPI_File_read_shared(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
29
         TYPE(MPI_Status) :: status
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
    MPI_File_seek(fh, offset, whence, ierror)
32
         TYPE(MPI_File), INTENT(IN) :: fh
33
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
34
         INTEGER, INTENT(IN) :: whence
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
    MPI_File_seek_shared(fh, offset, whence, ierror)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
40
         INTEGER, INTENT(IN) :: whence
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
    MPI_File_set_atomicity(fh, flag, ierror)
43
         TYPE(MPI_File), INTENT(IN) :: fh
44
         LOGICAL, INTENT(IN) :: flag
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
    MPI_File_set_info(fh, info, ierror)
48
```

```
TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 1
                                                                                 2
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 4
MPI_File_set_size(fh, size, ierror)
                                                                                 5
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 6
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
                                                                                 7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 8
                                                                                 9
MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
                                                                                 10
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 11
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
    TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
                                                                                 12
                                                                                 13
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                 14
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                 15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 16
MPI_File_sync(fh, ierror)
                                                                                 17
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 19
                                                                                 20
MPI_File_write(fh, buf, count, datatype, status, ierror)
                                                                                 21
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 22
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                 23
    INTEGER, INTENT(IN) :: count
                                                                                 24
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 25
    TYPE(MPI_Status) :: status
                                                                                 26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 27
MPI_File_write_all(fh, buf, count, datatype, status, ierror)
                                                                                 28
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 29
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                 30
    INTEGER, INTENT(IN) :: count
                                                                                 31
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 32
    TYPE(MPI_Status) :: status
                                                                                 33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 34
                                                                                 35
MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
                                                                                 36
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 37
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 38
    INTEGER, INTENT(IN) :: count
                                                                                 39
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 41
MPI_File_write_all_end(fh, buf, status, ierror)
                                                                                 42
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 43
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 44
    TYPE(MPI Status) :: status
                                                                                 45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 46
                                                                                 47
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
                                                                                 48
```

```
1
         TYPE(MPI_File), INTENT(IN) :: fh
2
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
3
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
4
         INTEGER, INTENT(IN) :: count
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         TYPE(MPI_Status) :: status
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
    MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
9
         TYPE(MPI_File), INTENT(IN) :: fh
10
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
11
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
12
         INTEGER, INTENT(IN) :: count
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         TYPE(MPI_Status) :: status
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
18
         TYPE(MPI_File), INTENT(IN) :: fh
19
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
20
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
21
         INTEGER, INTENT(IN) :: count
22
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
    MPI_File_write_at_all_end(fh, buf, status, ierror)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
27
         TYPE(MPI_Status) :: status
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
31
         TYPE(MPI_File), INTENT(IN) :: fh
32
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
33
         INTEGER, INTENT(IN) :: count
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Status) :: status
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
    MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
40
         INTEGER, INTENT(IN) :: count
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
    MPI_File_write_ordered_end(fh, buf, status, ierror)
45
         TYPE(MPI_File), INTENT(IN) :: fh
46
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
47
         TYPE(MPI_Status) :: status
48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 2
MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 4
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                 5
    INTEGER, INTENT(IN) :: count
                                                                                 6
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 7
    TYPE(MPI_Status) :: status
                                                                                 8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 9
                                                                                 10
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
                                                                                 11
             dtype_file_extent_fn, extra_state, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                 12
                                                                                 13
    PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
                                                                                 14
    PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
                                                                                 15
    PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
                                                                                 16
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                 17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 18
                                                                                 19
A.3.12 Language Bindings Fortran 2008 Bindings
                                                                                 20
                                                                                 21
MPI_F_sync_reg(buf)
                                                                                 22
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
                                                                                 23
MPI_Sizeof(x, size, ierror)
                                                                                 24
    TYPE(*), DIMENSION(..) :: x
                                                                                 25
    INTEGER, INTENT(OUT) :: size
                                                                                 26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 27
                                                                                 28
MPI_Status_f082f(f08_status, f_status, ierror)
                                                                                 29
    TYPE(MPI_Status), INTENT(IN) :: f08_status
                                                                                 30
    INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
                                                                                 31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 32
MPI_Status_f2f08(f_status, f08_status, ierror)
                                                                                 33
    INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
                                                                                 34
    TYPE(MPI_Status), INTENT(OUT) :: f08_status
                                                                                 35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 36
                                                                                 37
MPI_Type_create_f90_complex(p, r, newtype, ierror)
                                                                                 38
    INTEGER, INTENT(IN) :: p, r
                                                                                 39
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                 40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 41
                                                                                 42
MPI_Type_create_f90_integer(r, newtype, ierror)
    INTEGER, INTENT(IN) :: r
                                                                                 43
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                 44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 45
                                                                                 46
MPI_Type_create_f90_real(p, r, newtype, ierror)
                                                                                 47
    INTEGER, INTENT(IN) :: p, r
                                                                                 48
```

```
1
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
\mathbf{2}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Type_match_size(typeclass, size, datatype, ierror)
4
          INTEGER, INTENT(IN) :: typeclass, size
\mathbf{5}
          TYPE(MPI_Datatype), INTENT(OUT) :: datatype
6
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
9
     A.3.13 Tools / Profiling Interface Fortran 2008 Bindings
10
     MPI_Pcontrol(level)
11
          INTEGER, INTENT(IN) :: level
12
13
14
15
16
17
18
19
20
21
22
23
^{24}
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

A.4. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	787
A.4 Fortran Bindings with mpif.h or the mpi Module	1
A.4.1 Point-to-Point Communication Fortran Bindings	2
Ŭ	4
F binding MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	5
<pre><type> BUF(*)</type></pre>	6
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	7
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	9
<type> BUF(*)</type>	10
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	11
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)	12 13
<type> BUFFER(*)</type>	13
INTEGER SIZE, IERROR	15
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)	16
INTEGER(KIND=MPI_ADDRESS_KIND) BUFFER_ADDR	17
INTEGER SIZE, IERROR	18 19
MPI_CANCEL(REQUEST, IERROR)	20
INTEGER REQUEST, IERROR	21
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)	22
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	23
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	24 25
<type> BUF(*)</type>	26
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	27
MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)	28
INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR	29 30
LOGICAL FLAG	30
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)	32
<type> BUF(*)</type>	33
INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR	34
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)	35
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR	36 37
LOGICAL FLAG	38
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)	39
<pre><type> BUF(*) INTEGED COUNT DATATYDE SOURCE TAC COMM DEGUEST LEDDOD</type></pre>	40
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR	41 42
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	42
<type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</type>	44
	45
<pre>MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre>	46
<pre>INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR</pre>	47 48
,,,,,	-40

```
1
    MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
\mathbf{2}
         <type> BUF(*)
3
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
4
     MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)
5
         INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
6
\overline{7}
     MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
8
         <type> BUF(*)
9
         INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
10
     MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
11
         INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
12
13
    MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
14
         <type> BUF(*)
15
         INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
16
         IERROR
17
    MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
18
         <type> BUF(*)
19
         INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
20
21
    MPI_REQUEST_FREE(REQUEST, IERROR)
22
         INTEGER REQUEST, IERROR
23
     MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
24
         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
25
         LOGICAL FLAG
26
27
    MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
28
         <type> BUF(*)
29
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
30
     MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
^{31}
         <type> BUF(*)
32
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
33
34
     MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
35
         <type> BUF(*)
36
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
37
     MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
38
                   RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
39
         <type> SENDBUF(*), RECVBUF(*)
40
         INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
41
         SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
42
43
     MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
44
                   COMM, STATUS, IERROR)
45
         <type> BUF(*)
46
         INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
47
         STATUS(MPI_STATUS_SIZE), IERROR
48
```

788

<pre>MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)</pre>	1 2
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	3
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	4 5
<type> BUF(*)</type>	6
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR	7
MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	8
<pre><type> BUF(*)</type></pre>	9
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR	10
MPI_START(REQUEST, IERROR)	11
INTEGER REQUEST, IERROR	12
	13 14
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)	15
INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR	16
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)	17
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR	18
LOGICAL FLAG	19
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)	20
INTEGER COUNT, ARRAY_OF_REQUESTS(*),	21
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR	22 23
LOGICAL FLAG	23 24
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)	25
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),	26
IERROR	27
LOGICAL FLAG	28
MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,	29 30
ARRAY_OF_STATUSES, IERROR)	31
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR	32
ARRAI_OF_SIAIOSES(MFI_SIAIOS_SIZE,*), IERROR	33
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)	34
INTEGER STATUS(MPI_STATUS_SIZE), IERROR	35
LOGICAL FLAG	36
MPI_WAIT(REQUEST, STATUS, IERROR)	37 38
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR	39
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)	40
INTEGER COUNT, ARRAY_OF_REQUESTS(*),	41
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR	42
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)	43
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),	44
IERROR	45
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,	46 47
ARRAY_OF_STATUSES, IERROR)	47
manin_or_siniosos, industry	

```
1
         INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
\mathbf{2}
         ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
3
4
     A.4.2 Datatypes Fortran Bindings
5
6
     INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP)
7
         INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP
8
     INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)
9
         INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2
10
11
     MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
12
         <type> LOCATION(*)
13
         INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
14
         INTEGER IERROR
15
     MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
16
         <type> INBUF(*), OUTBUF(*)
17
         INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
18
19
     MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
20
                   POSITION, IERROR)
21
         CHARACTER*(*) DATAREP
22
         <type> INBUF(*), OUTBUF(*)
23
         INTEGER INCOUNT, DATATYPE, IERROR
24
         INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
25
26
    MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
         CHARACTER*(*) DATAREP
27
         INTEGER INCOUNT, DATATYPE, IERROR
28
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
29
30
     MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
^{31}
         INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
32
33
     MPI_TYPE_COMMIT(DATATYPE, IERROR)
34
         INTEGER DATATYPE, IERROR
35
     MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
36
         INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
37
38
     MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
39
                   ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
40
                   OLDTYPE, NEWTYPE, IERROR)
41
         INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
42
         ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR
43
     MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
44
                   ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
45
         INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
46
         INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
47
48
```

MFI_TIFE_CREATE_HINDEXED_DLUCK(COUNT, DLUCKLENGTH, AMAAT_OF_DT5FLACEMENT5,	1 2
	3
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)	4
	5 6
	7
	8
INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE	9
MFI_IIFE_CREATE_INDEXED_DEUCK(COUNT, DEUCKLENGIN, ARRAI_UF_DISPLACEMENTS,	10 11
	12
NEWTYPE, IERROR	13
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)	14
TNTECED OF NEWTYDE TEDDOD	15
TNTECED (VIND-MDT ADDDESS VIND) ID EVTENT	16
	$17 \\ 18$
MPI_IYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLUCKLENGTHS,	19
ARRAY_UF_DISPLACEMENTS, ARRAY_UF_TYPES, NEWTYPE, TERROR)	20
INIEGER COUNI, ARRAY_OF_BLOCKLENGINS(*), ARRAY_OF_IYPES(*), NEWIYPE,	21
	22
	23
	24
	25
	26
	27
MPI_IYPE_DUP(ULDIYPE, NEWIYPE, IERRUR)	28
INTEGER OLDTYPE, NEWTYPE, IERROR	29 30
MPI_TYPE_FREE(DATATYPE, IERROR)	31
INTEGER DATATYPE, IERROR	32
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	33
	34
	35
INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	36
ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	37
INTEGER(KIND=MP1_ADDRESS_KIND) ARRAY_UF_ADDRESSES(*)	38 39
	40
	41
	42
MPI_TYPE_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)	43
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR	44
INTEGER(KIND=MPI_COUNT_KIND) COUNT	45
	$\frac{46}{47}$
	48

1	IERROR
2 3 4 5	MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
6 7 8	MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT
9 10 11 12	MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
13 14 15	MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT
16 17 18 19 20	<pre>MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR</pre>
21 22 23	MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, SIZE, IERROR
23 24 25 26	MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) SIZE
27 28 29	MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
30 31 32 33	<pre>MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,</pre>
34 35 36 37 38 39 40 41	<pre>MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTSIZE, DATATYPE, IERROR) CHARACTER*(*) DATAREP <type> INBUF(*), OUTBUF(*) INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION INTEGER OUTSIZE, DATATYPE, IERROR</type></pre>
42	A.4.3 Collective Communication Fortran Bindings
43 44 45 46 47 48	<pre>MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR</type></pre>

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	1
RECVTYPE, COMM, IERROR)	2
<type> SENDBUF(*), RECVBUF(*)</type>	3
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	4
IERROR	5
MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,	6
DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)	7
<type> SENDBUF(*), RECVBUF(*)</type>	8
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	9 10
INFO, REQUEST, IERROR	10
	11
MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	12
RECVTYPE, COMM, INFO, REQUEST, IERROR)	13
<pre><type> SENDBUF(*), RECVBUF(*) </type></pre>	15
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR	16
LERKUR	17
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)	18
<type> SENDBUF(*), RECVBUF(*)</type>	19
INTEGER COUNT, DATATYPE, OP, COMM, IERROR	20
MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,	21
REQUEST, IERROR)	22
<type> SENDBUF(*), RECVBUF(*)</type>	23
INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR	24
	25
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	26
COMM, IERROR)	27
<type> SENDBUF(*), RECVBUF(*)</type>	28
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	29
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,	30
RDISPLS, RECVTYPE, COMM, IERROR)	31
<type> SENDBUF(*), RECVBUF(*)</type>	32
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	33
RECVTYPE, COMM, IERROR	34
MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,	35
RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)	36 37
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	37 38
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	38 39
RECVTYPE, COMM, INFO, REQUEST, IERROR	40
	41
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,	42
RDISPLS, RECVTYPES, COMM, IERROR)	43
<type> SENDBUF(*), RECVBUF(*)</type>	44
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),	45
RDISPLS(*), RECVTYPES(*), COMM, IERROR	46
MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,	47
RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)	48

```
1
         <type> SENDBUF(*), RECVBUF(*)
\mathbf{2}
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
3
                    RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR
4
     MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
5
                   RECVTYPE, COMM, INFO, REQUEST, IERROR)
6
         <type> SENDBUF(*), RECVBUF(*)
7
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
8
                    IERROR
9
10
    MPI_BARRIER(COMM, IERROR)
11
         INTEGER COMM, IERROR
12
    MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)
13
         INTEGER COMM, INFO, REQUEST, IERROR
14
15
    MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
16
         <type> BUFFER(*)
17
         INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
18
    MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)
19
         <type> BUFFER(*)
20
         INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR
21
22
     MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
23
         <type> SENDBUF(*), RECVBUF(*)
^{24}
         INTEGER COUNT, DATATYPE, OP, COMM, IERROR
25
     MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
26
                   IERROR)
27
         <type> SENDBUF(*), RECVBUF(*)
28
         INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
29
30
     MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
^{31}
                   ROOT, COMM, IERROR)
32
         <type> SENDBUF(*), RECVBUF(*)
33
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
34
     MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
35
                   RECVTYPE, ROOT, COMM, IERROR)
36
         <type> SENDBUF(*), RECVBUF(*)
37
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
38
                   COMM, IERROR
39
40
     MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
41
                   RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
42
         <type> SENDBUF(*), RECVBUF(*)
43
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
44
                    COMM, INFO, REQUEST, IERROR
45
     MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
46
                   ROOT, COMM, INFO, REQUEST, IERROR)
47
         <type> SENDBUF(*), RECVBUF(*)
48
```

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, 1 REQUEST, IERROR 2 MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) 5 <type> SENDBUF(*), RECVBUF(*) 6 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 7 8 MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, REQUEST, IERROR) 9 10<type> SENDBUF(*), RECVBUF(*) 11 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR 1213 MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, 14IERROR) 15<type> SENDBUF(*), RECVBUF(*) 16INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 1718 MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 19 COMM, REQUEST, IERROR) 20<type> SENDBUF(*), RECVBUF(*) 21INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 22 MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, 23RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) 24<type> SENDBUF(*), RECVBUF(*) 25INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 26RECVTYPE, COMM, REQUEST, IERROR 2728 MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 29RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) 30 <type> SENDBUF(*), RECVBUF(*) 31INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), 32 RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR 33 MPI_IBARRIER(COMM, REQUEST, IERROR) 34 INTEGER COMM, REQUEST, IERROR 35 36 MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR) 37 <type> BUFFER(*) 38 INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR 39 MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 40<type> SENDBUF(*), RECVBUF(*) 41 INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 4243 MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 44ROOT, COMM, REQUEST, IERROR) 45<type> SENDBUF(*), RECVBUF(*) 46INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, 47IERROR 48

```
1
    MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
\mathbf{2}
                   RECVTYPE, ROOT, COMM, REQUEST, IERROR)
3
         <type> SENDBUF(*), RECVBUF(*)
4
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
5
                   COMM, REQUEST, IERROR
6
     MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
7
                   IERROR)
8
         <type> SENDBUF(*), RECVBUF(*)
9
         INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR
10
11
     MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
12
                   REQUEST, IERROR)
13
         <type> SENDBUF(*), RECVBUF(*)
14
         INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
15
     MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
16
                   REQUEST, IERROR)
17
         <type> SENDBUF(*), RECVBUF(*)
18
         INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
19
20
     MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
21
         <type> SENDBUF(*), RECVBUF(*)
22
         INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
23
     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
24
                   ROOT, COMM, REQUEST, IERROR)
25
         <type> SENDBUF(*), RECVBUF(*)
26
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
27
                    IERROR
28
29
    MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
30
                   RECVTYPE, ROOT, COMM, REQUEST, IERROR)
31
         <type> SENDBUF(*), RECVBUF(*)
32
         INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
33
                    COMM, REQUEST, IERROR
34
     MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
35
         LOGICAL COMMUTE
36
         INTEGER OP, IERROR
37
38
     MPI_OP_CREATE( USER_FN, COMMUTE, OP, IERROR)
39
         EXTERNAL USER_FN
40
         LOGICAL COMMUTE
41
         INTEGER OP, IERROR
42
    MPI_OP_FREE(OP, IERROR)
43
         INTEGER OP, IERROR
44
45
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
46
         <type> SENDBUF(*), RECVBUF(*)
47
         INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
48
```

MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO, 1 REQUEST, IERROR) 2 <type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR 4 5MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) 6 <type> INBUF(*), INOUTBUF(*) 7 INTEGER COUNT, DATATYPE, OP, IERROR 8 9 MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 10IERROR) 11 <type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR 1213 MPI_REDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 14 IERROR) 15<type> SENDBUF(*), RECVBUF(*) 16INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR 1718 MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, 19 COMM, INFO, REQUEST, IERROR) 20<type> SENDBUF(*), RECVBUF(*) 21INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 22 MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 23 INFO, REQUEST, IERROR) 24<type> SENDBUF(*), RECVBUF(*) 25INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR 2627MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 28 <type> SENDBUF(*), RECVBUF(*) 29 INTEGER COUNT, DATATYPE, OP, COMM, IERROR 30 MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, 31IERROR) 32 <type> SENDBUF(*), RECVBUF(*) 33 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 34 35MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 36 ROOT, COMM, IERROR) 37 <type> SENDBUF(*), RECVBUF(*) 38 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR 39 MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 40RECVTYPE, ROOT, COMM, IERROR) 41 <type> SENDBUF(*), RECVBUF(*) 42INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 43 COMM, IERROR 44 45MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, 46RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) 47<type> SENDBUF(*), RECVBUF(*) 48

```
1
         INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
\mathbf{2}
                    COMM, INFO, REQUEST, IERROR
3
     MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
4
                   RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
5
         <type> SENDBUF(*), RECVBUF(*)
6
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
7
                    REQUEST, IERROR
8
9
10
     A.4.4 Groups, Contexts, Communicators, and Caching Fortran Bindings
11
    MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
12
         INTEGER COMM1, COMM2, RESULT, IERROR
13
14
     MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
15
         INTEGER COMM, GROUP, NEWCOMM, IERROR
16
     MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)
17
         INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
18
19
     MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
20
                   EXTRA_STATE, IERROR)
21
         EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
22
         INTEGER COMM_KEYVAL, IERROR
23
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
24
     MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
25
26
         INTEGER COMM, COMM_KEYVAL, IERROR
27
    MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
28
         INTEGER COMM, NEWCOMM, IERROR
29
30
     MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
31
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
     MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
37
         INTEGER COMM, INFO, NEWCOMM, IERROR
38
39
     MPI_COMM_FREE(COMM, IERROR)
40
         INTEGER COMM, IERROR
41
    MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
42
         INTEGER COMM_KEYVAL, IERROR
43
^{44}
     MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
45
         INTEGER COMM, COMM_KEYVAL, IERROR
46
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
47
         LOGICAL FLAG
48
```

MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR) INTEGER COMM, INFO_USED, IERROR	1 2
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR) INTEGER COMM, RESULTLEN, IERROR CHARACTER*(*) COMM_NAME	3 4 5 6
MPI_COMM_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR	7 8
MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR) INTEGER COMM, NEWCOMM, REQUEST, IERROR	9 10 11
MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR) INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR	12 13
<pre>MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>	14 15 16 17 18 19 20
MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	21 22 23 24 25
MPI_COMM_RANK(COMM, RANK, IERROR) INTEGER COMM, RANK, IERROR	25 26 27
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR	28 29 30
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR	31 32
MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	33 34 35 36
MPI_COMM_SET_INFO(COMM, INFO, IERROR) INTEGER COMM, INFO, IERROR	30 37 38
MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR) INTEGER COMM, IERROR CHARACTER*(*) COMM_NAME	39 40 41 42
MPI_COMM_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR	43 44
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR) INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR	45 46 47
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)	47

```
1
         INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
\mathbf{2}
     MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
3
         INTEGER COMM, IERROR
4
         LOGICAL FLAG
5
6
     MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
7
         INTEGER GROUP1, GROUP2, RESULT, IERROR
8
     MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
9
         INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
10
11
    MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
12
         INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
13
     MPI_GROUP_FREE(GROUP, IERROR)
14
         INTEGER GROUP, IERROR
15
16
     MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
17
         INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
18
    MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
19
         INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
20
21
    MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
22
         INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
23
     MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
^{24}
         INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
25
26
     MPI_GROUP_RANK(GROUP, RANK, IERROR)
27
         INTEGER GROUP, RANK, IERROR
28
    MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
29
         INTEGER GROUP, SIZE, IERROR
30
31
     MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
32
         INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
33
34
     MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
         INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
35
36
     MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
37
                   TAG, NEWINTERCOMM, IERROR)
38
         INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
39
         NEWINTERCOMM, IERROR
40
41
     MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
42
         INTEGER INTERCOMM, NEWINTRACOMM, IERROR
43
         LOGICAL HIGH
44
     MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
45
                   EXTRA STATE, IERROR)
46
         EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
47
         INTEGER TYPE_KEYVAL, IERROR
48
```

A.4. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	801
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	1
MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)	2
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	3
	5
MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	6
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	7
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	8
ATTRIBUTE_VAL_OUT	9
LOGICAL FLAG	10
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)	11
INTEGER TYPE_KEYVAL, IERROR	12 13
	14
MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) INTEGER DATATYPE, TYPE_KEYVAL, IERROR	15
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	16
LOGICAL FLAG	17
	18
MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR) INTEGER DATATYPE, RESULTLEN, IERROR	19
CHARACTER*(*) TYPE_NAME	20 21
	22
MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_I ATTRIBUTE_VAL_OUT, FLAG, IERROR)	N, 23
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	24
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	25
ATTRIBUTE_VAL_OUT	26
LOGICAL FLAG	27
MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STAT	²⁸ E, ₂₉
IERROR)	30
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	31
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	32
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)	33
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	34
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	35 36
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)	30
INTEGER DATATYPE, IERROR	38
CHARACTER*(*) TYPE_NAME	39
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,	40
EXTRA_STATE, IERROR)	41
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN	42 43
INTEGER WIN_KEYVAL, IERROR	43 44
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	45
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)	46
INTEGER WIN, WIN_KEYVAL, IERROR	47
	19

```
1
    MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
\mathbf{2}
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
3
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
4
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
5
                    ATTRIBUTE_VAL_OUT
6
         LOGICAL FLAG
7
     MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
8
         INTEGER WIN_KEYVAL, IERROR
9
10
     MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
11
         INTEGER WIN, WIN_KEYVAL, IERROR
12
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
13
         LOGICAL FLAG
14
     MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
15
         INTEGER WIN, RESULTLEN, IERROR
16
         CHARACTER*(*) WIN_NAME
17
18
     MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
19
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
20
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
21
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
22
                    ATTRIBUTE_VAL_OUT
23
         LOGICAL FLAG
^{24}
     MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
25
         INTEGER WIN, WIN_KEYVAL, IERROR
26
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
27
28
     MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
29
         INTEGER WIN, WIN_KEYVAL, IERROR
30
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
^{31}
    MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
32
         INTEGER WIN, IERROR
33
         CHARACTER*(*) WIN_NAME
34
35
36
     A.4.5 Process Topologies Fortran Bindings
37
38
    MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
         INTEGER COMM, NDIMS, IERROR
39
40
    MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
41
         INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
42
     MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
43
44
         INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
45
         LOGICAL PERIODS(*), REORDER
46
    MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
47
         INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
48
```

Unofficial Draft for Comment Only

LOGICAL PERIODS(*)

FORICKE LETTOPD(*)	
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR) INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR	2 3
LOGICAL PERIODS(*)	4 5
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)	6
INTEGER COMM, COORDS(*), RANK, IERROR	7 8
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR) INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR	9 10
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)	11
INTEGER COMM, NEWCOMM, IERROR	12 13
LOGICAL REMAIN_DIMS(*)	14
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR) INTEGER NNODES, NDIMS, DIMS(*), IERROR	15
	16 17
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS, INFO, REORDER, COMM_DIST_GRAPH, IERROR)	18
INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),	19
WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR	20 21
LOGICAL REORDER	22
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS, OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,	23
COMM_DIST_GRAPH, IERROR)	24 25
<pre>INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,</pre>	26
DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR LOGICAL REORDER	27
	28 29
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)	30
INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,	31
DESTINATIONS(*), DESTWEIGHTS(*), IERROR	32 33
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)	34
INTEGER COMM, INDEGREE, OUTDEGREE, IERROR LOGICAL WEIGHTED	35
	36 37
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR) INTEGER COMM, NNODES, NEDGES, IERROR	38
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,	39 40
IERROR)	40
INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR	42
LOGICAL REORDER	43 44
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)	44 45
INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR	46
<pre>MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR) INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR</pre>	47 48
INTEGEN CUTHT, MNODED, INDER(τ), EDGED(τ), NEWRANN, IERRUR	-10

1 MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR) $\mathbf{2}$ INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR 3 MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR) 4 INTEGER COMM, RANK, NNEIGHBORS, IERROR 56 MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 7 RECVTYPE, COMM, REQUEST, IERROR) 8 <type> SENDBUF(*), RECVBUF(*) 9 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 10 MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 11 DISPLS, RECVTYPE, COMM, REQUEST, IERROR) 12<type> SENDBUF(*), RECVBUF(*) 13 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 14 REQUEST, IERROR 1516MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 17RECVTYPE, COMM, REQUEST, IERROR) 18 <type> SENDBUF(*), RECVBUF(*) 19 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 20MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 21RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) 22 <type> SENDBUF(*), RECVBUF(*) 23INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 24RECVTYPE, COMM, REQUEST, IERROR 2526MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 27RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) 28 <type> SENDBUF(*), RECVBUF(*) 29INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 30 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 31REQUEST, IERROR 32 MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 33 RECVTYPE, COMM, IERROR) 34 <type> SENDBUF(*), RECVBUF(*) 35INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 36 37 MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 38 DISPLS, RECVTYPE, COMM, IERROR) 39 <type> SENDBUF(*), RECVBUF(*) 40INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 41 IERROR 42MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 43 RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 44 <type> SENDBUF(*), RECVBUF(*) 45INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 46INFO, REQUEST, IERROR 4748

MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 1 2 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 4 5 IERROR 6 MPI NEIGHBOR ALLTOALL (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 7 RECVTYPE, COMM, IERROR) <type> SENDBUF(*), RECVBUF(*) 9 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 10 11 MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR) 1213 <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 1415RECVTYPE, COMM, IERROR 16 MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, 17RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, 18 IERROR) 19 <type> SENDBUF(*), RECVBUF(*) 20INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), 21RECVTYPE, COMM, INFO, REQUEST, IERROR 22 23MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 24RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR) 25<type> SENDBUF(*), RECVBUF(*) 26INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 27INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM, 28 IERROR 29 MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, 30 RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, 31IERROR) 32 <type> SENDBUF(*), RECVBUF(*) 33 INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*) 34 INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM. 35INFO, REQUEST, IERROR 36 37 MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 38 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) 39 <type> SENDBUF(*), RECVBUF(*) 40INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 41 IERROR 42MPI_TOPO_TEST(COMM, STATUS, IERROR) 43 INTEGER COMM, STATUS, IERROR 444546A.4.6 MPI Environmental Management Fortran Bindings 47DOUBLE PRECISION MPI_WTICK() 48

```
1
     DOUBLE PRECISION MPI_WTIME()
\mathbf{2}
     MPI_ABORT(COMM, ERRORCODE, IERROR)
3
         INTEGER COMM, ERRORCODE, IERROR
4
\mathbf{5}
     MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
6
         INTEGER ERRORCLASS, IERROR
7
     MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
8
         INTEGER ERRORCLASS, ERRORCODE, IERROR
9
10
     MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
11
         INTEGER ERRORCODE, IERROR
12
         CHARACTER*(*) STRING
13
     MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
14
         INTEGER(KIND=MPI_ADDRESS_KIND SIZE
15
         INTEGER INFO, IERROR
16
         INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR
17
18
     MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
19
         INTEGER COMM, ERRORCODE, IERROR
20
     MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
21
         EXTERNAL COMM_ERRHANDLER_FN
22
         INTEGER ERRHANDLER, IERROR
23
24
     MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
25
         INTEGER COMM, ERRHANDLER, IERROR
26
     MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
27
         INTEGER COMM, ERRHANDLER, IERROR
28
29
     MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
30
         INTEGER ERRHANDLER, IERROR
^{31}
     MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
32
         INTEGER ERRORCODE, ERRORCLASS, IERROR
33
34
     MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
35
         INTEGER ERRORCODE, RESULTLEN, IERROR
36
         CHARACTER*(*) STRING
37
38
     MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
39
         INTEGER FH, ERRORCODE, IERROR
40
     MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
41
         EXTERNAL FILE_ERRHANDLER_FN
42
         INTEGER ERRHANDLER, IERROR
43
44
     MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
45
         INTEGER FILE, ERRHANDLER, IERROR
46
     MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
47
         INTEGER FILE, ERRHANDLER, IERROR
48
```

MPI_FINALIZE(IERROR) INTEGER IERROR	1 2
MPI_FINALIZED(FLAG, IERROR) LOGICAL FLAG INTEGER IERROR	3 4 5 6
MPI_FREE_MEM(BASE, IERROR) <type> BASE(*) INTEGER IERROR</type>	7 8 9
MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR) CHARACTER*(*) VERSION INTEGER RESULTLEN, IERROR	10 11 12 13
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR) CHARACTER*(*) NAME INTEGER RESULTLEN, IERROR	14 15 16 17
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR) INTEGER VERSION, SUBVERSION, IERROR	18 19 20
MPI_INIT(IERROR) INTEGER IERROR	20 21 22
MPI_INITIALIZED(FLAG, IERROR) LOGICAL FLAG INTEGER IERROR	23 24 25
MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR) INTEGER WIN, ERRORCODE, IERROR	26 27 28
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR) EXTERNAL WIN_ERRHANDLER_FN INTEGER ERRHANDLER, IERROR	29 30 31
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR) INTEGER WIN, ERRHANDLER, IERROR	32 33 34
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR) INTEGER WIN, ERRHANDLER, IERROR	35 36 37
A.4.7 The Info Object Fortran Bindings	38 39
MPI_INFO_CREATE(INFO, IERROR) INTEGER INFO, IERROR	40 41
MPI_INFO_DELETE(INFO, KEY, IERROR) INTEGER INFO, IERROR CHARACTER*(*) KEY	42 43 44 45
MPI_INFO_DUP(INFO, NEWINFO, IERROR) INTEGER INFO, NEWINFO, IERROR	46 47 48

```
1
     MPI_INFO_FREE(INFO, IERROR)
\mathbf{2}
         INTEGER INFO, IERROR
3
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
4
         INTEGER INFO, VALUELEN, IERROR
5
         CHARACTER*(*) KEY, VALUE
6
         LOGICAL FLAG
7
8
     MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
9
         INTEGER INFO, NKEYS, IERROR
10
     MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
11
         INTEGER INFO, N, IERROR
12
         CHARACTER*(*) KEY
13
14
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
15
         INTEGER INFO, VALUELEN, IERROR
16
         CHARACTER*(*) KEY
17
         LOGICAL FLAG
18
     MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
19
         INTEGER INFO, IERROR
20
         CHARACTER*(*) KEY, VALUE
21
22
23
     A.4.8 Process Creation and Management Fortran Bindings
^{24}
     MPI_CLOSE_PORT(PORT_NAME, IERROR)
25
26
         CHARACTER*(*) PORT_NAME
         INTEGER IERROR
27
28
     MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
29
         CHARACTER*(*) PORT_NAME
30
         INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
^{31}
     MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
32
33
         CHARACTER*(*) PORT_NAME
34
         INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
35
     MPI_COMM_DISCONNECT(COMM, IERROR)
36
         INTEGER COMM, IERROR
37
38
     MPI_COMM_GET_PARENT(PARENT, IERROR)
39
         INTEGER PARENT, IERROR
40
     MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
41
         INTEGER FD, INTERCOMM, IERROR
42
43
     MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
44
         INTEGER FD, INTERCOMM, IERROR
45
     MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
46
                   ARRAY_OF_ERRCODES, IERROR)
47
         CHARACTER*(*) COMMAND, ARGV(*)
48
```

INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR	1 2
<pre>MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV, ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR) INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(*)</pre>	3 4 5 6 7 8 9
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) CHARACTER*(*) SERVICE_NAME, PORT_NAME INTEGER INFO, IERROR	10 11 12 13
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR) INTEGER INFO, IERROR CHARACTER*(*) PORT_NAME	13 14 15 16
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) CHARACTER*(*) SERVICE_NAME, PORT_NAME INTEGER INFO, IERROR	17 18 19
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR) CHARACTER*(*) SERVICE_NAME, PORT_NAME INTEGER INFO, IERROR	20 21 22 23 24
A 4.0 One Sided Communications Fouture Dividings	25
A.4.9 One-Sided Communications Fortran Bindings	
<pre>A.4.9 One-Sided Communications Fortran Bindings MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>	26 27 28 29 30 31 32
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR) <type> ORIGIN_ADDR(*) INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE,TARGET_RANK, TARGET_COUNT,</type>	26 27 28 29 30 31
<pre>MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>	26 27 28 29 30 31 32 33 34 35 36 37

1 2	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR
3 4 5 6 7 8 9 10	<pre>MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,</pre>
11 12 13 14 15 16	<pre>MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>
17 18 19 20 21 22 23 24	<pre>MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>
25 26 27 28 29 30 31 23	<pre>MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,</pre>
32 33 34 35 36	MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR)
37 38 39 40 41	<type> ORIGIN_ADDR(*), RESULT_ADDR(*) INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR</type>

1 TARGET_DATATYPE, WIN, REQUEST, 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 If the Fortran compiler provides TYPE(C_PTR), then overloaded by: 6 INTERFACE MPI_WIN_ALLOCATE 7 SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & 8 WIN, IERROR) 9 IMPORT :: MPI_ADDRESS_KIND 10 INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 11 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR 12END SUBROUTINE 13 SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, & 14 WIN, IERROR) 15USE. INTRINSIC :: ISO C BINDING. ONLY : C PTR 16 IMPORT :: MPI_ADDRESS_KIND 17INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 18 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE 19 TYPE(C_PTR) :: BASEPTR 20END SUBROUTINE 21END INTERFACE 22 23MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 24INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR 25INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR 26If the Fortran compiler provides TYPE(C_PTR), then overloaded by: 27INTERFACE MPI_WIN_ALLOCATE_SHARED 28 SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, & 29BASEPTR, WIN, IERROR) 30 IMPORT :: MPI_ADDRESS_KIND 31INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 32 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR 33 34 END SUBROUTINE SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, & 35 BASEPTR, WIN, IERROR) 36 USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR 37 IMPORT :: MPI_ADDRESS_KIND 38 INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR 39 INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE 40TYPE(C_PTR) :: BASEPTR 41 42END SUBROUTINE END INTERFACE 43 44MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR) 45INTEGER WIN, IERROR 46<type> BASE(*) 47INTEGER (KIND=MPI_ADDRESS_KIND) SIZE 48

```
1
     MPI_WIN_COMPLETE(WIN, IERROR)
\mathbf{2}
         INTEGER WIN, IERROR
3
     MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
4
         <type> BASE(*)
5
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
6
         INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
\overline{7}
8
     MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
9
         INTEGER INFO, COMM, WIN, IERROR
10
     MPI_WIN_DETACH(WIN, BASE, IERROR)
11
         INTEGER WIN, IERROR
12
         <type> BASE(*)
13
14
     MPI_WIN_FENCE(ASSERT, WIN, IERROR)
15
         INTEGER ASSERT, WIN, IERROR
16
     MPI_WIN_FLUSH(RANK, WIN, IERROR)
17
         INTEGER RANK, WIN, IERROR
18
19
     MPI_WIN_FLUSH_ALL(WIN, IERROR)
20
         INTEGER WIN, IERROR
21
     MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
22
         INTEGER RANK, WIN, IERROR
23
^{24}
     MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
25
         INTEGER WIN, IERROR
26
     MPI_WIN_FREE(WIN, IERROR)
27
         INTEGER WIN, IERROR
28
^{29}
     MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
30
         INTEGER WIN, GROUP, IERROR
^{31}
32
     MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
         INTEGER WIN, INFO_USED, IERROR
33
34
     MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
35
         INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
36
37
     MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
         INTEGER ASSERT, WIN, IERROR
38
39
     MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
40
         INTEGER GROUP, ASSERT, WIN, IERROR
41
42
     MPI_WIN_SET_INFO(WIN, INFO, IERROR)
43
         INTEGER WIN, INFO, IERROR
44
     MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
45
         INTEGER WIN, RANK, DISP_UNIT, IERROR
46
         INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
47
     If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
48
```

```
1
  INTERFACE MPI_WIN_SHARED_QUERY
                                                                                    2
    SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
                                                                                    3
          BASEPTR, IERROR)
      IMPORT :: MPI_ADDRESS_KIND
                                                                                    4
      INTEGER :: WIN, RANK, DISP_UNIT, IERROR
                                                                                    5
                                                                                    6
      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
                                                                                    7
    END SUBROUTINE
    SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
                                                                                    8
          BASEPTR, IERROR)
                                                                                    9
                                                                                    10
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                    11
      IMPORT :: MPI_ADDRESS_KIND
      INTEGER :: WIN, RANK, DISP_UNIT, IERROR
                                                                                    12
      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
                                                                                    13
                                                                                    14
      TYPE(C_PTR) :: BASEPTR
                                                                                    15
    END SUBROUTINE
                                                                                    16
  END INTERFACE
                                                                                    17
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
                                                                                    18
    INTEGER GROUP, ASSERT, WIN, IERROR
                                                                                    19
                                                                                    20
MPI_WIN_SYNC(WIN, IERROR)
                                                                                    21
    INTEGER WIN, IERROR
                                                                                    22
MPI_WIN_TEST(WIN, FLAG, IERROR)
                                                                                    23
    INTEGER WIN, IERROR
                                                                                    24
    LOGICAL FLAG
                                                                                    25
                                                                                    26
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
                                                                                    27
    INTEGER RANK, WIN, IERROR
                                                                                    28
MPI_WIN_UNLOCK_ALL(WIN, IERROR)
                                                                                    29
    INTEGER WIN, IERROR
                                                                                    30
                                                                                    31
MPI_WIN_WAIT(WIN, IERROR)
                                                                                    32
    INTEGER WIN, IERROR
                                                                                    33
                                                                                    34
A.4.10 External Interfaces Fortran Bindings
                                                                                    35
                                                                                    36
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
                                                                                    37
    INTEGER REQUEST, IERROR
                                                                                    38
MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
                                                                                    39
              IERROR)
                                                                                    40
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
                                                                                    41
                                                                                    42
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER REQUEST, IERROR
                                                                                    43
                                                                                    44
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
                                                                                    45
    INTEGER REQUIRED, PROVIDED, IERROR
                                                                                    46
                                                                                    47
MPI_IS_THREAD_MAIN(FLAG, IERROR)
                                                                                    48
    LOGICAL FLAG
```

1	INTEGER IERROR
2 3	MPI_QUERY_THREAD(PROVIDED, IERROR) INTEGER PROVIDED, IERROR
4 5 6 7	MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG
8 9 10	MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
11 12 13 14	MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) COUNT
15 16	A.4.11 I/O Fortran Bindings
17 18 19 20 21 22 23	<pre>MPI_CONVERSION_FN_NULL(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION, EXTRA_STATE, IERROR) <type> USERBUF(*), FILEBUF(*) INTEGER COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) POSITION INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE</type></pre>
24 25	MPI_FILE_CLOSE(FH, IERROR) INTEGER FH, IERROR
26 27 28 29	MPI_FILE_DELETE(FILENAME, INFO, IERROR) CHARACTER*(*) FILENAME INTEGER INFO, IERROR
30 31	MPI_FILE_GET_AMODE(FH, AMODE, IERROR) INTEGER FH, AMODE, IERROR
32 33 34 35	MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG
36 37 38	MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
39 40 41	MPI_FILE_GET_GROUP(FH, GROUP, IERROR) INTEGER FH, GROUP, IERROR
42 43 44	MPI_FILE_GET_INFO(FH, INFO_USED, IERROR) INTEGER FH, INFO_USED, IERROR
45 46 47 48	MPI_FILE_GET_POSITION(FH, OFFSET, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	1 2 3
MPI_FILE_GET_SIZE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE	4 5 6 7
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR) INTEGER FH, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT	8 9 10 11
<pre>MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR) INTEGER FH, ETYPE, FILETYPE, IERROR CHARACTER*(*) DATAREP INTEGER(KIND=MPI_OFFSET_KIND) DISP</pre>	12 13 14 15
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR</type>	16 17 18 19
MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR</type>	20 21 22
<pre>MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)</pre>	23 24 25 26
<pre>MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)</pre>	27 28 29 30 31
MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR</type>	32 33 34 35
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR</type>	36 37 38
MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR</type>	39 40 41 42
<pre>MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)</pre>	43 44 45 46
MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)	47 48

1<type> BUF(*) $\mathbf{2}$ INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 3 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 4 MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) 5<type> BUF(*) 6 INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR 7 8 MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) 9 CHARACTER*(*) FILENAME 10 INTEGER COMM, AMODE, INFO, FH, IERROR 11 MPI_FILE_PREALLOCATE(FH, SIZE, IERROR) 12INTEGER FH, IERROR 13 INTEGER(KIND=MPI_OFFSET_KIND) SIZE 1415MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 16<type> BUF(*) 17INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 18 MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 19 <type> BUF(*) 20INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 2122MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) 23<type> BUF(*) 24 INTEGER FH, COUNT, DATATYPE, IERROR 25MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR) 26<type> BUF(*) 27INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR 2829MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 30 <type> BUF(*) 31 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 32 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 33 MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 34 <type> BUF(*) 35 INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR 36 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 37 38MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) 39 <type> BUF(*) 40INTEGER FH, COUNT, DATATYPE, IERROR 41 INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 42MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR) 43 <type> BUF(*) 44INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR 4546MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 47<type> BUF(*) 48

INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	1
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	2
<pre><type> BUF(*)</type></pre>	3 4
INTEGER FH, COUNT, DATATYPE, IERROR	5
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)	6
<pre><pre><pre><pre><pre><pre><pre><pre></pre></pre></pre></pre></pre></pre></pre></pre>	7
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR	8
	9
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	10
<pre><type> BUF(*) INTEGED EN COUNT DATATIONE OTATIO(NDI OTATIO GIZE) IEDDOD</type></pre>	11
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	12
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)	13
INTEGER FH, WHENCE, IERROR	14
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	15 16
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)	10
INTEGER FH, WHENCE, IERROR	18
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	19
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)	20
INTEGER FH, IERROR	21
LOGICAL FLAG	22
	23
MPI_FILE_SET_INFO(FH, INFO, IERROR)	24
INTEGER FH, INFO, IERROR	25
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)	26
INTEGER FH, IERROR	27 28
INTEGER(KIND=MPI_OFFSET_KIND) SIZE	29
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)	30
INTEGER FH, ETYPE, FILETYPE, INFO, IERROR	31
CHARACTER*(*) DATAREP	32
INTEGER(KIND=MPI_OFFSET_KIND) DISP	33
MPI_FILE_SYNC(FH, IERROR)	34
INTEGER FH, IERROR	35
	36
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	37
<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR</type>	38 39
INIEGER III, COONT, DRIVITE, DIRIOD(III I_DIRIOD_DIZE), IERCOR	40
MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)	41
<type> BUF(*)</type>	42
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR	43
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)	44
<type> BUF(*)</type>	45
INTEGER FH, COUNT, DATATYPE, IERROR	46
MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)	47
	48

Unofficial Draft for Comment Only

1 2	<type> BUF(*) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR</type>
3 4 5 6 7	<pre>MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>
8 9 10 11	<pre>MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>
12 13 14 15 16	MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET</type>
17 18 19 20	MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR) <type> BUF(*) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR</type>
21 22 23	<pre>MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>
24 25 26 27	<pre>MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) <type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR</type></pre>
28 29 30	<pre>MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR) <type> BUF(*) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR</type></pre>
31 32 33 34	<pre>MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)</pre>
35 36 37 38 39 40 41	<pre>MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR) CHARACTER*(*) DATAREP EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE INTEGER IERROR</pre>
42 43 44 45 46 47 48	<pre>A.4.12 Language Bindings Fortran Bindings MPI_F_SYNC_REG(BUF)</pre>

INTEGER SIZE, IERROR	1
MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)	2 3
TYPE(MPI_Status) :: F08_STATUS	4
INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR	5
MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)	6
INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR	7
TYPE(MPI_Status) :: F08_STATUS	8 9
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)	9 10
INTEGER P, R, NEWTYPE, IERROR	11
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)	12
INTEGER R, NEWTYPE, IERROR	13
	14
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR) INTEGER P, R, NEWTYPE, IERROR	15
	16 17
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)	18
INTEGER TYPECLASS, SIZE, DATATYPE, IERROR	19
	20
A.4.13 Tools / Profiling Interface Fortran Bindings	21
MPI_PCONTROL(LEVEL)	22
INTEGER LEVEL	23
	24
	25 26
A.4.14 Deprecated Fortran Bindings	20
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)	28
INTEGER COMM, KEYVAL, IERROR	29
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	30
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR	31
LOGICAL FLAG	32
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)	33
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR	34 35
	36
MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)	37
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	38
ATTRIBUTE_VAL_OUT, IERR	39
LOGICAL FLAG	40
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)	41
EXTERNAL COPY_FN, DELETE_FN	42 43
INTEGER KEYVAL, IERROR	43
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	45
MPI_KEYVAL_FREE(KEYVAL, IERROR)	46
INTEGER KEYVAL, IERROR	47
,,,,	48

1	MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
3	INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
4	ATTRIBUTE_VAL_OUT, IERR
5	LOGICAL FLAG
6	
7 8	MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR
9	SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
10	ATTRIBUTE_VAL_OUT, FLAG, IERR)
11	INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
12	ATTRIBUTE_VAL_OUT, IERR
13	LOGICAL FLAG
14	SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
15 16	INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
17	
18	
19	
20 21	
21	
23	
24	
25	
26	
27	
28	
29	
30	
31 32	
33	
34	
35	
36	
37	
38	
39	
40	
41 42	
42 43	
43	
45	
46	
47	
48	

Annex B

Change-Log

Annex B.1 summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown. If not otherwise noted, the section and page references refer to the locations of the change or new functionality in this version of the standard. Changes in Annexes B.2–B.5 were already introduced in the corresponding sections in previous versions of this standard.

B.1	Changes from Version 3.1 to Version 3.2	23 24		
	5	25		
B.1.1	Changes in MPI-3.2	26		
1	Section $3.8.4$ on page 76.	27		
1.	Cancelling a send request by calling MPI_CANCEL has been deprecated and may	28		
	removed in a future version of the MPI specification.	29		
		30		
2.	Sections 3.7.3, 3.9, 5.13, 7.8, and 7.9 on pages 55, 77, 223, 363, and 369.	31		
	Persistent collective communication and persistent neighborhood communication are	32		
	added to the standard.			
3.	Section $6.4.2$ on page 266 , and MPI-3.1 Section $6.4.2$ on page 237 .			
	The functions MPI_COMM_DUP and MPI_COMM_IDUP were updated to no longer			
	propagate info hints. This change may affect backward compatibility.			
			4	Sections 6.4.4, 11.9.7, and 12.9.9 an areas 970, 469, and 551, and
4.	Sections 6.4.4, 11.2.7, and 13.2.8 on pages 279, 462, and 551, and MDI 2.1 Sections 6.4.4, 11.2.7, and 12.2.8 on pages 248, 415, and 500			
	MPI-3.1 Sections 6.4.4, 11.2.7, and 13.2.8 on pages 248, 415, and 500. The definition of info hints was updated to allow applications to provide assertions			
	regarding their usage of MPI objects and operations.	42		
	regarding their usage of this robjects and operations.	43		
5.	Section $6.4.4$ on page 279.	44		
	The new info hints mpi_assert_no_any_tag, mpi_assert_no_any_source,			
	mpi_assert_exact_length, and mpi_assert_allow_overtaking were added for use with commu-	46		
	nicators.	47		
		48		

1 2	6.	Section 6.4.2 on page 266. The MPI_COMM_IDUP_WITH_INFO function was added.
3 4 5 6 7	7.	Sections 6.4.4, 11.2.7, and 13.2.8 on pages 279, 462, and 551. The semantics of the MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, MPI_WIN_GET_INFO, MPI_FILE_SET_INFO, and MPI_FILE_GET_INFO were clarified.
8 9 10 11	8.	Section 7.5. MPI_DIMS_CREATE is now guaranteed to return MPI_SUCCESS if the number of di- mensions passed to the routine is set to 0 and the number of nodes is set to 1.
12 13 14 15 16	9.	Sections 2.8, 8.3, 8.5, and 8.7 on pages 20, 382, 394, and 399. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_SELF instead of MPI_COMM_WORLD. The definition of MPI_ERRORS_ARE_FATAL was clarified to cover all connected processes, and a new error handler, MPI_ERRORS_ABORT, was created to limit the scope of aborting.
17 18 19 20 21	10.	Section 12.3 on page ?? The mpi_f08 binding incorrectly had the dummy parameter flag in the MPI F08 binding for MPI_SET_STATUS_CANCELLED marked as INTENT(OUT). It has been fixed to be INTENT(IN).
22 23 24 25	11.	Sections 8.3 and 8.4 on pages 382 and 391. Clarified definition of errors to say that MPI should continue whenever possible and allow the user to recover from errors.
26 27	B.2	Changes from Version 3.0 to Version 3.1
28 29	B.2.	1 Fixes to Errata in Previous Versions of MPI
30 31 32 33 34	1.	Chapters 3–17, Annex A.3 on page 732, and Example 5.21 on page 195, and MPI-3.0 Chapters 3-17, Annex A.3 on page 707, and Example 5.21 on page 187. Within the mpi_f08 Fortran support method, BIND(C) was removed from all SUBROUTINE, FUNCTION, and ABSTRACT INTERFACE definitions.
35 36 37	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.
38 39 40 41	3.	Section 3.8.2 on page 71, and MPI-3.0 Section 3.8.2 on page 67. The flag arguments of the Fortran interfaces of MPI_IMPROBE were originally incor- rectly defined as INTEGER (instead as LOGICAL).
42	4	Section 6.4.9 on men 2000, and MDL 2.0 Section 6.4.9 on men 2027
43 44 45	1.	Section 6.4.2 on page 266, 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.

6.	Section 7.6 on page 348, 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).	1 2 3 4 5
7.	Section 8.1.1 on page 375, 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.	6 7 8 9
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), ?? and ?? (Profiling interface), and corresponding sections in MPI-3.0. The linker name concept was substituted by defining specific procedure names.	10 11 12 13 14 15 16
9.	Section 11.2.1 on page 449, 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.	17 18 19 20
10.	Section 11.3.4 on page 470, 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.	21 22 23 24
11.	Section 11.3.4 on page 470, 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.	24 25 26 27
12.	Section ?? on page ??, 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.	28 29 30 31
13.	Section ?? on page ??, and MPI-3.0 Section 14.3.3 on page 563. New paragraph clarifying variable name equivalence in the tools information interface.	31 32 33
14.	Sections ??, ??, and ?? on pages ??, ??, and ??, 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.	34 35 36 37 38 39 40
15.	Section ?? on page ?? , and MPI-3.0 Section 14.3.7 on page 573. Clarify return code of MPI_T_PVAR_{START,STOP,RESET} routines.	41 42
16.	Section ?? on page ??, 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.	43 44 45
17.	Section 17.1.4 on page 627, 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.	46 47 48

Unofficial Draft for Comment Only

1 2 3	18.	Section 17.1.5 on page 628, 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.
4 5 6	19.	Section 17.1.6 on page 633, and MPI-3.0 Section 17.1.6 on page 611. The requirements on BIND(C) procedure interfaces were removed.
7 8 9 10 11	20.	Annexes A.2, A.3, and A.4 on pages 708, 732, and 787, 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.
12 13 14 15 16	21.	Annex A.3.4 on page 753, 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.
	3.2.2	Changes in MPI-3.1
18 19 20 21	1.	Sections 2.6.4 and 4.1.5 on pages 20 and 106. 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.
22 23 24 25 26 27 28	2.	Sections 8.1.1, 8.7, and 12.4 on pages 375, 399, and 532. 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.
29 30	3.	Section 11.2.1 on page 449. The same_disp_unit info key was added for use in RMA window creation routines.
31 32 33 34	4.	Sections 13.4.2 and 13.4.3 on pages 560 and 565. Added MPI_FILE_IREAD_AT_ALL, MPI_FILE_IWRITE_AT_ALL, MPI_FILE_IREAD_ALL, and MPI_FILE_IWRITE_ALL
35 36 37 38	5.	Sections ??, ??, and ?? on pages ??, ??, and ??. Clarified that NULL parameters can be provided in MPI_T_{CVAR PVAR CATEGORY}_GET_INFO routines.
39 40 41 42 43 44	6.	Sections ??, ??, and ?? on pages ??, ??, and ??. 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.
45 46 47 48		

B.3.	CHANGES FROM VERSION 2.2 TO VERSION 3.0	825
B.3	Changes from Version 2.2 to Version 3.0	1
B.3.1	Fixes to Errata in Previous Versions of MPI	2 3
1.	Sections 2.6.2 and 2.6.3 on pages 19 and 19, and MPI-2.2 Section 2.6.2 on page lines 41-42, Section 2.6.3 on page 18, lines 15-16, and Section 2.6.4 on page 18, lines 41. This is an MPI-2 erratum: The scope for the reserved prefix MPI_ and the C namespace MPI is now any name as originally intended in MPI-1.	5 40- 6
	Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, and Annex A.1.1 on pages 27, 184, 592, 685, and MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, Annex A.1.1 on pages 27, 164, 433, 472 and 513 This is an MPI-2.2 erratum: New named predefined datatypes MPI_CXX_BOOL, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran correspond to the C++ types bool, std::complex <float>, std::complex<double>, and std::complex<long double="">. These datatypes also correspond to the depreca C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEA, which were removed in MPI-3.0. The r standard C++ types Complex<> were substituted by the standard types std::complex<></long></double></float>	and 11 12 , 13 14 ding 15 16 ated 17 .EX, 18
3.	Sections 5.9.2 on pages 184 and MPI-2.2 Section 5.9.2, page 165, line 47. This is an MPI-2.2 erratum: MPI_C_COMPLEX was added to the "Complex" rection group.	25
4.	Section 7.5.5 on page 335, and MPI-2.2, Section 7.5.5 on page 257, C++ interface page 264, line 3. This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree now defined as int& indegree and int& outdegree in the C++ interface of MPI_DIST_GRAPH_NEIGHBORS_COUNT.	27
5.	Section 13.5.2, Table 13.2 on page 592, and MPI-2.2, Section 13.5.3, Table 13.2 page 433. This was an MPI-2.2 erratum: The MPI_C_BOOL "external32" representation is rected to a 1-byte size.	2 on ³² ₃₃
6.	MPI-2.2 Section 16.1.16 on page 471, line 45. This is an MPI-2.2 erratum: The constant MPI::_LONG_LONG should be MPI::LONG_LONG.	36 37 38 39
7.	Annex A.1.1 on page 685, Table "Optional datatypes (Fortran)," and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37-41. This is an MPI-2.2 erratum: The C++ datatype handles MPI::INTEGER16, MPI::REAL16, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16, MPI::F_COMPLEX32 were added to the table.	40 41 42 43 44 45 46 47 48

1	B.3.2	2 Changes in MPI-3.0
2 3 4 5 6	1.	Section 2.6.1 on page 17, Section 15.2 on page 618 and all other chapters. The C++ bindings were removed from the standard. See errata in Section B.3.1 on page 825 for the latest changes to the MPI C++ binding defined in MPI-2.2. This change may affect backward compatibility.
7 8 9 10 11 12 13 14 15 16	2.	Section 2.6.1 on page 17, Section 14.1 on page 613 and Section 15.1 on page 617. The deprecated functions MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT, MPI_TYPE_LB, MPI_TYPE_UB, MPI_ERRHANDLER_CREATE (and its callback function prototype MPI_Handler_function), MPI_ERRHANDLER_SET, MPI_ERRHANDLER_GET, the dep- recated special datatype handles MPI_LB, MPI_UB, and the constants MPI_COMBINER_HINDEXED_INTEGER, MPI_COMBINER_HVECTOR_INTEGER, MPI_COMBINER_STRUCT_INTEGER were removed from the standard. This change may affect backward compatibility.
17 18 19	3.	Section 2.3 on page 10. Clarified parameter usage for IN parameters. C bindings are now const-correct where backward compatibility is preserved.
20 21 22 23 24	4.	Section 2.5.4 on page 15 and Section 7.5.4 on page 328. The recommended C implementation value for MPI_UNWEIGHTED changed from NULL to non-NULL. An additional weight array constant (MPI_WEIGHTS_EMPTY) was introduced.
25 26 27 28	5.	Section 2.5.4 on page 15 and Section 8.1.1 on page 375. Added the new routine MPI_GET_LIBRARY_VERSION to query library specific versions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.
29 30 31 32 33 34 35 36 37 38	6.	Sections 2.5.8, 3.2.2, 3.3, 5.9.2, on pages 17, 27, 29, 184, Sections 4.1, 4.1.7, 4.1.8, 4.1.11, 12.3 on pages 87, 112, 114, 117, 531, and Annex A.1.1 on page 685. New inquiry functions, MPI_TYPE_SIZE_X, MPI_TYPE_GET_EXTENT_X, MPI_TYPE_GET_TRUE_EXTENT_X, and MPI_GET_ELEMENTS_X, return their results as an MPI_Count value, which is a new type large enough to represent element counts in memory, file views, etc. A new function, MPI_STATUS_SET_ELEMENTS_X, modifies the opaque part of an MPI_Status object so that a call to MPI_GET_ELEMENTS_X returns the provided MPI_Count value (in Fortran, INTEGER (KIND=MPI_COUNT_KIND)). The corresponding predefined datatype is MPI_COUNT.
39 40 41 42	7.	Chapter 3 on page 25 until Chapter 17 on page 621. In the C language bindings, the array-arguments' interfaces were modified to consistently use use [] instead of *.
43 44 45		Exceptions are MPI_INIT, which continues to use char ***argv (correct because of subtle rules regarding the use of the & operator with char *argv []), and MPI_INIT_THREAD, which is changed to be consistent with MPI_INIT.
46 47 48	8.	Sections 3.2.5, 4.1.5, 4.1.11, 4.2 on pages 32, 106, 117, 138. 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.

- Section 3.2.6 on page 34, and Section 3.8 on page 68.
 MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE, and MPI_MPROBE.
- Section 3.8 on page 68 and Section 3.11 on page 86. The use of MPI_PROC_NULL in probe operations was clarified. A special predefined message MPI_MESSAGE_NO_PROC was defined for the use of matching probe (i.e., the new MPI_MPROBE and MPI_IMPROBE) with MPI_PROC_NULL.
- 11. Sections 3.8.2, 3.8.3, 17.2.4, A.1.1 on pages 71, 73, 670, 685.

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.

- Section 4.1.2 on page 89 and Section 4.1.13 on page 122. The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant MPI_COMBINER_HINDEXED_BLOCK were added.
- Chapter 5 on page 149 and Section 5.12 on page 205.
 Added nonblocking interfaces to all collective operations.
- Sections 6.4.2, 6.4.4, 11.2.7, on pages 266, 279, 462. The new routines MPI_COMM_DUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were added. The routine MPI_COMM_DUP must also duplicate info hints.
- 15. Section 6.4.2 on page 266. Added MPI_COMM_IDUP.
- 16. Section 6.4.2 on page 266. Added the new communicator construction routine MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.
- Section 6.4.2 on page 266.
 Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type constant MPI_COMM_TYPE_SHARED.
- 18. Section 6.6.2 on page 292.

In MPI-2.2, communication involved in an MPI_INTERCOMM_CREATE operation 46 could interfere with point-to-point communication on the parent communicator with 47 the same tag or MPI_ANY_TAG. This interference has been removed in MPI-3.0. 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

1	19.	Section 6.8 on page 313 .
2		Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type
3		and window names.
4		
5	20.	Section $7.5.8$ on page 346 .
		MPI_CART_MAP can also be used for a zero-dimensional topologies.
6		
7	21.	Section 7.6 on page 348 and Section 7.7 on page 357.
8		The following neighborhood collective communication routines were added to sup-
9		
10		port sparse communication on virtual topology grids: MPI_NEIGHBOR_ALLGATHER,
11		MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL,
		${\sf MPI_NEIGHBOR_ALLTOALLV}, \; {\sf MPI_NEIGHBOR_ALLTOALLW} \; {\rm and} \; {\rm the} \; {\rm nonblocking}$
12		variants MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV,
13		MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and
14		MPI_INEIGHBOR_ALLTOALLW. The displacement arguments in
15		MPI_NEIGHBOR_ALLTOALLW and MPI_INEIGHBOR_ALLTOALLW were defined as
16		
17		address size integers. In $MPI_DIST_GRAPH_NEIGHBORS,$ an ordering rule was added
		for communicators created with MPI_DIST_GRAPH_CREATE_ADJACENT.
18		
19	22.	Section 8.7 on page 399 and Section 12.4.3 on page 535.
20		The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE was clarified. After
21		MPI is initialized, the application can access information about the execution envi-
22		ronment by querying the new predefined info object MPI_INFO_ENV.
23		· · · · · · · · · · · · · · · · · · ·
24	23.	Section 8.7 on page 399.
		Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.
25		
26	24.	Chapter 11 on page 447.
27		Substantial revision of the entire One-sided chapter, with new routines for window
28		creation, additional synchronization methods in passive target communication, new
29		
30		one-sided communication routines, a new memory model, and other changes.
31	25	Section ?? on page ??.
	20.	A new MPI Tool Information Interface was added.
32		A new MP1 1001 information interface was added.
33		The following changes are related to the Fortran language support.
34		
35	26.	Section 2.3 on page 10, and Sections 17.1.1, 17.1.2, 17.1.7 on pages 621, 622, and 637.
36		The new mpi_08 Fortran module was introduced.
37		•
	27.	Section 2.5.1 on page 12, and Sections 17.1.2, 17.1.3, 17.1.7 on pages 622, 625, and 637.
38		Handles to opaque objects were defined as named types within the mpi_08 Fortran
39		module. The operators .EQ., .NE., ==, and /= were overloaded to allow the compari-
40		son of these handles. The handle types and the overloaded operators are also available
41		
42		through the mpi Fortran module.
43	28	Sections 2.5.4, 2.5.5 on pages 15, 16, Sections 17.1.1, 17.1.10, 17.1.11, 17.1.12, 17.1.13
44	<i>2</i> 0.	
		on pages 621, 648, 649, 650, 653, and Sections 17.1.2, 17.1.3, 17.1.7 on pages 622, 625,
45		637.
46		Within the mpi_08 Fortran module, choice buffers were defined as assumed-type and
47		assumed-rank according to Fortran 2008 TS 29113 [41], and the compile-time constant
48		MPI_SUBARRAYS_SUPPORTED was set to .TRUE With this, Fortran subscript triplets

can be used in nonblocking MPI operations; vector subscripts are not supported in nonblocking operations. If the compiler does not support this Fortran TR 29113 feature, the constant is set to .FALSE..

- 29. Section 2.6.2 on page 19, Section 17.1.2 on page 622, and Section 17.1.7 on page 637. The ierror dummy arguments are OPTIONAL within the mpi_08 Fortran module.
- 30. Section 3.2.5 on page 32, Sections 17.1.2, 17.1.3, 17.1.7, on pages 622, 625, 637, and Section 17.2.5 on page 672.
 Within the mpi_08 Fortran module, the status was defined as TYPE(MPI_Status). Additionally, within both the mpi and the mpi_f08 modules, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and TYPE(MPI_Status) are defined. New conversion routines were added: MPI_STATUS_F2F08, MPI_STATUS_F082F, MPI_Status_c2f08, and MPI_Status_f082c, In mpi.h, the new type MPI_F08_status, and the external variables MPI_F08_STATUS_IGNORE and MPI_F08_STATUSES_IGNORE were added.
- 31. Section 3.6 on page 47. In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument of MPI_BUFFER_DETACH is incorrectly defined and the argument is therefore unused.
- 32. Section 4.1 on page 87, Section 4.1.6 on page 110, and Section 17.1.15 on page 653. The Fortran alignments of basic datatypes within Fortran derived types are implementation dependent; therefore it is recommended to use the BIND(C) attribute for derived types in MPI communication buffers. If an array of structures (in C/C++) or derived types (in Fortran) is to be used in MPI communication buffers, it is recommended that the user creates a portable datatype handle and additionally applies MPI_TYPE_CREATE_RESIZED to this datatype handle.
- 33. Sections 4.1.10, 5.9.5, 5.9.7, 6.7.4, 6.8, 8.3.1, 8.3.2, 8.3.3, 14.1, 17.1.9 on pages 117, 191, 197, 308, 313, 384, 386, 388, 613, and 639. In some routines, the dummy argument names were changed because they were identical to the Fortran keywords TYPE and FUNCTION. The new dummy argument names must be used because the mpi and mpi_08 modules guarantee keyword-based actual argument lists. The argument name type was changed in MPI_TYPE_DUP, the Fortran USER_FUNCTION of MPI_OP_CREATE, MPI_TYPE_SET_ATTR, MPI_TYPE_GET_ATTR, MPI_TYPE_DELETE_ATTR, MPI_TYPE_GET_NAME, MPI_TYPE_DELETE_ATTR, MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, the callback prototype definition MPI_Type_delete_attr_function, and the predefined callback function MPI_TYPE_NULL_DELETE_FN; function was changed in MPI_OP_CREATE, MPI_FILE_CREATE_ERRHANDLER, MPI_ERRHANDLER, CREATE_ERRHANDLER, and MPI_ERRHANDLER_CREATE. For consistency reasons, INOUBUF was changed to INOUTBUF in MPI_REDUCE_LOCAL, and intracomm to newintracomm in MPI_INTERCOMM_MERGE.
- 34. Section 6.7.2 on page 299.
 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.

 $\mathbf{2}$

 24

1 2 3 4	35.	Section 8.2 on page 379. With the mpi and mpi_f08 Fortran modules, MPI_ALLOC_MEM now also supports TYPE(C_PTR) C-pointers instead of only returning an address-sized integer that may be usable together with a non-standard Cray-pointer.
5 6 7 8	36.	Section 17.1.15 on page 653, and Section 17.1.7 on page 637. Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers in MPI operations.
9 10 11 12 13 14 15 16 17	37.	Section 17.1.16 on page 655 to Section 17.1.19 on page 666, Section 17.1.7 on page 637, and Section 17.1.8 on page 638. 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 17.1.7.
18 19 20	38.	Section 17.1.2 on page 622. Within the mpi_08 Fortran module, dummy arguments are now declared with INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces.
21 22 23	39.	Section 17.1.3 on page 625, and Section 17.1.7 on page 637. The existing mpi Fortran module must implement compile-time argument checking.
24 25 26	40.	Section 17.1.4 on page 627. The use of the mpif.h Fortran include file is now strongly discouraged.
27 28 29 30 31	41.	Section A.1.1, Table " <i>Predefined functions</i> " on page 693, Section A.1.3 on page 700, and Section A.3.4 on page 753. Within the new mpi_f08 module, all callback prototype definitions are now defined with explicit interfaces PROCEDURE(MPI) that have the BIND(C) attribute; user-written callbacks must be modified if the mpi_f08 module is used.
32 33 34 35	42.	Section A.1.3 on page 700. In some routines, the Fortran callback prototype names were changed from \dots _FN to \dots _FUNCTION to be consistent with the other language bindings.
36 37 38	B.4	Changes from Version 2.1 to Version 2.2
39 40 41 42	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.
43 44 45 46	2.	Section 2.6 on page 17, and Section 15.2 on page 618. The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.
47 48	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.

		~
4.	Section 3.2.2 on page 27.	5 6
	MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL,	7
	$MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, \mathrm{and}$	8
	$MPI_C_LONG_DOUBLE_COMPLEX$ are now valid predefined MPI data types.	9
5	Section 3.4 on page 39, Section 3.7.2 on page 51, Section 3.9 on page 77, and Section 5.1	10
0.	on page 149.	11
	The read access restriction on the send buffer for blocking, non blocking and collective	12
	API has been lifted. It is permitted to access for read the send buffer while the	13
	operation is in progress.	14
C		15
6.	Section 3.7 on page 50.	16
	The Advice to users for IBSEND and IRSEND was slightly changed.	17 18
7.	Section 3.7.3 on page 55.	19
	The advice to free an active request was removed in the Advice to users for	20
	MPI_REQUEST_FREE.	21
0		22
8.	Section 3.7.6 on page 67.	23
	MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.	24
9.	Section 5.8 on page 176 .	25
	"In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and	26
	MPI_ALLTOALLW for intracommunicators.	27
10	Section 5.0.2 on page 184	28
10.	Section 5.9.2 on page 184. Predefined parameterized datatypes (e.g., returned by	29
	MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g.	30
	MPI_REAL8) have been added to the list of valid datatypes in reduction operations.	31
	with <u>Encloy</u> have been added to the list of valid datatypes in reduction operations.	32
11.	Section $5.9.2$ on page 184.	33
	$MPI_(U)INT\{8,16,32,64\}_T$ are all considered C integer types for the purposes of the	34
	predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran	35 36
	integer types. MPI_C_BOOL is considered a Logical type.	37
	MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and	38
	MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types.	39
12.	Section 5.9.7 on page 197.	40
	The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been	41
	added.	42
19	Section 5 10.1 on page 100	43
19.	Section 5.10.1 on page 199. The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan-	44
	dard.	45
	uuru.	46
14.	Section $5.11.2$ on page 203.	47
	Added in place argument to MPI_EXSCAN.	48

Unofficial Draft for Comment Only

1 2 3 4 5 6 7	15.	Section 6.4.2 on page 266, and Section 6.6 on page 288. Implementations that did not implement MPI_COMM_CREATE on intercommuni- cators will need to add that functionality. As the standard described the behav- ior of this operation on intercommunicators, it is believed that most implementa- tions already provide this functionality. Note also that the C++ binding for both MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms.
8 9 10 11	16.	Section 6.4.2 on page 266. MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm is an intracommunicator. If comm is an intercommunicator it was clarified that all processes in the same local group of comm must specify the same value for group.
12 13 14 15 16 17	17.	Section 7.5.4 on page 328. New functions for a scalable distributed graph topology interface has been added. In this section, the functions MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE, the constants MPI_UNWEIGHTED, and the derived C++ class Distgraphcomm were added.
18 19 20 21	18.	Section 7.5.5 on page 335. For the scalable distributed graph topology interface, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS and the constant MPI_DIST_GRAPH were added.
22 23 24 25	19.	Section 7.5.5 on page 335. Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT.
26 27 28	20.	Section 8.1.1 on page 375. The subversion number changed from 1 to 2.
29 30 31	21.	Section 8.3 on page 382, Section 14.2 on page 616, and Annex A.1.3 on page 700. Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names.
32 33 34 35 36	22.	Section 8.7.1 on page 405. Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Implementors must now also register all implementation-internal attribute deletion callbacks on MPI_COMM_SELF before returning from MPI_INIT/MPI_INIT_THREAD.
 37 38 39 40 41 42 43 44 	23.	Section 11.3.4 on page 470. The restriction added in MPI 2.1 that the operation MPI_REPLACE in MPI_ACCUMULATE can be used only with predefined datatypes has been removed. MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0. Also, a clarification has been made that MPI_REPLACE can be used only in MPI_ACCUMULATE, not in collective operations that do reductions, such as MPI_REDUCE and others.
45 46 47 48	24.	Section 12.2 on page 523. Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for MPI::Grequest::Start() for consistency with the rest of MPI functions that take function pointer arguments.

MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes 4 in the external 32 representation. 56 26. Section 17.2.7 on page 677. 7 The description was modified that it only describes how an MPI implementation be-8 haves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example 9 16.17 was replaced with three new examples 17.13, 17.14, and 17.15 on pages 678-67910 explicitly detailing cross-language attribute behavior. Implementations that matched 11 the behavior of the old example will need to be updated. 1213 27. Annex A.1.1 on page 685. 14Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 698). 1528. Annex A.1.1 on page 685. Table Named Predefined Datatypes. 16Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, 1718 MPI_C_FLOAT_COMPLEX, MPI_C_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes. 19 2021**B.5** Changes from Version 2.0 to Version 2.1 22 231. Section 3.2.2 on page 27, and Annex A.1 on page 685. 24 In addition, the MPI_LONG_LONG should be added as an optional type; it is a syn-25onym for MPI_LONG_LONG_INT. 262. Section 3.2.2 on page 27, and Annex A.1 on page 685. 27MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), 28 MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved 29from optional to official and they are therefore defined for all three language bindings. 30 31 3. Section 3.2.5 on page 32. 32 MPI_GET_COUNT with zero-length datatypes: The value returned as the 33 count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes 34 have been transferred is zero. If the number of bytes transferred is greater than zero, 35 MPI_UNDEFINED is returned. 36 37 4. Section 4.1 on page 87. 38 General rule about derived datatypes: Most datatype constructors have replication 39 count or block length arguments. Allowed values are non-negative integers. If the 40 value is zero, no elements are generated in the type map and there is no effect on 41 datatype bounds or extent. 425. Section 4.3 on page 144. 43 MPI_BYTE should be used to send and receive data that is packed using 44MPI_PACK_EXTERNAL. 4546 6. Section 5.9.6 on page 195. 47If comm is an intercommunicator in MPI_ALLREDUCE, then both groups should pro-48

Unofficial Draft for Comment Only

1

 $\mathbf{2}$

1 2		vide count and datatype arguments that specify the same type signature (i.e., it is not necessary that both groups provide the same count value).
3 4 5 6 7	7.	Section 6.3.1 on page 256. MPI_GROUP_TRANSLATE_RANKS and MPI_PROC_NULL: MPI_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL as the translated rank.
8 9 10	8.	Section 6.7 on page 297. About the attribute caching functions:
11 12 13 14 15 16 17		Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (<i>End of advice to implementors.</i>)
18 19 20 21 22 23	9.	Section 6.8 on page 313. In MPI_COMM_GET_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For- tran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_OBJECT_NAME.
25 26	10.	Section 7.4 on page 322. About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must have identical values on all processes of the group of comm_old.
27 28 29 30 31	11.	Section 7.5.1 on page 324. In MPI_CART_CREATE: If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.
32 33 34 35	12.	Section 7.5.3 on page 326. In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$, then MPI_COMM_NULL is returned in all processes.
	13.	Section 7.5.3 on page 326. In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be non-symmetric.
41 42 43 44		Advice to users. Performance implications of using multiple edges or a non- symmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (<i>End of advice to users.</i>)
45 46 47 48	14.	Section 7.5.5 on page 335. In MPI_CARTDIM_GET and MPI_CART_GET: If comm is associated with a zero- dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and MPI_CART_GET will keep all output arguments unchanged.

15.	Section 7.5.5 on page 335. In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol- ogy, coord is not significant and 0 is returned in rank.	1 2 3
16.	Section 7.5.5 on page 335. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.	4 5 6 7
17.	Section 7.5.6 on page 343. In MPI_CART_SHIFT: It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.	8 9 10 11 12 13
18.	Section 7.5.7 on page 345. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology.	14 15 16 17
18.1.	Section 8.1.1 on page 375. The subversion number changed from 0 to 1.	18 19 20
19.	Section 8.1.2 on page 377. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.	21 22 23 24 25
20.	Section 8.3 on page 382. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.	26 27 28 29 30 31 32 33
21.	Section 8.7 on page 399, 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 441.	34 35 36 37 38
22.	Section 8.7 on page 399. About MPI_ABORT:	39 40 41
	Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (<i>End of advice to users.</i>)	42 43 44 45
	Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)	46 47 48

1 2 3 4 5 6 7 8 9	23.	Section 9 on page 409. 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.
10 11 12 13	24.	Section 11.3 on page 463. MPI_PROC_NULL is a valid target rank in the MPI RMA calls MPI_ACCUMULATE, MPI_GET, and MPI_PUT. The effect is the same as for MPI_PROC_NULL in MPI point- to-point communication. See also item 25 in this list.
14 15 16 17 18	25.	Section 11.3 on page 463. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch. See also item 24 in this list.
19 20 21	26.	Section 11.3.4 on page 470. MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.
22 23 24 25 26 27	27.	Section 13.2.8 on page 551. 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 29 30 31	28.	Section 13.2.8 on page 551. About MPI_FILE_GET_INFO: If no hint exists for the file associated with fh, a handle to a newly created info object is returned that contains no key/value pair.
32 33 34	29.	Section 13.3 on page 554. If a file does not have the mode MPI_MODE_SEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.
35 36 37	30.	Section 13.5.2 on page 591. The bias of 16 byte doubles was defined with 10383. The correct value is 16383.
38 39 40	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.
41 42	32.	Section 17.1.9 on page 639. About MPI_TYPE_CREATE_F90_XXX:
43 44 45 46 47 48		Advice to implementors. An application may often repeat a call to $MPI_TYPE_CREATE_F90_XXX$ with the same combination of (XXX,p,r) . The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, the MPI implementation should return the same datatype handle for the same (

33.	Section $A.1.1$ on	page <u>685</u> .			
	MPI_BOTTOM is	defined as voi	1 *	const	MPI::BOTTOM.

 $\mathbf{5}$

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 communication performance of large multicomputers. In <i>Proceedings of the 1991 International 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 23	[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
24 25 26	[18]	J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993. 1.2
27 28 29	[19]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Concepts, June 1991. 1.2
30 31	[20]	Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992. 1.2
32 33 34 35	[21]	D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991. 6.1.2
36 37 38 39	[22]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard. The In- ternational Journal of Supercomputer Applications and High Performance Computing, 8, 1994. 1.3
40 41	[23]	Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org. 1.3
42 43 44 45	[24]	Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing. 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 instrumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990. 1.2

- [26] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009. 3.8.2
- [27] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. 1.2
- [28] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Available via world wide web from http://www.uni-karlsruhe.de/~Michael.Hennecke/Publications/#MPI_F90. 17.1.3
- [29] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. 3.8.1, 3.8.2
- [30] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 5.12
- [31] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum mechanical calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 15th European PVM/MPI Users' Group Meeting, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. 7.6
- [32] T. Hoefler and A. Lumsdaine. Message progression in parallel computing to thread or not to thread? In Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society, Oct. 2008. 5.12
- [33] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07.* IEEE Computer Society/ACM, Nov. 2007. 5.12
- [34] T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimization for medical image reconstruction algorithms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 5.12
- [35] T. Hoefler and J. L. Traeff. Sparse collective operations for MPI. In Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop, May 2009. 7.6
- [36] Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI common practice, issues, and extensions. In Cotronis et al. [14], pages 345–355. 6.4.2
- [37] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. 13.5.2
- [38] International Organization for Standardization, Geneva, ISO 8859-1:1987. Information 46 processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet 47 No. 1, 1987. 13.5.2

1

 $\mathbf{2}$

3

4

5 6

 $\overline{7}$

8

9

10

11

12

13

14

15 16

17

18

19

20

21

22

23 24

25

26

27 28

29

30

 31

32

33

34

35

36 37

38

39

40

41

42 43

44

1 2 3	[39]	International Organization for Standardization, Geneva, ISO/IEC 9945-1:1996(E). Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language], December 1996. 12.4, 13.2.1
4 5 6 7	[40]	International Organization for Standardization, Geneva, ISO/IEC 1539-1:2010. Information technology – Programming languages – Fortran – Part 1: Base language, November 2010. 17.1.1, 17.1.2
8 9 10 11 12 13	[41]	International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TS 29113. TS on further interoperability with C, 2012. http://www.nag.co.uk/sc22wg5/, successfully balloted DTS at ftp://ftp.nag.co.uk/sc22wg5/N1901-N1950/N1917.pdf. 17.1.1, 17.1.1, 17.1.2, 17.1.7, 28
14 15	[42]	Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. <i>The High Performance Fortran Handbook</i> . MIT Press, 1993. 4.1.4
16 17 18 19	[43]	David Kotz. Disk-directed I/O for MIMD multiprocessors. In <i>Proceedings of the 1994 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	[51]	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
44 45 46 47 48	[52]	A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, <i>Proceedings of the Fifth Distributed Memory Concurrent Computing Conference</i> , pages 767–776. IEEE Press, 1990. 1.2, 6.1.2

- [53] 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
- [54] 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
- [55] 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
- [56] 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
- [57] 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
- [58] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. 13.5.2
- [59] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2

 $\mathbf{2}$

 $\overline{7}$

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, 106, 657 access epoch, 48413 action 14in function names, 10 15active, 55, 319 16active target communication, 484 17addresses, 121 18 absolute, 16, 106, 657 19correct use, **121** relative displacement, 16, 106 20all-reduce, 195 21nonblocking, 219 22 persistent, 237 23all-to-all, 176 24nonblocking, 215 25persistent, 233 26array arguments, 14 27assertions, 498 ASYNCHRONOUS 28Fortran attribute, 660 29 attribute, <u>253</u>, <u>297</u>, **677** 30 caching, 252 31 32 backward incompatibilies, 619 33 barrier synchronization, 155 34nonblocking, 207 persistent, 225 35 blocking, <u>11</u>, <u>39</u>, 42, 557 36 I/O, 558 37 bounds of datatypes, 112 38 broadcast, 156 39 nonblocking, 208 40 persistent, 225 41 buffer allocation, 47 buffered, <u>40</u>, 50, <u>51</u> 42nonblocking, 5043 buffered send, 4344 45С 46language binding, 19 47caching, 251, 252, 297 callback functions 48

language interoperability, 676 prototype definitions, 700 deprecated, 705 cancel, 18, 56, <u>68</u>, **76**, 403, 616, 821 canonical pack and unpack, 144 Cartesian topology, 322, 324 change-log, 821 choice, 16 class in function names, 10 clock synchronization, 378 collective, 11, 557 collective communication, 149 correctness, 241 file data access operations, 576 neighborhood, 348 nonblocking, 205 commit. 115 COMMON blocks, 663 communication, 447 collective, 149 modes, 39 one-sided, 447 point-to-point, 25 RMA, 447 communicator, 29, 251, 252 completes operation, 11 completion, 55 multiple, 60 connected, 441 constants, 15, 681, 685 context, 251, 252, 254 conversion, 38 counts. 17 create in function names, 10

data, 27 data conversion, 38 datatypes, 87, 675 delete in function names, $\underline{10}$ deprecated interfaces, 17, 613 derived datatype, <u>12</u>, **87**, **653** disconnected, 442 displacement, $\underline{541}$, 555distributed graph topology, <u>322</u>, **328** dynamically attached memory, 456 elementary datatype, <u>541</u>, 555 empty, 55 end of file, 543envelope, 25, 29 environmental inquiries, 377 equivalent datatypes, 12 error handling, 20, 382 error codes and classes, 391, 394 error handlers, 384, 394, 677 I/O, 607, 608 one-sided communication, 500 program error, 20resource error, 21 establishing communication, 429 etype, <u>541</u>, 555 exception, 382exclusive scan nonblocking, 223 persistent, 241 explicit offsets, 558, 560 exposure epoch, 484extent of datatypes, $\underline{88}$, $\underline{111}$, $\mathbf{112}$ true extent, **114** external32 file data representation, 589extra-state, 681 fairness, 44, 66 file, <u>541</u> data access, 557 collective operations, 576 explicit offsets, **560** individual file pointers, 565 seek, 578 shared file pointers, 573 split collective, **580** end of file, 543filetype, 542 handle, 543interoperability, 587 manipulation, 543

> offset, **16**, <u>542</u> pointer, <u>543</u> size, <u>543</u>

view, 541, <u>542</u> , 554	1
file size, 603	2
finished, 405	3
Fortran	4
language binding, 19 , 621	5
Fortran support, 621	6
gather, 157	7
nonblocking, 209	8
persistent, 226	9
gather-to-all, 173	10
nonblocking, 213	11
persistent, 231	12
general datatype, $\underline{87}$	13
generalized requests, 523 , 523	14
get	
in function names, $\underline{10}$	15
graph	16
topology, <u>322</u> , 326	17
group, 251 , <u>252</u> , <u>254</u> , 290	18
group objects, $\underline{254}$	19
handles, <u>12</u> , 670	20
host rank, 377	21
	22
immediate, 51	23
inactive, 55	24
inclusive scan, 202	
nonblocking, 222	25
persistent, 240	26
independent, $\underline{442}$	27
individual file pointers, 558 , 565	28
info object, 409	29
file info, 551	30
keys, 706	31
values, 706	32
initiation, 51	33
inter-communication, $\underline{253}$, $\underline{289}$	
inter-communicator, $\underline{253}$, $\underline{289}$	34
collective operations, 153, 154	35
interlanguage communication, 682	36
internal	31
file data representation, 588	38
interoperability, 587	39
intra-communication, $\underline{253}$, $\underline{288}$	40
intra-communicator, $\underline{252}$, $\underline{288}$	41
collective operations, 152	42
intra-communicator objects, $\underline{255}$	
I/O, 541 IO mult 277	43
IO rank, 377	44
is in function names 10	45
in function names, $\underline{10}$	46
language binding, 17 , 621	47
interoperability, 669	48
· · · · · · · · · · · · · · · · · · ·	

General Index

1 summary, 685 $\mathbf{2}$ lb_marker, 100, 104, <u>110</u>, 110, 115 3 erased, <u>113</u> local, 11, 40 4 local group, 267 $\mathbf{5}$ loosely synchronous model, 318 6 lower bound, 1107 lower-bound markers, 110 8 macros, 20 9 main thread, 53410 matched receives, 73 11 matching 12type, **35**, 118, **602** 13 matching probe, 71 14memory 15allocation, 379 16system, 12 memory model, 448, 483 17separate, 448, 455 18 unified, 448, 455 19message, 2520data, 27 21envelope. 29 22 modes, 39 23module variables, 663 mpi module 24 Fortran support, 625 25mpi_f08 module 26Fortran support, 622 27mpiexec, 400, 405, 407 28mpif.h include file 29 Fortran support, 627 30 mpirun, 406 31 multiple completions, 60 32 named datatype, 12 33 names, **429** 34 name publishing, 434 35 naming objects, **313** 36 native 37 file data representation, <u>588</u> neighborhood collective communication, 348 38 nonblocking, 358 39 non-local, <u>11</u>, <u>39</u>, <u>40</u> 40nonblocking, 11, 50, 464, 557 41 communication, 5042completion, 55 43 Fortran problems, 656 44 I/O. 558 initiation, 51 45 request objects, 51 46 null handle, 55 47null processes, 86 48

offset, **16**, $\underline{542}$ one-sided communication, **447** Fortran problems, **657** opaque objects, **12**, **674** operation completes, <u>11</u> origin, <u>448</u>

pack, 138 canonical, 144 packing unit, 140 parallel procedure, 318 passive target communication, 484 persistent communication requests, 58, 77 collective persistent, 223, 363 Fortran problems, 657 point-to-point communication, 25 portable datatype, 12ports, **429** predefined datatype, 12 predefined reduction operations, 184 private window copy, 482 probe, 68 probe, matching, 71 process creation, 415 process group, 29processes, 20 processor name, 378 program error, 20 prototype definitions, 700 deprecated, 705 public window copy, 482rank, 254 ready, <u>40</u>, <u>50</u>, <u>51</u> nonblocking, 50ready send, 43receive, 25, 26, 30 buffer, 26 complete, 50context, 290start call, 50reduce, **182** nonblocking, 218 persistent, 236 reduce-scatter, 198 nonblocking, 220, 221 persistent, 238, 239 reduction operations, 181, 677 predefined, 184 process-local, 197 scan, 202 user-defined, 191 related, 140

relative displacement, 16, 106

remote group, 267 Remote Memory Access, see RMA removed interfaces, 17, 617 request complete I/O, 558 request objects, 51 resource error, 21 $RMA, \underline{447}$ communication calls, 463 request-based, 477 memory model, 482 synchronization calls, 484 scan, 202 inclusive, 202 scatter, 167 nonblocking, 211 persistent, 229 seek, 578 semantics file consistency, 597 nonblocking communications, 59 point-to-point communication, 43 semantics and correctness one-sided communication, 500 send, <u>25</u>, **26** buffer, 25complete, 50context, 290 start, 50send-receive, 83 separate memory model, 448, 455, 483 sequential storage, $\underline{121}$ Set in function names, 10shared file pointers, <u>558</u>, **573** shared memory allocation, 453 signals, 22 singleton init, 440 size changing I/O, 603 source, 290split collective, 557, 580 standard, <u>39</u>, 50 nonblocking, 50standard send, 43starting processes, 416, 418 startup, **399** portable, 406 state, 14 status, 32, 672 associating information, 530 ignore, 34 test, **67**

strong synchronization, $\underline{485}$	1
synchronization, 447, 464	2
synchronization calls	3
RMA, 484	4
synchronous, 40 , 50, 51 , 55	5
nonblocking, $\underline{50}$	6
synchronous send, 43	
system memory, $\underline{12}$	7
	8
tag values, 377	9
target, $\underline{448}$	10
thread compliant, $\underline{532}$, $\underline{536}$	11
threads, 532	12
timers and synchronization, 398	13
topologies, 321	14
topology	15
Cartesian, <u>322</u> , 32 4	
distributed graph, <u>322</u> , 328	16
graph, <u>322</u> , 326	17
virtual, 322	18
true extent of datatypes, 114	19
type map, $\underline{88}$	20
type matching, 35 , 118	21
type signature, $\underline{88}$	22
types, 698	23
ub_marker, 100, 104, 105, <u>110</u> , 110, 115	
erased, <u>113</u>	24
unified memory model, 448, 454, <u>483</u>	25
universe size, 439	26
unnamed datatype, $\underline{12}$	27
unpack, 138	28
canonical, 144	29
upper bound, <u>110</u>	30
upper-bound markers, 110	
user functions at process termination, 405	31
user-defined data representations, 593	32
user-defined reduction operations, 191	33
	34
version inquiries, 375	35
view, 541, 542, 554	36
virtual topology, 252, 253, 322	37
· · · · · · · · · · · · · · · · · · ·	38
weak synchronization, $\underline{485}$	39
window	
allocation, 451	40
creation, 449	41
dynamically attached memory, 456	42
shared memory allocation, 453	43
	44
	45
	46
	46
	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, 517, 659, 667 Attributes between languages, 678 1415C/Fortran handle conversion, 671 16Cartesian virtual topologies, 369 17Client-server code, 66 18 with blocking probe, 70 with blocking probe, wrong, 70 19 20Datatype 213D array, 129 22 absolute addresses, 134 23array of structures, 131 24 elaborate example, 142, 143 matching type, 118 25matrix transpose, 130 26union, 13527Datatypes 28 matching, 36 29 not matching, 3630 untyped, 36 31 Deadlock if not buffered, 4632 with MPI_Bcast, 242 33 wrong message exchange, 4534 35 False matching of collective operations, 245 36 Fortran 90 copying and sequence problem, 649, 37 651,652 38 Fortran 90 derived types, 653 Fortran 90 heterogeneous communication, 646, 39 647 40 Fortran 90 invalid KIND, 642 41 Fortran 90 MPI_TYPE_MATCH_SIZE 42implementation, 646 43 Fortran 90 overlapping communication and 44 computation, 664, 665, 668 45Fortran 90 register optimization, 656–658 46Independence of nonblocking operations, 248 47Intercommunicator, 271, 275 48

Interlanguage communication, 682 Intertwined matching pairs, 44 Message exchange, 45Mixing blocking and nonblocking collective operations, 244 Mixing collective and point-to-point requests, 247MPI_ACCUMULATE, 472 MPI_Accumulate, 515, 516, 519 MPI_Aint, 131 MPI_Aint_add, 519 MPI_Allgather, 175 MPI_ALLOC_MEM, 381 MPI_Alloc_mem, 382, 519 MPI_ALLREDUCE, 196 MPI_Alltoall, 246 MPI_ASYNC_PROTECTS_NONBLOCKING, 517MPI_Barrier, 403, 505–508, 514–516 MPI_Bcast, 157, 242-245 MPI_BSEND, 44 MPI_Buffer_attach, 48, 402 MPI_Buffer_detach, 48 MPI_BYTE, 36 MPI_Cancel, 403 MPI_CART_COORDS, 344 MPI_CART_GET, 369 MPI_CART_RANK, 344 MPI_CART_SHIFT, 344, 369 MPI_CART_SUB, 346 MPI_CHARACTER, 37 MPI_Comm_create, 271, 283, 284, 286 MPI_Comm_create_keyval, 311 MPI_Comm_dup, 285 MPI_Comm_get_attr, 311 MPI_Comm_group, 271, 286, 311 MPI_Comm_remote_size, 275 MPI_Comm_set_attr, 311 MPI_COMM_SPAWN, 420

MPI_Comm_spawn, 420

MPI_COMM_SPAWN_MULTIPLE, 426 MPI_Comm_spawn_multiple, 426 MPI_Comm_split, 275, 294, 296 MPI_Compare_and_swap, 516, 519 MPI_DIMS_CREATE, 325, 369 MPI_DIST_GRAPH_CREATE, 333 MPI_Dist_graph_create, 334 MPI_DIST_GRAPH_CREATE_ADJACENT, 333 MPI_F_sync_reg, 517 MPI_FILE_CLOSE, 566, 569 MPI_FILE_GET_AMODE, 550 MPI_FILE_IREAD, 569 MPI_FILE_OPEN, 566, 569 MPI_FILE_READ, 566 MPI_FILE_SET_ATOMICITY, 603 MPI_FILE_SET_VIEW, 566, 569 MPI_FILE_SYNC, 604 MPI_Finalize, 402, 403 MPI_FREE_MEM, 381 MPI_Free_mem, 519 MPI_Gather, 143, 160, 161, 165 MPI_Gatherv, 143, 162–165 MPI_GET, 468, 469 MPI_Get, 505-508, 513, 514 MPI_Get_accumulate, 515, 516, 519 MPI_GET_ADDRESS, 107, 653, 654, 675 MPI_Get_address, 131, 134, 135, 142 MPI_GET_COUNT, 120 MPI_GET_ELEMENTS, 120 MPI_GRAPH_CREATE, 327, 340 MPI_GRAPH_NEIGHBORS, 340 MPI_GRAPH_NEIGHBORS_COUNT, 340 MPI_Grequest_complete, 528 MPI_Grequest_start, 528 MPI_Group_excl, 283 MPI_Group_free, 271, 283, 284 MPI_Group_incl, 271, 284, 286 MPI_Iallreduce, 247 MPI_Ialltoall, 246 MPI_Ibarrier, 244–247 MPI_Ibcast, 209, 247, 248 MPI_INFO_ENV, 401 MPI_Intercomm_create, 294, 296 MPI_Iprobe, 403 MPI_IRECV, 57-60, 66 MPI_Irecv, 247 MPI_ISEND, 57-59, 66 MPI_Op_create, 194, 195, 204 MPI_Pack, 141-143 MPI_Pack_size, 143 MPI_PROBE, 70 MPI_Put, 490, 496, 506–508, 513, 514 MPI_RECV, 36, 37, 44-46, 60, 70, 118

MPI_Recv, 246	1
MPI_REDUCE, 185, 186, 189	2
MPI_Reduce, 189, 190, 194, 195	- 3
MPI_REQUEST_FREE, 58	
MPI_Request_free, 402	4
MPI_Rget, 518	5
	6
MPI_Rput, 518 MPI_Scan, 204	7
	8
MPI_Scatter, 170 MPI_Scatterry, 170, 171	9
MPI_Scattery, 170, 171	
MPI_SEND, 36, 37, 45, 46, 60, 70, 118	10
MPI_Send, 131, 134, 135, 142, 246, 247	11
MPI_SENDRECV, 129, 130	12
MPI_SENDRECV_REPLACE, 344	13
MPI_SSEND, 44, 60	14
MPI_Test_cancelled, 403	15
MPI_TYPE_COMMIT, 116, 129, 130, 468,	
653, 654	16
MPI_Type_commit, 131, 134, 135, 142,	17
161-165, 171, 204	18
MPI_TYPE_CONTIGUOUS, 90, 110, 118, 120	19
$MPI_Type_contiguous, 161$	20
MPI_TYPE_CREATE_DARRAY, 106	21
MPI_TYPE_CREATE_HVECTOR, 129, 130	22
MPI_Type_create_hvector, 131, 134	
MPI_TYPE_CREATE_INDEXED_BLOCK,	23
468	24
MPI_TYPE_CREATE_RESIZED, 653, 654	25
MPI_TYPE_CREATE_STRUCT, 98, 110,	26
130,653,654	27
MPI_Type_create_struct, 131, 134, 135, 142,	28
164, 165, 204	
MPI_TYPE_CREATE_SUBARRAY, 612	29
MPI_TYPE_EXTENT, 468	30
MPI_TYPE_FREE, 468	31
MPI_Type_get_contents, 136	32
MPI_Type_get_envelope, 136	33
MPI_TYPE_GET_EXTENT, 129, 130, 469,	34
472	35
MPI_Type_get_extent, 131	
MPI_TYPE_INDEXED, 93, 129	36
MPI_Type_indexed, 131, 134	37
MPI_TYPE_VECTOR, 91, 129, 130	38
MPI_Type_vector, 162, 163, 165, 171	39
MPI_Unpack, 142, 143	40
MPI_User_function, 195	41
MPI_WAIT, 57–60, 66, 569	42
MPI_Wait, 244–247	
MPI_Waitall, 247, 518	43
MPI_WAITANY, 66	44
MPI_Waitany, 518	45
MPI_WAITSOME, 66	40
MPI_Win_attach, 519	47
MPI_Win_complete, 490, 508, 514	48
_ · · · · · · · · · · · · · · · · · · ·	

```
1
      MPI_WIN_CREATE, 468, 469, 472
\mathbf{2}
      MPI_Win_create_dynamic, 519
      MPI_Win_detach, 519
3
      MPI_WIN_FENCE, 468, 469, 472
4
      MPI_Win_fence, 513
\mathbf{5}
      MPI_Win_flush, 507, 515, 516, 519
6
      MPI_Win_flush_all, 516
\overline{7}
      MPI_Win_flush_local, 506
8
      MPI_WIN_FREE, 469, 472
9
      MPI_Win_lock, 496, 505-508
      MPI_Win_lock_all, 517-519
10
      MPI_Win_post, 508, 514
11
      MPI_Win_start, 490, 508, 514
12
      MPI_Win_sync, 506, 507, 515, 516
13
           shared memory windows, 517
14
      MPI_Win_unlock, 496, 505-508
15
      MPI_Win_unlock_all, 518, 519
16
      MPI_Win_wait, 508, 514
17
      mpiexec, 401, 408
18
      Neighborhood collective communication, 369
19
      No Matching of Blocking and Nonblocking
20
               collective operations, 246
21
      Non-deterministic program with MPI_Bcast,
22
               243
23
      Non-overtaking messages, 44
      Nonblocking operations, 57, 58
24
           message ordering, 59
25
          progress, 60
26
27
      Overlapping Communicators, 247
28
      Pipelining nonblocking collective operations,
29
               247
30
      Progression of nonblocking collective
31
               operations, 246
32
33
      Shared memory windows
34
           MPI_Win_sync, 517
35
      Threads and MPI, 533
36
      Topologies, 369
37
      Typemap, 89–91, 93, 98, 106
38
39
      Virtual topologies, 369
40
41
42
43
44
45
46
47
48
```

MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

MPI::_LONG_LONG, 825 MPI::BOOL, 825 MPI::COMPLEX, 825 MPI::DOUBLE_COMPLEX, 825 MPI::F_COMPLEX16, 825 MPI::F_COMPLEX32, 825 MPI::F_COMPLEX4, 825 MPI::F_COMPLEX8, 825 MPI::INTEGER16, 825 MPI::LONG_DOUBLE_COMPLEX, 825 MPI::LONG_LONG, 825 MPI::REAL16, 825 MPI_2DOUBLE_PRECISION, 188, 691 MPI_2INT, 188, 691 MPI_2INTEGER, 188, 691 MPI_2REAL, 188, 691 MPI_ADDRESS_KIND, 15, 16, 16, 28, 298, 649, 678, 688 MPI_AINT, 28, 29, 185, 457, 689, 690, 831, 833 MPI_ANY_SOURCE, 30, 31, 43, 54, 55, 68, 69, 71-73, 81, 84, 85, 279, 319, 378, 687 MPI_ANY_TAG, 15, 30, 31, 33, 54, 55, 68, 69, 71-75, 81, 84-86, 279, 687, 827 MPI_APPNUM, 441, 694 MPI_ARGV_NULL, 16, 420, 421, 648, 696 MPI_ARGVS_NULL, 16, 425, 648, 696 MPI_ASYNC_PROTECTS_NONBLOCKING, 15, 517, 622, 623, 625, 627, 630, 637, 639, 660, 688, 830 MPI_BAND, 184, 185, 692 MPI_BOR, 184, 185, 692 MPI_BOTTOM, 10, 15, 16, 34, 106, 121, 122, 152, 330, 332, 422, 457, 461, 624, 626, 633, 648, 653, 655, 657, 658, 661-663, 666, 675, 676, 682, 687, 837 MPI_BSEND_OVERHEAD, 49, 687 MPI_BXOR, 184, 185, 692 MPI_BYTE, 27, 28, 35-38, 144, 185, 542, 588,

	12
589, 602, 682, 689, 690, 833	13
MPI_C_BOOL, 28, 185, 689, 825, 831, 833	14
MPI_C_COMPLEX, 28, 185, 689, 825, 831,	15
833	16
MPI_C_DOUBLE_COMPLEX, 28, 185, 689,	17
831, 833	18
MPI_C_FLOAT_COMPLEX, 185, 689, 831,	19
833	20
MPI_C_LONG_DOUBLE_COMPLEX, 28,	21
185, 689, 831, 833	22
MPI_CART, 335, 693	
MPI_CHAR, 28, 38, 98, 186, 187, 689, 830, 831	23
MPI_CHARACTER, 27, 37, 38, 186, 187, 690	24
MPI_COMBINER_CONTIGUOUS, 123, 126,	25
695	26
MPI_COMBINER_DARRAY, 123, 128, 695	27
MPI_COMBINER_DUP, 123, 126, 695	28
MPI_COMBINER_F90_COMPLEX, 123, 128,	29
695 MDI COMDINED FOO INTECTED 122 120	30
MPI_COMBINER_F90_INTEGER, 123, 128, 695	31
MPI_COMBINER_F90_REAL, 123, 128, 695	
MPI_COMBINER_HINDEXED, 18, 123, 127,	32
695	33
MPI_COMBINER_HINDEXED_BLOCK, 123,	34
127, 695, 827	35
MPI_COMBINER_HINDEXED_INTEGER,	36
18, 618, 826	37
MPI_COMBINER_HVECTOR, 18, 123, 127,	38
695	39
MPI_COMBINER_HVECTOR_INTEGER,	40
18,618,826	41
MPI_COMBINER_INDEXED, 123, 127, 695	42
MPI_COMBINER_INDEXED_BLOCK, 123,	43
127,695	44
MPI_COMBINER_NAMED, 123, 126, 695	
MPI_COMBINER_RESIZED, 123, 129, 695	45
MPI_COMBINER_STRUCT, 18, 123, 127, 695	46
MPI_COMBINER_STRUCT_INTEGER, 18,	47
618,826	48

> 5 6

1 MPI_COMBINER_SUBARRAY, 123, 128, 695 $\mathbf{2}$ MPI_COMBINER_VECTOR, 123, 126, 695 MPI_COMM_DUP_FN, 18, 300, 693, 829 3 MPI_COMM_NULL, 255, 270, 271, 273-275, 4 278, 279, 315, 324, 326, 423, 442, 444, 5692, 834 6 MPI_COMM_NULL_COPY_FN, 18, 300, 624, 7 676, 693, 829 8 MPI_COMM_NULL_DELETE_FN, 18, 301, 9 693 10 MPI_COMM_PARENT, 315 MPI_COMM_SELF, 21, 255, 273, 280, 297, 11 315, 382, 383, 405, 442, 544, 619, 691, 12822, 832 13MPI_COMM_TYPE_SHARED, 278, 691, 827 14MPI_COMM_WORLD, 15, 22, 30, 255-257, 15265, 267, 280, 284, 292, 315, 325, 377, 16378, 383, 385, 395, 402, 403, 405, 407, 415, 416, 418, 419, 423, 425, 439-442, 17 536, 537, 587, 607, 608, 619, 670, 681, 18 691, 822, 835 19MPI_COMPLEX, 27, 185, 591, 640, 690 20MPI_COMPLEX16, 185, 690 21MPI_COMPLEX32, 185, 690 22MPI_COMPLEX4, 185, 690 23MPI_COMPLEX8, 185, 690 24 MPI_CONGRUENT, 266, 291, 691 25MPI_CONVERSION_FN_NULL, 596, 693 MPI_COUNT, 28, 29, 185, 689, 690, 826 26MPI_COUNT_KIND, 15, 28, 688 27MPI_CXX_BOOL, 29, 185, 690, 825 28MPI_CXX_DOUBLE_COMPLEX, 29, 185, 29690, 825 30MPI_CXX_FLOAT_COMPLEX, 29, 185, 690, 31 825 MPI_CXX_LONG_DOUBLE_COMPLEX, 29, 32 185, 690, 825 33 MPI_DATATYPE_NULL, 116, 692 34MPI_DISPLACEMENT_CURRENT, 555, 35 696, 836 36 MPI_DIST_GRAPH, 335, 693, 832 37 MPI_DISTRIBUTE_BLOCK, 103, 696 38MPI_DISTRIBUTE_CYCLIC, 103, 696 39 MPI_DISTRIBUTE_DFLT_DARG, 103, 696 MPI_DISTRIBUTE_NONE, 103, 696 40MPI_DOUBLE, 28, 185, 639, 689 41 MPI_DOUBLE_COMPLEX, 27, 185, 591, 640, 42690 43MPI_DOUBLE_INT, 188, 189, 691 44 MPI_DOUBLE_PRECISION, 27, 185, 640, 45690 46MPI_DUP_FN, 18, 301, 614, 694 47MPI_ERR_ACCESS, 393, 547, 609, 686 MPI_ERR_AMODE, 393, 545, 609, 686 48

MPI_ERR_ARG, 392, 685 MPI_ERR_ASSERT, 392, 501, 686 MPI_ERR_BAD_FILE, 393, 609, 686 MPI_ERR_BASE, 381, 392, 501, 686 MPI_ERR_BUFFER, 392, 685 MPI_ERR_COMM, 392, 685 MPI_ERR_CONVERSION, 393, 596, 609, 686 MPI_ERR_COUNT, 392, 685 MPI_ERR_DIMS, 392, 685 MPI_ERR_DISP, 392, 501, 686 MPI_ERR_DUP_DATAREP, 393, 594, 609, 686 MPI_ERR_FILE, 393, 609, 686 MPI_ERR_FILE_EXISTS, 393, 609, 686 MPI_ERR_FILE_IN_USE, 393, 547, 609, 686 MPI_ERR_GROUP, 392, 685 MPI_ERR_IN_STATUS, 33, 35, 56, 63, 65, 385, 392, 527, 560, 686 MPI_ERR_INFO, 392, 686 MPI_ERR_INFO_KEY, 392, 411, 686 MPI_ERR_INFO_NOKEY, 392, 411, 686 MPI_ERR_INFO_VALUE, 392, 411, 686 MPI_ERR_INTERN, 383, 392, 685 MPI_ERR_IO, 393, 609, 686 MPI_ERR_KEYVAL, 311, 392, 686 MPI_ERR_LASTCODE, 391, 393, 395, 396, 687 MPI_ERR_LOCKTYPE, 392, 501, 686 MPI_ERR_NAME, 392, 436, 686 MPI_ERR_NO_MEM, 380, 392, 686 MPI_ERR_NO_SPACE, 393, 609, 686 MPI_ERR_NO_SUCH_FILE, 393, 547, 609, 686 MPI_ERR_NOT_SAME, 393, 609, 686 MPI_ERR_OP, 392, 500, 685 MPI_ERR_OTHER, 391, 392, 685 MPI_ERR_PENDING, 63, 392, 685 MPI_ERR_PORT, 392, 433, 686 MPI_ERR_QUOTA, 393, 609, 686 MPI_ERR_RANK, 392, 500, 685 MPI_ERR_READ_ONLY, 393, 609, 686 MPI_ERR_REQUEST, 392, 685 MPI_ERR_RMA_ATTACH, 393, 501, 686 MPI_ERR_RMA_CONFLICT, 392, 501, 686 MPI_ERR_RMA_FLAVOR, 393, 455, 501, 686 MPI_ERR_RMA_RANGE, 393, 501, 686 MPI_ERR_RMA_SHARED, 393, 501, 686 MPI_ERR_RMA_SYNC, 392, 501, 686 MPI_ERR_ROOT, 392, 685 MPI_ERR_SERVICE, 392, 436, 686 MPI_ERR_SIZE, 392, 501, 686 MPI_ERR_SPAWN, 392, 421, 422, 686 MPI_ERR_TAG, 392, 685 MPI_ERR_TOPOLOGY, 392, 685

NDI EDD TELINGATE AND ACT
MPI_ERR_TRUNCATE, 392, 685
MPI_ERR_TYPE, 392, 685
MPI_ERR_UNKNOWN, 391, 392, 685
MPI_ERR_UNSUPPORTED_DATAREP, 393,
609,686
MPI_ERR_UNSUPPORTED_OPERATION,
393, 609, 686
MPI_ERR_WIN, 392, 501, 686
MPI_ERRCODES_IGNORE, 16, 422, 648, 696
MPI_ERRHANDLER_NULL, 390, 692
MPI_ERROR, 32, 56, 206, 477, 688, 822, 829
MPI_ERRORS_ABORT, 383, 822
MPI_ERRORS_ARE_FATAL, 383, 384, 397,
500, 607, 688, 822
MPI_ERRORS_RETURN, 383, 384, 398, 405,
608, 681, 688
MPI_F08_STATUS_IGNORE, 673, 697, 829
MPI_F08_STATUSES_IGNORE, 673, 697, 829
MPI_F_STATUS_IGNORE, 672, 697
MPI_F_STATUSES_IGNORE, 672, 697
MPI_FILE_NULL, 546, 608, 692
MPI_FLOAT, 28, 98, 183, 185, 590, 689
MPI_FLOAT_INT, 12, 188, 189, 691
MPI_GRAPH, 335, 693
MPI_GROUP_EMPTY, 254, 260, 261, 270,
271, 273, 692
MPI_GROUP_NULL, 254, 264, 692
MPI_HOST, 377, 691
MPI_IDENT, 258, 266, 691
MPI_IN_PLACE, 16, 152, 179, 627, 648, 687
MPI_INFO, 224
MPI_INFO_ENV, 400, 401, 691, 828
MPI_INFO_NULL, 333, 414, 422, 431, 545,
547, 556, 692
MPI_INT, 12, 28, 88, 184, 590, 591, 639, 681,
683, 689
MPI_INT16_T, 28, 184, 689, 831, 833
MPI_INT32_T, 28, 184, 689, 831, 833
MPI_INT64_T, 28, 184, 689, 831, 833
MPI_INT8_T, 28, 184, 689, 831, 833
MPI_INTEGER, 27, 35, 184, 639, 640, 683,
690 690
MPI_INTEGER1, 27, 185, 690
MPI_INTEGER16, 185, 690
MPI_INTEGER2, 27, 185, 591, 690
NEL INTERCEPT A OF 105 400
MPI_INTEGER4, 27, 185, 690
MPI_INTEGER8, 185, 644, 690
MPI_INTEGER8, 185, 644, 690 MPI_INTEGER_KIND, 15, 688
MPI_INTEGER8, 185, 644, 690 MPI_INTEGER_KIND, 15, 688 MPI_IO, 377, 691
MPI_INTEGER8, 185, 644, 690 MPI_INTEGER_KIND, 15, 688 MPI_IO, 377, 691 MPI_KEYVAL_INVALID, 301–303, 687
MPI_INTEGER8, 185, 644, 690 MPI_INTEGER_KIND, 15, 688 MPI_IO, 377, 691 MPI_KEYVAL_INVALID, 301–303, 687 MPI_LAND, 184, 185, 692
MPI_INTEGER8, 185, 644, 690 MPI_INTEGER_KIND, 15, 688 MPI_IO, 377, 691 MPI_KEYVAL_INVALID, 301–303, 687 MPI_LAND, 184, 185, 692 MPI_LASTUSEDCODE, 395, 694
MPI_INTEGER8, 185, 644, 690 MPI_INTEGER_KIND, 15, 688 MPI_IO, 377, 691 MPI_KEYVAL_INVALID, 301–303, 687 MPI_LAND, 184, 185, 692

MPI_LOCK_SHARED, 493, 494, 687	1
MPI_LOGICAL, 27, 185, 690	2
MPI_LONG, 28, 184, 689	3
MPI_LONG_DOUBLE, 28, 185, 689	4
MPI_LONG_DOUBLE_INT, 188, 691	5
MPI_LONG_INT, 188, 189, 691	
MPI_LONG_LONG, 28, 184, 689, 833	6
MPI_LONG_LONG_INT, 28, 184, 689, 833	7
MPI_LOR, 184, 185, 692	8
MPI_LXOR, 184, 185, 692	9
MPI_MAX, 182, 184, 185, 204, 692	10
MPI_MAX_DATAREP_STRING, 15, 557,	11
594, 688	
MPI_MAX_ERROR_STRING, 15, 391, 396,	12
688	13
MPI_MAX_INFO_KEY, 15, 392, 409, 412, 688	14
MPI_MAX_INFO_VAL, 15, 392, 409, 688	15
MPI_MAX_LIBRARY_VERSION_STRING,	16
· · · · · · · · · · · · · · · · · · ·	17
15, 376, 688, 826 MDI MAX OD JECTE NAME 15, 214, 217	
MPI_MAX_OBJECT_NAME, 15, 314–317,	18
688, 828, 834	19
MPI_MAX_PORT_NAME, 15, 431, 688	20
MPI_MAX_PROCESSOR_NAME, 15, 379,	21
688, 835	22
MPI_MAXLOC, 184, 187, 188, 191, 692	23
MPI_MESSAGE_NO_PROC, 73–75, 687, 827	
MPI_MESSAGE_NULL, 73–75, 692, 827	24
MPI_MIN, 184, 185, 692	25
MPI_MINLOC, 184, 187, 188, 191, 692	26
MPI_MODE_APPEND, 544, 545, 695	27
MPI_MODE_CREATE, 544, 545, 553, 695	28
MPI_MODE_DELETE_ON_CLOSE, 544–546,	29
695	
MPI_MODE_EXCL, 544, 545, 695	30
MPI_MODE_NOCHECK, 494, 499, 500, 695	31
MPI_MODE_NOPRECEDE, 488, 499, 500,	32
695	33
MPI_MODE_NOPUT, 499, 695	34
MPI_MODE_NOSTORE, 499, 695	35
MPI_MODE_NOSUCCEED, 499, 500, 695	
MPI_MODE_RDONLY, 544, 545, 550, 695	36
MPI_MODE_RDWR, 544, 545, 695	37
MPI_MODE_SEQUENTIAL, 544, 545, 548,	38
555, 560, 565, 578, 600, 695, 836	39
MPI_MODE_UNIQUE_OPEN, 544, 545, 695	40
MPI_MODE_WRONLY, 544, 545, 695	41
MPI_NO_OP, 450, 474, 476, 692, 823	
MPI_NULL_COPY_FN, 18, 301, <u>614</u> , 694	42
MPI_NULL_DELETE_FN, 18, 301, <u>614</u> , 694	43
MPI_OFFSET, 28, 185, 689, 690, 831, 833	44
MPI_OFFSET_KIND, 15, <u>17</u> , 28, 602, 649, 688	45
MPI_OP_NULL, 194, 692	46
MPI_ORDER_C, 15, 100, 103, 104, 696	47
MPI_ORDER_FORTRAN, 15, 100, 103, 696	48

1	MPI_PACKED, 12, 27, 28, 35, 36, 139, 140,
2	144, 591, 682, 689, 690
3	MPI_PROC_NULL, 26, 30–32, 69, 73–75, 86,
4	154, 156, 158, 160, 168, 170, 184, 257,
5	344, 348, 349, 377, 378, 455, 464, 687,
6	827, 834, 836
7	MPI_PROD, 184, 185, 692
8	MPI_REAL, 27, 35, 185, 591, 639, 640, 647, 690
9	MPI_REAL16, 185, 690
10	MPI_REAL2, 27, 185, 690
11	MPI_REAL4, 27, 185, 639, 644, 690
12	MPI_REAL8, 27, 185, 639, 690, 831
13	MPI_REPLACE, 472–474, 476, 516, 692, 832,
14	836
15	MPI_REQUEST_NULL, 55–58, 61–64, 526,
16	692 NDL DOOT 154, 605
10	MPI_ROOT, 154, 687 MPI_SEEK_CUR, 572, 579, 696
18	MPI_SEEK_COR, 572, 579, 696 MPI_SEEK_END, 572, 579, 696
	MPI_SEEK_SET, 572, 573, 579, 696
19	MPI_SHORT, 28, 184, 689
20	MPI_SHORT_INT, 188, 691
21	MPI_SIGNED_CHAR, 28, 184, 186, 187, 689,
22	833
23	MPI_SIMILAR, 258, 266, 291, 691
24	MPI_SOURCE, 32, 206, 688, 822, 829
25	MPI_STATUS_IGNORE, 10, 15, 34, 35, 525,
26	560, 626, 648, 672, 673, 682, 696, 697, 827
27	MPI_STATUS_SIZE, 15, 32, 628, 688, 829
28	MPI_STATUSES_IGNORE, 14, 15, 34, 35,
29	525, 527, 648, 672, 673, 696, 697
30	MPI_SUBARRAYS_SUPPORTED, 15, 622,
31	623, 626-630, 634-637, 650, 651, 688,
32	828
33	MPI_SUBVERSION, 15, 376, 697
34	MPI_SUCCESS, 19, 55, 63, 65, 300, 301, 303–306, 308, 309, 325, 391, 392, 397,
35	303-300, 508, 509, 525, 591, 592, 591, 392, 391, 398, 422, 596, 614, 685, 822
36	MPI_SUM, 184, 185, 472, 681, 692
37	MPI_T_BIND_MPI_COMM, 698
38	MPI_T_BIND_MPI_DATATYPE, 698
39	MPI_T_BIND_MPI_ERRHANDLER, 698
40	MPI_T_BIND_MPI_FILE, 698
41	MPI_T_BIND_MPI_GROUP, 698
42	MPI_T_BIND_MPI_INFO, 698
43	MPI_T_BIND_MPI_MESSAGE, 698
44	MPI_T_BIND_MPI_OP, 698 MPI_T_BIND_MPI_REQUEST, 698
45	MPI_T_BIND_MPI_WIN, 698
46	MPI_T_BIND_NO_OBJECT, 698
47	MPI_T_CVAR_HANDLE_NULL, 697
48	MPI_T_ENUM_NULL, 697

MPI_T_ERR_CANNOT_INIT, 687 MPI_T_ERR_CVAR_SET_NEVER, 687 MPI_T_ERR_CVAR_SET_NOT_NOW, 687 MPI_T_ERR_INVALID, 687, 824 MPI_T_ERR_INVALID_HANDLE, 687 MPI_T_ERR_INVALID_INDEX, 687 MPI_T_ERR_INVALID_ITEM, 687 MPI_T_ERR_INVALID_NAME, 687, 824 MPI_T_ERR_INVALID_SESSION, 687 MPI_T_ERR_MEMORY, 687 MPI_T_ERR_NOT_INITIALIZED, 687 MPI_T_ERR_OUT_OF_HANDLES, 687 MPI_T_ERR_OUT_OF_SESSIONS, 687 MPI_T_ERR_PVAR_NO_ATOMIC, 687 MPI_T_ERR_PVAR_NO_STARTSTOP, 687 MPI_T_ERR_PVAR_NO_WRITE, 687 MPI_T_PVAR_ALL_HANDLES, 698 MPI_T_PVAR_CLASS_AGGREGATE, 698 MPI_T_PVAR_CLASS_COUNTER, 698 MPI_T_PVAR_CLASS_GENERIC, 698 MPI_T_PVAR_CLASS_HIGHWATERMARK, 698 MPI_T_PVAR_CLASS_LEVEL, 698 MPI_T_PVAR_CLASS_LOWWATERMARK, 698 MPI_T_PVAR_CLASS_PERCENTAGE, 698 MPI_T_PVAR_CLASS_SIZE, 698 MPI_T_PVAR_CLASS_STATE, 698 MPI_T_PVAR_CLASS_TIMER, 698 MPI_T_PVAR_HANDLE_NULL, 697 MPI_T_PVAR_SESSION_NULL, 697 MPI_T_SCOPE_ALL, 698 MPI_T_SCOPE_ALL_EQ, 698 MPI_T_SCOPE_CONSTANT, 698 MPI_T_SCOPE_GROUP, 698 MPI_T_SCOPE_GROUP_EQ, 698 MPI_T_SCOPE_LOCAL, 698 MPI_T_SCOPE_READONLY, 698 MPI_T_VERBOSITY_MPIDEV_ALL, 697 MPI_T_VERBOSITY_MPIDEV_BASIC, 697 MPI_T_VERBOSITY_MPIDEV_DETAIL, 697 MPI_T_VERBOSITY_TUNER_ALL, 697 MPI_T_VERBOSITY_TUNER_BASIC, 697 MPI_T_VERBOSITY_TUNER_DETAIL, 697 MPI_T_VERBOSITY_USER_ALL, 697 MPI_T_VERBOSITY_USER_BASIC, 697 MPI_T_VERBOSITY_USER_DETAIL, 697 MPI_TAG, 32, 206, 688, 822, 829 MPI_TAG_UB, 29, 377, 677, 680, 691 MPI_THREAD_FUNNELED, 536, 695 MPI_THREAD_MULTIPLE, 536, 537, 539, 695.824 MPI_THREAD_SERIALIZED, 536, 695

MPI_THREAD_SINGLE, 536, 537, 695 MPI_TYPE_DUP_FN, <u>308</u>, 693 MPI_TYPE_NULL_COPY_FN, 308, 693 MPI_TYPE_NULL_DELETE_FN, 308, 693, 829 MPI_TYPECLASS_COMPLEX, 645, 696 MPI_TYPECLASS_INTEGER, 645, 696 MPI_TYPECLASS_REAL, 645, 696 MPI_UB, 4, 18, 618, 826 MPI_UINT16_T, 28, 184, 689, 831, 833 MPI_UINT32_T, 28, 184, 689, 831, 833 MPI_UINT64_T, 28, 184, 689, 831, 833 MPI_UINT8_T, 28, 184, 689, 831, 833 MPI_UNDEFINED, 33, 34, 61, 62, 65, 110, 113, 115, 120, 141, 256, 257, 274, 275, 335, 346, 347, 641, 687, 827, 833 MPI_UNEQUAL, 258, 266, 291, 691 MPI_UNIVERSE_SIZE, 418, 439, 440, 694 MPI_UNSIGNED, 28, 184, 689 MPI_UNSIGNED_CHAR, 28, 184, 186, 187, 689 MPI_UNSIGNED_LONG, 28, 184, 689 MPI_UNSIGNED_LONG_LONG, 28, 184, 689, 833 MPI_UNSIGNED_SHORT, 28, 184, 689 MPI_UNWEIGHTED, 16, 330, 332–334, 342, 343, 648, 696, 826, 832 MPI_VAL, 12, 670 MPI_VERSION, 15, 376, 697 MPI_WCHAR, 28, 186, 187, 317, 591, 689, 833 MPI_WEIGHTS_EMPTY, 16, 330, 332, 648, 696, 826 MPI_WIN_BASE, 460, 461, 681, 694 MPI_WIN_CREATE_FLAVOR, 460, 461, 694 MPI_WIN_DISP_UNIT, 460, 461, 694 MPI_WIN_DUP_FN, <u>305</u>, 693 MPI_WIN_FLAVOR_ALLOCATE, 461, 694 MPI_WIN_FLAVOR_CREATE, 461, 694 MPI_WIN_FLAVOR_DYNAMIC, 461, 694 MPI_WIN_FLAVOR_SHARED, 455, 461, 694 MPI_WIN_MODEL, 460, 461, 483, 694 MPI_WIN_NULL, 460, 692 MPI_WIN_NULL_COPY_FN, 305, 693 MPI_WIN_NULL_DELETE_FN, 305, 693 MPI_WIN_SEPARATE, 461, 483, 503, 694 MPI_WIN_SIZE, 460, 461, 694 MPI_WIN_UNIFIED, 461, 483, 503, 512, 694 MPI_WTIME_IS_GLOBAL, 377, 378, 399, 677, 691

1

 $\mathbf{2}$

3

4

5

6

7

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

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, 28, <u>89</u> , 89, 92, 94, 97,
14	107 - 109, 112 - 114, 123, 144 - 146, 449,
15	451, 453, 456, 457, 465, 467, 471, 473,
16	475, 476, 478-481, 590, 594, 649, 678,
17	699
18	MPI_Comm, 12, <u>26</u> , 258, 264–270, 274, 277,
19	278, 280, 281, 290–293, 299, 302–304,
	691, 692, 699 MBL Crownt 17, 17, 28, 600, 826
20	MPI_Count, <u>17</u> , 17, 28, 699, 826 MPI_Datatype, <u>89</u> , 657, 689–692, 699
21	MPI_ERR, 391
22	MPI_Errhandler, 384, 385–390, 671, 688, 692,
23	699
24	MPI_F08_status, <u>673</u> , 697, 699, 829
25	MPI_File, 388, 389, <u>543</u> , 546–552, 554, 556,
26	560-579, 581-587, 590, 599, 600, 671,
27	692, 699
28	MPI_Fint, <u>670</u> , 670, 697, 699, 833
29	MPI_Group, 256, 257, 257–262, 264, 291, 461,
30	$489, \overline{490, 549}, 671, 692, 699$
31	MPI_Info, 379, <u>409</u> , 409–414, 418, 421, 423,
32	430, 434 436, 443, 462, 463, 543, 546,
33	551, 552, 554, 671, 691, 692, 699, 836
34	MPI_Message, <u>71</u> , 671, 687, 692, 699, 827
	MPI_Offset, <u>17</u> , 17, 28, 547–549, 554, 556,
35	560-565, 572, 573, 578, 579, 581, 582,
36	594, <u>602</u> , 602, 669, 699 MPI_Op, 182, <u>191</u> , 194, 196–200, 202, 203,
37	$\begin{array}{c} \text{M11} \underline{-0}p, 182, \underline{191}, 194, 190-200, 202, 203, \\ 218-223, 236-241, 471, 473, 475, 480, \end{array}$
38	481, 671, 692, 699
39	MPI_Request, 51–54, <u>56</u> , 57, 58, 60–65, 67, 76,
40	78-82, 524, 527, 563-565, 569-571,
41	575, 576, 650, 671, 692, 699
42	MPI_Status, <u>30</u> , 32–34, 56, 57, 60–65, 67–69,
43	71, 73, 77, 83, 85, 119, 525, 530, 531,
44	560-562, 566-568, 574, 575, 577, 578,
45	582-587, 625, 672-674, 696, 699, 822,
46	826, 829
47	MPI_T_cvar_handle, 697
48	MPI_T_enum, 697

MPI_T_pvar_handle, 697
MPI_T_pvar_session, 697
MPI_Win, 305–307, 387, <u>449</u> , <u>451</u> , <u>453</u> , <u>456</u> ,
459, 461-463, 465, 467, 471, 473, 475,
476, 478-481, 486, 489-498, 671, 692,
699

MPI Callback Function Prototype Index

This index lists the C typedef names for callback routines, such as those used with attribute caching or user-defined reduction operations. Fortran example prototypes are given near the text of the C name.

14
15
16
17
18
19
20
21
22
23
24
25
26
20
28
29
30
31
32
33
34
35
36
37
38
39
40
41

MPI Function Index

481, 484, 509, 515, 516, 832, 836

451-455, 458, 466, 496, 634-636, 648,

78 9 10 11MPI SET STATUS CANCELLED, 822 12MPI_ABORT, 192, 383, 401, 404, 442, 670, 835 MPI_ACCUMULATE, 447, 463, 471, 472, 474, 1314MPI_ADD_ERROR_CLASS, 394, 395 15MPI_ADD_ERROR_CODE, 395 16MPI_ADD_ERROR_STRING, 396, 396 17MPI_ADDRESS, 18, 617, 633, 826 18 MPI_AINT_ADD, 20, 106, 108, 108, 457, 824 19MPI_AINT_DIFF, 20, 106, 108, 108, 109, 457, 20MPI_ALLGATHER, 149, 153, 154, 173, 2122MPI_ALLGATHER_INIT, 231 23MPI_ALLGATHERV, 149, 153, 154, <u>174</u>, 175, 24 25MPI_ALLGATHERV_INIT, 232 26MPI_ALLOC_MEM, 379, 380, 381, 392, 2728 $\mathrm{MPI_ALLOC_MEM_CPTR},\, 380,\, 823$ 29MPI_ALLREDUCE, 149, 152-154, 184, 191, 30 31 MPI_ALLREDUCE_INIT, 237 32 MPI_ALLTOALL, 149, 153, 154, 176, 177–179, 33 34MPI_ALLTOALL_INIT, 233 MPI_ALLTOALLV, 149, 153, 154, 178, 178, 35 36 MPI_ALLTOALLV_INIT, 234 37 MPI_ALLTOALLW, 149, 153, 154, 180, 181, 38 39 MPI_ALLTOALLW_INIT, 235 40MPI_ATTR_DELETE, 18, 311, 614, 615 41MPI_ATTR_GET, 18, 311, 615, 677, 678 42MPI_ATTR_PUT, 18, 311, 615, 677, 678, 680, 4346

824

215

823, 830

216, 831

218, 831

681

48

173-175, 177, 214

196, 196, 220, 833

179, 181, 217, 831

 $\mathbf{5}$ 6

MPI_BARRIER, 149, 153, 155, 155, 208, 44506-508, 604 45MPI_BARRIER_INIT, 225

MPI_BCAST, 149, 153, 156, 156, 157, 183, 47209, 245

The underlined page numbers refer to the function definitions. MPI_BCAST_INIT, 225 MPI_BSEND, 41, 49 MPI_BSEND_INIT, 79, 82 MPI_BUFFER_ATTACH, 47, 56 MPI_BUFFER_DETACH, 47, 829 MPI_CANCEL, 18, 43, 56, 68, 75, 76, 76, 77, 206, 224, 403, 477, 523, 526, 527, 616, 821 MPI_CART_COORDS, 323, 338, 339, 835 MPI_CART_CREATE, 289, 322, 323, 324, 324-326, 337, 345-348, 649, 823, 834 MPI_CART_GET, 323, 337, 337, 834 MPI_CART_MAP, 323, 346, 347, 828 MPI_CART_RANK, 323, 338, 338, 835 MPI_CART_SHIFT, 323, 343, 344, 344, 348, 835 MPI_CART_SUB, 323, 345, 345-347, 835 MPI_CARTDIM_GET, 323, 336, 337, 834 MPI_CLOSE_PORT, <u>431</u>, 431, 435 MPI_COMM_ACCEPT, 430, 432, 432, 433, 440, 441 MPI_COMM_C2F, 670 MPI_COMM_CALL_ERRHANDLER, 396, 398 MPI_COMM_COMPARE, 266, 291 MPI_COMM_CONNECT, 392, 433, 433, 440, 441 MPI_COMM_CREATE, 264, 267, 270, 270-275, 323, 832 MPI_COMM_CREATE_ERRHANDLER, 18, 384, 384, 386, 617, 702, 704, 829 MPI_COMM_CREATE_GROUP, 266, 267, 273, 273-275, 827 MPI_COMM_CREATE_KEYVAL, 18, 298, 299, 300, 301, 311, 613, 676, 677, 701, 703, 829, 834 MPI_COMM_DELETE_ATTR, 18, 298, 301-303, 304, 311, 615 MPI_COMM_DISCONNECT, 311, 423, 441, 442, 442, 443

MPI_COMM_DUP, 258, 264, 267, 267-269, 271, 292, 294, 298, 300, 304, 311, 319,

612 001 007
613, 821, 827
MPI_COMM_DUP_FN, 18, <u>300</u> , 300–302, 629,
693, 824, 829
MPI_COMM_DUP_WITH_INFO, 267, <u>268</u> ,
268, 269, 279, 827
MPI_COMM_F2C, <u>670</u>
MPI_COMM_FREE, 264, 268, <u>278</u> , 292, 294,
301, 302, 304, 311, 402, 405, 423,
441-443, 614
MPI_COMM_FREE_KEYVAL, 18, 298, <u>302</u> ,
311, 614
MPI_COMM_GET_ATTR, 18, 298, <u>303</u> , 303,
311, 377, 615, 630, 678, 680
MPI_COMM_GET_ERRHANDLER, 18, 384,
<u>386, 617, 835</u>
MPI_COMM_GET_INFO, 279, 280, <u>281</u> , 281,
822, 827
MPI_COMM_GET_NAME, 314, <u>315</u> , 315,
316, 834
MPI_COMM_GET_PARENT, 315, 383, 419,
422, 423
MPI_COMM_GROUP, 14, 256, <u>258</u> , 258, 259,
264-266, 291, 384, 835
MPI_COMM_IDUP, 264, 267, <u>268</u> , 269, 270,
289, 298, 300, 304, 311, 821, 822, 827
MPI_COMM_IDUP_WITH_INFO, 267, 269,
269, 270, 279, 822
MPI_COMM_JOIN, <u>443</u> , 444, 445
MPI_COMM_NULL_COPY_FN, 18, <u>300</u> ,
300-302, 624, 676, 693, 824, 829
$MPI_COMM_NULL_DELETE_FN, 18, \underline{301},$
301, 302, 693, 824
MPI_COMM_RANK, <u>265</u> , 265, 291, 631
MPI_COMM_RANK_F08, 631
MPI_COMM_REMOTE_GROUP, <u>291</u>
MPI_COMM_REMOTE_SIZE, 291, 292
MPI_COMM_SET_ATTR, 18, 298, 301, <u>302</u> ,
311, 614, 630, 678, 681
MPI_COMM_SET_ERRHANDLER, 18, 384,
385, 617
MPI_COMM_SET_INFO, 225, 279, <u>280</u> , 280,
822, 827
MPI_COMM_SET_NAME, <u>314</u> , 314
MPI_COMM_SIZE, <u>265</u> , 265, 266, 291
MPI_COMM_SPAWN, 400, 407, 416, 417, <u>418</u> ,
$418,419,421{-}423,425{-}427,440,441$
MPI_COMM_SPAWN_MULTIPLE, 401, 407,
$416, 417, 423, \underline{424}, 425, 441$
MPI_COMM_SPLIT, 267, 270, 271, <u>274</u> ,
$274-276, 319, 323, 324, 326, \overline{345}, 347,$
348, 832
MPI_COMM_SPLIT_TYPE, <u>277</u> , 279, 827
MPI_COMM_TEST_INTER, 289, 290
MPI_COMPARE_AND_SWAP, 447, 464, <u>476</u> ,
$\underline{\mathbf{H}}_{\mathbf{L}} = \underbrace{\mathbf{C}}_{\mathbf{L}} \underbrace{\mathbf{H}}_{\mathbf{L}} \underbrace{\mathbf{L}}_{\mathbf{L}} \underbrace{\mathbf{L}} \mathbf$

515	1
MPI_CONVERSION_FN_NULL, <u>596</u> , 693,	2
824	3
MPI_CWIN_GET_ATTR, 630	4
MPI_DIMS_CREATE, 323, <u>325</u> , 325, 326, 822	
MPI_DIST_GRAPH_CREATE, 279, 322, 323,	5
328, <u>331</u> , 332–334, 343, 348, 832	6
	7
MPI_DIST_GRAPH_CREATE_ADJACENT,	8
279, 322, 323, 328, <u>329</u> , 330, 334, 343,	
348, 828, 832	9
MPI_DIST_GRAPH_NEIGHBORS, 323, 341,	10
$\underline{342}, 343, 348, 828, 832$	11
MPI_DIST_GRAPH_NEIGHBORS_COUNT,	12
$323, 341, \underline{342}, 343, 825, 832$	
MPI_DUP_FN, 18, 301, 614, 694	13
MPI_ERRHANDLER_C2F, 671	14
MPI_ERRHANDLER_CREATE, 18, 617, 826,	15
829	16
MPI_ERRHANDLER_F2C, <u>671</u>	17
MPI_ERRHANDLER_FREE, 384, <u>390</u> , 402,	18
835	19
MPI_ERRHANDLER_GET, 18, 617, 826, 835	20
MPI_ERRHANDLER_SET, 18, 617, 826	21
MPI_ERROR_CLASS, <u>391</u> , 391, 394	
MPI_ERROR_STRING, <u>390</u> , 391, 394, 396	22
MPI_EXSCAN, 150, 153, 184, 191, <u>203</u> , 203,	23
223, 831	24
MPI_EXSCAN_INIT, <u>241</u>	25
MPI_F_SYNC_REG, 107, 517, 622, <u>638</u> , 638,	
639, 659–663, 665, 830	26
	27
MPI_FETCH_AND_OP, 447, 464, 472, 474,	28
<u>475, 475</u>	29
MPI_FILE_C2F, <u>671</u>	30
MPI_FILE_CALL_ERRHANDLER, <u>397</u> , 398	
MPI_FILE_CLOSE, 443, 543, 544, <u>546</u> , 546	31
MPI_FILE_CREATE_ERRHANDLER, 384,	32
388, 389, 702, 704, 829	33
MPI_FILE_DELETE, 545, <u>546</u> , 546, 547, 551,	34
553, 608	
MPI_FILE_F2C, 671	35
MPI_FILE_GET_AMODE, <u>550</u> , 550	36
MPI_FILE_GET_ATOMICITY, 599, 600	37
·	38
MPI_FILE_GET_BYTE_OFFSET, 565, <u>573</u> ,	39
573, 579	
MPI_FILE_GET_ERRHANDLER, 384, <u>389</u> ,	40
608, 835	41
MPI_FILE_GET_GROUP, <u>549</u> , 549	42
MPI_FILE_GET_INFO, 551, <u>552</u> , 552, 553,	43
822, 836	
MPI_FILE_GET_POSITION, <u>572</u> , 573	44
MPI_FILE_GET_POSITION_SHARED, 578,	45
<u>579, 579, 600</u>	46
MPI_FILE_GET_SIZE, <u>549</u> , 549, 603	47
MPI_FILE_GET_TYPE_EXTENT, 589, <u>590</u> ,	48

594, 602, 609, 836 MPI_FILE_SYNC, 546, 558, 597, 598, 600,

1	596
2	MPI_FILE_GET_VIEW, <u>556</u> , 557
3	MPI_FILE_IXXX, 558
4	MPI_FILE_IREAD, 557, <u>569</u> , 569, 580, 597,
5	598
6	MPI_FILE_IREAD_ALL, 557, <u>570</u> , 570, 824
7	MPI_FILE_IREAD_AT, 557, 563, 563
8	MPI_FILE_IREAD_AT_ALL, 557, <u>563</u> , 564, 824
9	MPI_FILE_IREAD_SHARED, 557, <u>575</u> , 576
10	MPI_FILE_IWRITE, 557, <u>571</u> , 571
11	MPI_FILE_IWRITE_ALL, 557, 571, 572, 824
12	MPI_FILE_IWRITE_AT, 557, <u>564</u> , 564
13	MPI_FILE_IWRITE_AT_ALL, 557, <u>565</u> , 565,
14	824
15	MPI_FILE_IWRITE_SHARED, 557, 576, 576
16	MPI_FILE_OPEN, 393, 535, <u>543</u> , 543–545, 551, 553, 555, 573, 602, 603, 608, 609
17	MPI_FILE_PREALLOCATE, 547, 548, 548,
18	598, 603
19	MPI_FILE_READ, 557, <u>566</u> , 566, 567, 569,
20	602, 603
21	MPI_FILE_READ_ALL, 557, <u>567</u> , 567, 570,
22	580, 581
23	MPI_FILE_READ_ALL_BEGIN, 557, 580,
24	581, <u>583</u> , 597, 664 MPI_FILE_READ_ALL_END, 557, 580, 581,
25	$\frac{584}{597}, 664$
26	MPI_FILE_READ_AT, 557, <u>560</u> , 560, 561, 563
27	MPI_FILE_READ_AT_ALL, 557, <u>561</u> , 561,
28	564
29	MPI_FILE_READ_AT_ALL_BEGIN, 557,
30	<u>581, 664</u> MPI_FILE_READ_AT_ALL_END, 557, <u>582</u> ,
31	$\begin{array}{c} \text{MP1_FILE_READ_A1_ALL_END, 557, } \underline{562}, \\ 664 \end{array}$
32	MPI_FILE_READ_ORDERED, 557, 577, 577
33	MPI_FILE_READ_ORDERED_BEGIN, 557,
34	<u>585</u> , 664
35	MPI_FILE_READ_ORDERED_END, 557,
36	<u>586,</u> 664
37	MPI_FILE_READ_SHARED, 557, <u>574</u> , 574,
38	576, 577 MPI_FILE_SEEK, 572, 572, 573
39	MPI_FILE_SEEK, <u>572</u> , 572, 578, 578, 579,
40	600
41	MPI_FILE_SET_ATOMICITY, 545, 598, 599,
42	599
43	MPI_FILE_SET_ERRHANDLER, 384, <u>389</u> ,
44	608
45	MPI_FILE_SET_INFO, <u>551</u> , 551–553, 822, 836 MPI_FILE_SET_SIZE, <u>547</u> , 547, 548, 598,
46	$\begin{array}{c} \text{MP1_FILE_SE1_SIZE, } \underline{547}, 547, 548, 598, \\ 601, 603 \end{array}$
47	MPI_FILE_SET_VIEW, 101, 393, 544,
48	551-553, 554, 555, 556, 573, 579, 588,
	, , , , , , , , , , , , , , , , ,

600, 605 MPI_FILE_WRITE, 557, 558, 568, 568, 569, 571,602 $MPI_FILE_WRITE_ALL, 557, \underline{568}, 569, 572$ MPI_FILE_WRITE_ALL_BEGIN, 557, 584, 650, 664 MPI_FILE_WRITE_ALL_END, 557, 585, 664 MPI_FILE_WRITE_AT, 557, 558, 561, 562, 564MPI_FILE_WRITE_AT_ALL, 557, 562, 562, 565MPI_FILE_WRITE_AT_ALL_BEGIN, 557, 582,664 MPI_FILE_WRITE_AT_ALL_END, 557, 583, 664 MPI_FILE_WRITE_ORDERED, 557, 577, 578, 578 MPI_FILE_WRITE_ORDERED_BEGIN, 557, 586, 664 MPI_FILE_WRITE_ORDERED_END, 557, 587,664 MPI_FILE_WRITE_SHARED, 557, 558, 575, 575-578 MPI_FINALIZE, 15, 22, 376, 377, 401, 401-406, 442, 534, 544, 670, 672, 673, 828.835 MPI_FINALIZED, 400, 403, 405, 406, 406, 532, 538, 670, 824 MPI_FREE_MEM, <u>380</u>, 381, 392, 452, 453 MPI_GATHER, 149, 152, 153, <u>157</u>, 160, 167, 168, 173, 174, 183, 210 MPI_GATHER_INIT, 226 MPI_GATHERV, 149, 153, 159, 160, 161, 169, 175, 211 MPI_GATHERV_INIT, 227 MPI_GET, 447, 463, 467, 468, 474, 480, 484, 506, 508, 509, 517, 662, 836 MPI_GET_ACCUMULATE, 447, 463, 472, 473, 474, 475, 482, 509, 515, 823 MPI_GET_ADDRESS, 18, 89, 106, 107, 107-109, 121, 457, 617, 633, 653, 657, 658, 675, 676 MPI_GET_COUNT, 33, 33, 34, 55, 120, 477, 531, 559, 826, 833 MPI_GET_ELEMENTS, 55, 119-121, 531, 532, 559, 826 MPI_GET_ELEMENTS_X, 55, 119–121, 531, 560, 826 MPI_GET_LIBRARY_VERSION, 376, 376, 400, 403, 532, 823, 824, 826 MPI_GET_PROCESSOR_NAME, 378, 379,

Unofficial Draft for Comment Only

MPI_GET_VERSION, <u>375</u>, 376, 400, 403, 532, 538, 637, 824 MPI_GRAPH_CREATE, 322, 323, 326, 326, 328, 333, 336, 340, 347, 348, 834 MPI_GRAPH_GET, 323, 336, 336 MPI_GRAPH_MAP, 323, <u>347</u>, 348 MPI_GRAPH_NEIGHBORS, 323, 339, 340, 348, 832 MPI_GRAPH_NEIGHBORS_COUNT, 323, 339, 340, 832 MPI_GRAPHDIMS_GET, 323, 335, 336 MPI_GREQUEST_COMPLETE, 524–526, 527, 527 MPI_GREQUEST_START, 524, 525, 702, 704, 832 MPI_GROUP_C2F, 671 MPI_GROUP_COMPARE, 257, 261 MPI_GROUP_DIFFERENCE, 260 MPI_GROUP_EXCL, 261, 261, 263 MPI_GROUP_F2C, <u>671</u> MPI_GROUP_FREE, <u>264</u>, 264–266, 384, 402, 835 MPI_GROUP_INCL, 260, 261, 262 MPI_GROUP_INTERSECTION, 259 MPI_GROUP_RANGE_EXCL, 263, 263 MPI_GROUP_RANGE_INCL, 262, 262 MPI_GROUP_RANK, 256, 266 MPI_GROUP_SIZE, 256, 265 MPI_GROUP_TRANSLATE_RANKS, 257, 257,834 MPI_GROUP_UNION, 259 MPI_IALLGATHER, 149, 153, 154, 213 MPI_IALLGATHERV, 149, 153, 154, 214 MPI_IALLREDUCE, 149, 153, 154, 219 MPI_IALLTOALL, 149, 153, 154, 215 MPI_IALLTOALLV, 149, 153, 154, 216 MPI_IALLTOALLW, 149, 153, 154, 217 MPI_IBARRIER, 149, 153, 206, 207, 208, 245 MPI_IBCAST, 149, 153, 208, 209, 249 MPI_IBSEND, 52, 56, 82 MPI_IEXSCAN, 150, 153, 223 MPI_IGATHER, 149, 153, 209 MPI_IGATHERV, 149, 153, 210 MPI_IMPROBE, 68, 71, 72, 72, 73, 75, 534, 822, 827 MPI_IMRECV, 71–73, 75, 75, 827 MPI_INEIGHBOR_ALLGATHER, 323, 358, 828 MPI_INEIGHBOR_ALLGATHERV, 323, 359, 828 MPI_INEIGHBOR_ALLTOALL, 323, 360, 828 MPI_INEIGHBOR_ALLTOALLV, 323, 361, 828

MPI_INEIGHBOR_ALLTOALLW, <u>324</u> , <u>362</u> ,	1
828	2
MPI_INFO_C2F, <u>671</u>	3
MPI_INFO_CREATE, <u>410</u> , 410	4
MPI_INFO_DELETE, 392, <u>411</u> , 411, 413	5
MPI_INFO_DUP, <u>414</u> , 414	
MPI_INFO_F2C, <u>671</u>	6
MPI_INFO_FREE, 281, 402, <u>414</u> , 463, 552	7
MPI_INFO_GET, 409, 411, 836	8
MPI_INFO_GET_NKEYS, 409, 413, 413, 836	9
MPI_INFO_GET_NTHKEY, 409, 413, 836	10
MPI_INFO_GET_NKEYS, 409, <u>413</u> , 413, 836 MPI_INFO_GET_NTHKEY, 409, <u>413</u> , 836 MPI_INFO_GET_VALUELEN, 409, <u>412</u> , 836	11
MPI_INFO_SET, <u>410</u> , 411–413	12
MPI_INIT, 15, 22, 255, 376, 377, <u>399</u> , 399,	
400, 403–406, 419–421, 423, 439, 440,	13
535-538, 669, 670, 672, 673, 826, 828,	14
830, 832	15
MPI_INIT_THREAD, 255, 400, 405, <u>535</u> ,	16
536-538, 669, 826, 828, 832	17
MPI_INITIALIZED, 400, 403, <u>404</u> , 404–406,	18
532, 538, 670, 824	19
MPI_INTERCOMM_CREATE, 267, 273, 274,	20
292, <u>293</u> , 294, 827	
MPI_INTERCOMM_MERGE, 267, 273, 289,	21
292, 293, 294, 829	22
MPI_IPROBE, 34, <u>68</u> , 68–73, 75, 534, 827	23
MPI_IRECV, <u>54</u> , 75, 651, 652, 655, 656	24
MPI_IREDUCE, 149, 153, 154, <u>218</u> , 219	25
MPI_IREDUCE_SCATTER, 149, 153, 154,	26
<u>221</u>	27
MPI_IREDUCE_SCATTER_BLOCK, 149,	28
$153, 154, \underline{220}$	29
MPI_IRSEND, 54	30
MPI_IS_THREAD_MAIN, 532, 536, <u>538</u> , 824	
MPI_ISCAN, 150, 153, <u>222</u>	31
MPI_ISCATTER, 149, 153, <u>211</u>	32
MPI_ISCATTERV, 149, 153, <u>212</u>	33
MPI_ISEND, <u>52</u> , 82, 629, 630, 633, 650, 652,	34
657	35
MPI_ISSEND, <u>53</u>	36
MPI_KEYVAL_CREATE, 18, <u>613</u> , 615, 705	37
MPI_KEYVAL_FREE, 18, 311, <u>614</u>	38
MPI_LOOKUP_NAME, 392, 430, 435, <u>436</u> ,	
436	39
MPI_MESSAGE_C2F, <u>671</u> , 827	40
MPI_MESSAGE_F2C, <u>671</u> , 827 MPI_MPROBE, 68, 71, 72, <u>73</u> , 73, 75, 534, 827	41
	42
MPI_MRECV, 71–73, <u>74</u> , 74, 75, 827 MPI_NEIGHBOR_ALLGATHER, <u>323</u> , <u>349</u> ,	43
$\begin{array}{c} \text{MP1_NEIGHBOR_ALLGATHER, 323, } \underline{349}, \\ 351, 352, 358, 828 \end{array}$	44
MPI_NEIGHBOR_ALLGATHER_INIT, <u>363</u>	45
MPI_NEIGHBOR_ALLGATHERV, 323, <u>351</u> ,	46
359, 828	47
MPI NEIGHBOR ALLGATHERV INIT. 364	48

1	MPI_NEIGHBOR_ALLTOALL, <u>323</u> , <u>353</u> , 354,
2	360, 828
3	MPI_NEIGHBOR_ALLTOALL_INIT, <u>365</u>
4	MPI_NEIGHBOR_ALLTOALLV, 323, <u>354</u> ,
5	361, 828
6	MPI_NEIGHBOR_ALLTOALLV_INIT, <u>366</u>
7	MPI_NEIGHBOR_ALLTOALLW, 323, 356,
	356, 363, 828
8	MPI_NEIGHBOR_ALLTOALLW_INIT, <u>368</u>
9	MPI_NULL_COPY_FN, 18, 19, 301, <u>614</u> , 694
10	MPI_NULL_DELETE_FN, 18, 301, <u>614</u> , 694
11	MPI_OP_C2F, <u>671</u>
12	MPI_OP_COMMUTATIVE, <u>198</u> , 831
13	MPI_OP_CREATE, <u>191</u> , 191, 193, 629, 700,
14	703, 829 MPI_OP_F2C, <u>671</u>
15	MPI_OP_F2C, <u>071</u> MPI_OP_FREE, <u>194</u> , 402
16	MPI_OPEN_PORT, <u>430</u> , 430, 432, 433,
17	$\begin{array}{c} \text{MF1_OFEN_FOR1, } \underline{430}, 430, 430, 432, 433, \\ 435-437 \end{array}$
18	MPI_PACK, 49, <u>138</u> , 141, 144, 591, 595
	MPI_PACK_EXTERNAL, 8, 144, <u>145</u> , 643,
19	833
20	MPI_PACK_EXTERNAL_SIZE, 146, 147
21	MPI_PACK_SIZE, 49, <u>141</u> , 141, 827
22	MPI_PROBE, 31, 34, 35, 68, <u>69</u> , 69–71, 73, 75,
23	534, 827
24	MPI_PUBLISH_NAME, 430, <u>434</u> , 434–436
25	MPI_PUT, 447, 463, <u>465</u> , 467, 472, 478, 484,
26	490, 500, 502, 507, 508, 517, 650, 662,
27	836
28	MPI_QUERY_THREAD, 532, <u>538</u> , 539, 824
29	MPI_RACCUMULATE, 447, 463, 464, 472,
	$474, \underline{480}, 481$
30	MPI_RECV, 26, <u>30</u> , 32, 34, 69, 71, 72, 88, 118,
31	119, 139, 140, 150, 158, 246, 532, 604,
32	657, 658, 661, 662
33	MPI_RECV_INIT, <u>81</u> , 81
34	MPI_REDUCE, 149, 153, 154, <u>182</u> , 182–184,
35	191-194, 196, 199, 201-203, 219, 472, 474, 476, 822
36	474, 476, 832 MDI REDUCE INIT 226
37	MPI_REDUCE_INIT, <u>236</u> MPI_REDUCE_LOCAL, <u>183</u> , <u>184</u> , <u>191</u> , <u>197</u> ,
38	829. 831
39	MPI_REDUCE_SCATTER, 149, 153, 154,
40	184, 191, 200, 200, 201, 222
41	MPI_REDUCE_SCATTER_BLOCK, 149,
	153, 154, 184, 191, <u>199</u> , 199, 200, 221,
42	831
43	MPI_REDUCE_SCATTER_BLOCK_INIT,
44	238
45	MPI_REDUCE_SCATTER_INIT, 239
46	MPI_REGISTER_DATAREP, 393, 593,
47	594–596, 609, 702, 705
48	$MPI_REQUEST_C2F, \underline{671}$

MPI_REQUEST_F2C, 671 MPI_REQUEST_FREE, <u>58</u>, 58, 76, 82, 206, 224, 401, 402, 477, 526, 527, 831 MPI_REQUEST_GET_STATUS, 35, 67, 68, 525, 831 MPI_RGET, 447, 463, 464, <u>479</u>, 480 MPI_RGET_ACCUMULATE, 447, 464, 472, 474, <u>481</u>, 482 MPI_RPUT, 447, 463, 464, 478, 478, 479 MPI_RSEND, 42 MPI_RSEND_INIT, 80 MPI_SCAN, 150, 153, 184, 191, 202, 202, 204, 222MPI_SCAN_INIT, 240 MPI_SCATTER, 149, 153, <u>167</u>, 167, 169, 170, 199, 212 MPI_SCATTER_INIT, 229 MPI_SCATTERV, 149, 153, 169, 169-171, 201, 213 MPI_SCATTERV_INIT, 230 MPI_SEND, 25, <u>26</u>, 27, 34, 37, 88, 117, 118, 139, 246, 544, 604, 657, 658, 661, 662 MPI_SEND_INIT, <u>78</u>, 82 MPI_SENDRECV, 84, 343 MPI_SENDRECV_REPLACE, 85 MPI_SIZEOF, 622, 645, 646 MPI_SSEND, 41 MPI_SSEND_INIT, 79 MPI_START, <u>81</u>, 81-83, 224, 657 MPI_STARTALL, <u>82</u>, 82, 224, 657 MPI_STATUS_C2F, 672 MPI_STATUS_C2F08, 673, 829 MPI_STATUS_F082C, 673, 829 MPI_STATUS_F082F, 674, 829 MPI_STATUS_F2C, 672 MPI_STATUS_F2F08, 674, 829 MPI_STATUS_SET_CANCELLED, 532 MPI_STATUS_SET_ELEMENTS, <u>530</u>, 531 MPI_STATUS_SET_ELEMENTS_X, 531, 531, 826 MPI_T_CATEGORY_GET_INDEX, 824 MPI_T_CATEGORY_GET_INFO, 823, 824 MPI_T_CVAR_GET_INDEX, 824 MPI_T_CVAR_GET_INFO, 823, 824 MPI_T_PVAR_GET_INDEX, 824 MPI_T_PVAR_GET_INFO, 823, 824 MPI_T_PVAR_HANDLE_FREE, 823 MPI_T_PVAR_READ, 823 MPI_T_PVAR_READRESET, 823 MPI_T_PVAR_RESET, 823 MPI_T_PVAR_START, 823 MPI_T_PVAR_STOP, 823 MPI_T_PVAR_WRITE, 823

MPI_TEST, 35, 55, 56, <u>57</u> , 57, 58, 60, 62, 67,
76, 82, 224, 401, 527, 558, 559 MPI_TEST_CANCELLED, 55–57, <u>77</u> , 77, 525,
532, 560
MPI_TESTALL, 60, <u>63</u> , 64, 525–527, 530, 534
MPI_TESTANY, 60, <u>61</u> , 62, 66, 525–527, 530,
534
MPI_TESTSOME, 60, <u>65</u> , 65, 66, 525–527,
530, 534
MPI_TOPO_TEST, 323, <u>335</u> , 335
MPI_TYPE_C2F, <u>670</u>
MPI_TYPE_COMMIT, <u>115</u> , 116, 671
MPI_TYPE_CONTIGUOUS, 12, <u>89</u> , 89, 91,
110, 123, 542, 590
MPI_TYPE_CREATE_DARRAY, 12, 34, <u>102</u> ,
102, 123
MPI_TYPE_CREATE_F90_COMPLEX, 12,
$123, 125, 185, 591, 622, \underline{641}, 643$
MPI_TYPE_CREATE_F90_INTEGER, 12,
$123, 125, 185, 591, 622, \underline{641}, 643$
MPI_TYPE_CREATE_F90_REAL, 12, 123,
$125, 185, 591, 622, \underline{640}, 641-643, 831$
MPI_TYPE_CREATE_HINDEXED, 12, 18,
89, <u>94</u> , 94, 97, 98, 123, 617
MPI_TYPE_CREATE_HINDEXED_BLOCK,
12, 89, <u>96</u> , 96, 123, 827
MPI_TYPE_CREATE_HVECTOR, 12, 18,
89, <u>92</u> , 92, 123, 617
MPI_TYPE_CREATE_INDEXED_BLOCK,
12, <u>96</u> , 96, 123 MPI_TYPE_CREATE_KEYVAL, <u>298</u> , <u>308</u> ,
311, 677, 701, 704, 834
MPI_TYPE_CREATE_RESIZED, 18, 89, 110,
<u>113</u> , 114, 123, 590, 618, 829
MPI_TYPE_CREATE_STRUCT, 12, 18, 89,
<u>97, 97, 98, 111, 123, 181, 617</u>
MPI_TYPE_CREATE_SUBARRAY, 12, 15,
<u>99</u> , 101, 103, 123
MPI_TYPE_DELETE_ATTR, 298, <u>311</u> , 311,
829
MPI_TYPE_DUP, 12, <u>117</u> , 117, 123, 829
MPI_TYPE_DUP_FN, <u>308</u> , 308, 693, 824
MPI_TYPE_EXTENT, 18, 617, 826
MPI_TYPE_F2C, <u>670</u>
MPI_TYPE_FREE, <u>116</u> , 125, 309, 402
MPI_TYPE_FREE_KEYVAL, 298, <u>309</u> , 311
MPI_TYPE_GET_ATTR, 298, <u>310</u> , 311, 630,
678, 829
MPI_TYPE_GET_CONTENTS, 123, <u>124</u> ,
125, 126 MDI TYDE CET ELEMENTS 110
MPI_TYPE_GET_ELEMENTS, <u>119</u> MPI_TYPE_GET_ELEMENTS, <u>119</u>
MPI_TYPE_GET_ELEMENTS_X, <u>119</u> MPI_TYPE_GET_ENVELOPE, <u>122</u> , 123–125,
$\begin{array}{c} \text{MP1_1 YPE_GE1_ENVELOPE, } \underline{122}, 123 - 125, \\ 642 \end{array}$
044

MPI_TYPE_GET_EXTENT, 18, <u>112</u> , 115,	1
617,646,675	2
MPI_TYPE_GET_EXTENT_X, <u>112</u> , 826	3
MPI_TYPE_GET_NAME, <u>317</u> , 829 MPI_TYPE_GET_TRUE_EXTENT, <u>114</u> , 114 MPI_TYPE_GET_TRUE_EXTENT_X, <u>114</u> ,	4
MPI_TYPE_GET_TRUE_EXTENT, <u>114</u> , 114	5
$MPI_TYPE_GET_TRUE_EXTENT_X, \underline{114},$	6
114, 826	7
MPI_TYPE_HINDEXED, 18, 617, 826	
MPI_TYPE_HVECTOR, 18, 617, 826	8
MPI_TYPE_INDEXED, 12, <u>93</u> , 93–95, 123	9
MPI_TYPE_LB, 18, 617, 826	10
MPI_TYPE_MATCH_SIZE, 622, <u>645</u> , 646, 829	11
$\mathrm{MPI_TYPE_NULL_COPY_FN, } \underline{308}, 308, 693,$	12
824	13
MPI_TYPE_NULL_DELETE_FN, <u>308</u> , 693,	14
824, 829	15
MPI_TYPE_SET_ATTR, 298, <u>310</u> , 311, 630,	
678, 681, 829	16
MPI_TYPE_SET_NAME, <u>316</u> , 829	17
MPI_TYPE_SIZE, <u>109</u> , 110, 827	18
MPI_TYPE_SIZE_X, <u>109</u> , 110, 826	19
MPI_TYPE_STRUCT, 18, 617, 826	20
MPI_TYPE_UB, 18, 617, 826	21
MPI_TYPE_VECTOR, 12, <u>90</u> , 90–92, 94, 123	22
MPI_UNPACK, <u>139</u> , 140, 144, 595	23
MPI_UNPACK_EXTERNAL, 8, <u>146</u> , 643	24
MPI_UNPUBLISH_NAME, 392, <u>435</u> , 436 MPI_WAIT, 33, 35, 55, <u>56</u> , 56–61, 63, 76, 82,	
MP1_WA11, 55, 55, 55, 50, 50-01, 63, 76, 82, 206, 224, 246, 401, 523, 527, 534, 558,	25
559, 580, 597-599, 650, 651, 656, 657,	26
661	27
MPI_WAITALL, 60, <u>62</u> , 63, 64, 206, 247, 477,	28
525-527, 530, 534	29
MPI_WAITANY, 43, <u>60</u> , 60, 61, 66, 525–527,	30
530, 534	31
MPI_WAITSOME, 60, <u>64</u> , 65, 66, 525–527,	32
530, 534	33
MPI_WIN_ALLOCATE, 448, <u>451</u> , 452, 454,	
460, 461, 466, 496, 634, 636, 823	34
MPI_WIN_ALLOCATE_CPTR, 452, 823	35
MPI_WIN_ALLOCATE_SHARED, 448, 453,	36
453, 455, 460, 461, 496, 636, 823	37
MPI_WIN_ALLOCATE_SHARED_CPTR,	38
454, 823	39
MPI_WIN_ATTACH, 456, 457, <u>458</u> , 458, 459,	40
496	41
MPI_WIN_C2F, <u>671</u>	42
MPI_WIN_CALL_ERRHANDLER, <u>397</u> , 398	
MPI_WIN_COMPLETE, 460, 485, <u>489</u> ,	43
490-492, 501, 508	44
MPI_WIN_CREATE, 448, <u>449</u> , 451–454, 457,	45
458, 460, 461, 500, 535	46
MPI_WIN_CREATE_DYNAMIC, 393, 448,	47
456, 456-458, 460, 461, 501	48

1	MPI_WIN_CREATE_ERRHANDLER, 384,
2	386, 387, 702, 704, 829
3	MPI_WIN_CREATE_KEYVAL, 298, <u>304</u> , 311,
4	677, 701, 703, 834
	MPI_WIN_DELETE_ATTR, 298, <u>307</u> , 311
5	MPI_WIN_DETACH, 456, 459, 459, 460
6	MPI_WIN_DUP_FN, <u>305</u> , <u>305</u> , <u>693</u> , <u>824</u>
7	MPI_WIN_F2C, <u>671</u>
8	MPI_WIN_FENCE, 460, 468, 484, <u>486</u> , 487,
9	488, 498, 499, 501, 502, 505, 510, 662
10	MPI_WIN_FLUSH, 455, 477, 479, 496, 497,
11	501, 515, 517
12	MPI_WIN_FLUSH_ALL, 477, 479, 497, 501
	MPI_WIN_FLUSH_LOCAL, 477, 497, 501
13	MPI_WIN_FLUSH_LOCAL_ALL, 477, 498,
14	498, 501
15	MPI_WIN_FREE, 306, 402, 443, <u>459</u> , 460
16	MPI_WIN_FREE_KEYVAL, 298, <u>306</u> , 311
17	MPI_WIN_GET_ATTR, 298, <u>307</u> , <u>311</u> , 460,
18	461, 678, 681
19	MPI_WIN_GET_ERRHANDLER, 384, 388,
20	835
	MPI_WIN_GET_GROUP, <u>461</u> , <u>462</u>
21	MPI_WIN_GET_INFO, 462, 463, 463, 822,
22	827
23	MPI_WIN_GET_NAME, <u>318</u>
24	MPI_WIN_LOCK, 450, 460, 485, <u>493</u> ,
25	494-496, 498, 500, 502, 505-507
26	MPI_WIN_LOCK_ALL, 450, 485, <u>494</u> , 494,
27	495, 498, 500, 502, 507, 515, 516
28	MPI_WIN_NULL_COPY_FN, <u>305</u> , 305, 693,
29	824
	$MPI_WIN_NULL_DELETE_FN, \underline{305}, \underline{693},$
30	824
31	MPI_WIN_POST, 460, 485, 489, <u>490</u> , 490–493,
32	495, 498, 499, 501, 508, 510
33	MPI_WIN_SET_ATTR, 298, <u>306</u> , 311, 460,
34	461,630,678,681
35	MPI_WIN_SET_ERRHANDLER, 384, <u>387</u>
36	MPI_WIN_SET_INFO, <u>462</u> , 462, 463, 822, 827
37	MPI_WIN_SET_NAME, <u>317</u>
	MPI_WIN_SHARED_QUERY, 453, <u>455</u> , 636,
38	823
39	MPI_WIN_SHARED_QUERY_CPTR, 456,
40	823
41	MPI_WIN_START, 460, 485, <u>489</u> , 489–493,
42	498, 499, 508, 515
43	MPI_WIN_SYNC, <u>498</u> , 498, 501–504, 508,
44	515-517
45	MPI_WIN_TEST, <u>492</u> , 492
	MPI_WIN_UNLOCK, 460, 479, 485, <u>494</u> , 496,
46	501, 502, 505, 506
47	MPI_WIN_UNLOCK_ALL, 479, 485, 494,
48	495, 501, 502, 505, 516

 $\begin{array}{l} \text{MPI_WIN_WAIT, 460, 485, \underline{491}, 491, 492,} \\ & 495, 501, 502, 505, 508, 509 \\ \text{MPI_WTICK, 20, } \underline{399}, 399 \\ \text{MPI_WTIME, 20, 378, } \underline{398}, 398, 399 \\ \text{mpiexec, } 400, 405, 406, \underline{407}, 536, 537 \\ \text{mpirun, } 406 \\ \end{array}$

PMPI_, 630 PMPI_AINT_ADD, 20 PMPI_AINT_DIFF, 20 PMPI_ISEND, 630, 633 PMPI_WTICK, 20 PMPI_WTIME, 20