

MPI: A Message-Passing Interface Standard

Message Passing Interface Forum

June 12, 1995

This work was supported in part by ARPA and NSF under grant ASC-9310330, the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and by the Commission of the European Community through Esprit project P6643.

1 **Version 1.1: June, 1995.** Beginning in March, 1995, the Message Passing Interface Forum
2 reconvened to correct errors and make clarifications in the MPI document of May 5, 1994,
3 referred to below as Version 1.0. These discussions resulted in Version 1.1, which is this
4 document. The changes from Version 1.0 are minor. A version of this document with all
5 changes marked is available. This paragraph is an example of a change.

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

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

14 This is the final report, Version 1.0, of the Message Passing Interface Forum. This
15 document contains all the technical features proposed for the interface. This copy of the
16 draft was processed by L^AT_EX on June 12, 1995.

17 Please send comments on MPI to `mpi-comments@cs.utk.edu`. Your comment will be
18 forwarded to MPIF committee members who will attempt to respond.

19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40 ©1993, 1994, 1995 University of Tennessee, Knoxville, Tennessee. Permission to copy
41 without fee all or part of this material is granted, provided the University of Tennessee
42 copyright notice and the title of this document appear, and notice is given that copying is
43 by permission of the University of Tennessee.
44
45
46
47
48

Contents

		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
Acknowledgments	vi	11
		12
1 Introduction to MPI	1	13
1.1 Overview and Goals	1	14
1.2 Who Should Use This Standard?	2	15
1.3 What Platforms Are Targets For Implementation?	3	16
1.4 What Is Included In The Standard?	3	17
1.5 What Is Not Included In The Standard?	3	18
1.6 Organization of this Document	4	19
		20
2 MPI Terms and Conventions	6	21
2.1 Document Notation	6	22
2.2 Procedure Specification	6	23
2.3 Semantic Terms	7	24
2.4 Data Types	8	25
2.4.1 Opaque objects	8	26
2.4.2 Array arguments	9	27
2.4.3 State	9	28
2.4.4 Named constants	9	29
2.4.5 Choice	10	30
2.4.6 Addresses	10	31
2.5 Language Binding	10	32
2.5.1 Fortran 77 Binding Issues	10	33
2.5.2 C Binding Issues	11	34
2.6 Processes	12	35
2.7 Error Handling	12	36
2.8 Implementation issues	14	37
2.8.1 Independence of Basic Runtime Routines	14	38
2.8.2 Interaction with signals in POSIX	14	39
2.9 Examples	15	40
		41
3 Point-to-Point Communication	16	42
3.1 Introduction	16	43
3.2 Blocking Send and Receive Operations	17	44
3.2.1 Blocking send	17	45
3.2.2 Message data	17	46
3.2.3 Message envelope	19	47
3.2.4 Blocking receive	20	48

1	3.2.5	Return status	21
2	3.3	Data type matching and data conversion	23
3	3.3.1	Type matching rules	23
4	3.3.2	Data conversion	25
5	3.4	Communication Modes	27
6	3.5	Semantics of point-to-point communication	30
7	3.6	Buffer allocation and usage	34
8	3.6.1	Model implementation of buffered mode	36
9	3.7	Nonblocking communication	36
10	3.7.1	Communication Objects	38
11	3.7.2	Communication initiation	38
12	3.7.3	Communication Completion	41
13	3.7.4	Semantics of Nonblocking Communications	44
14	3.7.5	Multiple Completions	45
15	3.8	Probe and Cancel	51
16	3.9	Persistent communication requests	55
17	3.10	Send-receive	59
18	3.11	Null processes	60
19	3.12	Derived datatypes	61
20	3.12.1	Datatype constructors	62
21	3.12.2	Address and extent functions	69
22	3.12.3	Lower-bound and upper-bound markers	70
23	3.12.4	Commit and free	72
24	3.12.5	Use of general datatypes in communication	73
25	3.12.6	Correct use of addresses	76
26	3.12.7	Examples	77
27	3.13	Pack and unpack	85
28			
29	4	Collective Communication	91
30	4.1	Introduction and Overview	91
31	4.2	Communicator argument	94
32	4.3	Barrier synchronization	94
33	4.4	Broadcast	94
34	4.4.1	Example using <code>MPI_BCAST</code>	95
35	4.5	Gather	95
36	4.5.1	Examples using <code>MPI_GATHER</code> , <code>MPI_GATHERV</code>	98
37	4.6	Scatter	104
38	4.6.1	Examples using <code>MPI_SCATTER</code> , <code>MPI_SCATTERV</code>	106
39	4.7	Gather-to-all	109
40	4.7.1	Examples using <code>MPI_ALLGATHER</code> , <code>MPI_ALLGATHERV</code>	110
41	4.8	All-to-All Scatter/Gather	111
42	4.9	Global Reduction Operations	113
43	4.9.1	Reduce	113
44	4.9.2	Predefined reduce operations	114
45	4.9.3	<code>MINLOC</code> and <code>MAXLOC</code>	116
46	4.9.4	User-Defined Operations	120
47	4.9.5	All-Reduce	124
48	4.10	Reduce-Scatter	125

4.11	Scan	126	1
4.11.1	Example using <code>MPL_SCAN</code>	126	2
4.12	Correctness	128	3
			4
5	Groups, Contexts, and Communicators	132	5
5.1	Introduction	132	6
5.1.1	Features Needed to Support Libraries	132	7
5.1.2	MPI's Support for Libraries	133	8
5.2	Basic Concepts	135	9
5.2.1	Groups	135	10
5.2.2	Contexts	135	11
5.2.3	Intra-Communicators	136	12
5.2.4	Predefined Intra-Communicators	136	13
5.3	Group Management	137	14
5.3.1	Group Accessors	137	15
5.3.2	Group Constructors	138	16
5.3.3	Group Destructors	142	17
5.4	Communicator Management	143	18
5.4.1	Communicator Accessors	143	19
5.4.2	Communicator Constructors	144	20
5.4.3	Communicator Destructors	147	21
5.5	Motivating Examples	148	22
5.5.1	Current Practice #1	148	23
5.5.2	Current Practice #2	149	24
5.5.3	(Approximate) Current Practice #3	149	25
5.5.4	Example #4	150	26
5.5.5	Library Example #1	151	27
5.5.6	Library Example #2	153	28
5.6	Inter-Communication	155	29
5.6.1	Inter-communicator Accessors	157	30
5.6.2	Inter-communicator Operations	158	31
5.6.3	Inter-Communication Examples	160	32
5.7	Caching	167	33
5.7.1	Functionality	167	34
5.7.2	Attributes Example	172	35
5.8	Formalizing the Loosely Synchronous Model	174	36
5.8.1	Basic Statements	174	37
5.8.2	Models of Execution	174	38
			39
6	Process Topologies	176	40
6.1	Introduction	176	41
6.2	Virtual Topologies	177	42
6.3	Embedding in MPI	177	43
6.4	Overview of the Functions	178	44
6.5	Topology Constructors	179	45
6.5.1	Cartesian Constructor	179	46
6.5.2	Cartesian Convenience Function: <code>MPI_DIMS_CREATE</code>	179	47
6.5.3	General (Graph) Constructor	180	48

1	6.5.4	Topology inquiry functions	182
2	6.5.5	Cartesian Shift Coordinates	186
3	6.5.6	Partitioning of Cartesian structures	187
4	6.5.7	Low-level topology functions	188
5	6.6	An Application Example	189
6			
7	7	MPI Environmental Management	191
8	7.1	Implementation information	191
9	7.1.1	Environmental Inquiries	191
10	7.2	Error handling	193
11	7.3	Error codes and classes	196
12	7.4	Timers and synchronization	198
13	7.5	Startup	198
14			
15	8	Profiling Interface	201
16	8.1	Requirements	201
17	8.2	Discussion	201
18	8.3	Logic of the design	202
19	8.3.1	Miscellaneous control of profiling	202
20	8.4	Examples	203
21	8.4.1	Profiler implementation	203
22	8.4.2	MPI library implementation	204
23	8.4.3	Complications	205
24	8.5	Multiple levels of interception	206
25			
26		Bibliography	207
27	A	Language Binding	210
28	A.1	Introduction	210
29	A.2	Defined Constants for C and Fortran	210
30	A.3	C bindings for Point-to-Point Communication	214
31	A.4	C Bindings for Collective Communication	217
32	A.5	C Bindings for Groups, Contexts, and Communicators	218
33	A.6	C Bindings for Process Topologies	219
34	A.7	C bindings for Environmental Inquiry	220
35	A.8	C Bindings for Profiling	220
36	A.9	Fortran Bindings for Point-to-Point Communication	220
37	A.10	Fortran Bindings for Collective Communication	224
38	A.11	Fortran Bindings for Groups, Contexts, etc.	226
39	A.12	Fortran Bindings for Process Topologies	227
40	A.13	Fortran Bindings for Environmental Inquiry	228
41	A.14	Fortran Bindings for Profiling	229
42			
43		MPI Function Index	230
44			
45			
46			
47			
48			

Acknowledgments

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 served in positions of responsibility and are listed below.

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

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

Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz
Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster
Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower
Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison
Leslie Hart	Tom Haupt	Don Heller	Tom Henderson
Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga
James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe
Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley
Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer
Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison
Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson
Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith

The University of Tennessee and Oak Ridge National Laboratory made the draft available by anonymous FTP mail servers and were instrumental in distributing the document.

MPI operated on a very tight budget (in reality, it had no budget when the first meeting was announced). ARPA and NSF have supported research at various institutions that have

made a contribution towards travel for the U.S. academics. Support for several European participants was provided by ESPRIT.

Chapter 1

Introduction to MPI

1.1 Overview and Goals

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. Each vendor has implemented its own variant. More recently, several systems have demonstrated that a message passing system can be efficiently and portably implemented. It is thus an appropriate time to try to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.

In designing MPI we have 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 has been strongly influenced by work at the IBM T. J. Watson Research Center [1, 2], Intel's NX/2 [23], Express [22], nCUBE's Vertex [21], p4 [7, 6], and PARMACS [5, 8]. Other important contributions have come from Zipcode [24, 25], Chimp [14, 15], PVM [4, 11], Chameleon [19], and PICL [18].

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

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [12]. MPI1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI1 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 which it was decided to place the standardization process on a more formal footing, and to

generally adopt the procedures and organization of the High Performance Fortran Forum. Subcommittees were formed for the major component areas of the standard, and an email discussion service established for each. In addition, the goal of producing a draft MPI standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI standard at the Supercomputing 93 conference in November 1993. These meetings and the email discussion together constituted the MPI Forum, membership of which has been open to all members of the high performance computing community.

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 build upon lower level message passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing 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 provide hardware support for, 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 and allow overlap of computation and communication and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for the interface.
- 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 is not too different from current practice, such as PVM, NX, Express, p4, etc., and provides extensions that allow greater flexibility.
- 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 Who Should Use This Standard?

This standard is intended for use by all those who want to write portable message-passing programs in Fortran 77 and 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.

1.3 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 are possible. The paradigm will not be made obsolete by architectures combining the shared- and 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. Although no explicit support for threads is provided, the interface has been designed so as not to prejudice their use. With this version of MPI no support is provided for dynamic spawning of tasks.

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. Several proprietary, native implementations of MPI, and a public domain, portable implementation of MPI are in progress at the time of this writing [17, 13].

1.4 What Is Included In The Standard?

The standard includes:

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Bindings for Fortran 77 and C
- Environmental Management and inquiry
- Profiling interface

1.5 What Is Not Included In The Standard?

The standard does not specify:

- Explicit shared-memory operations
- 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
- Explicit support for threads
- Support for task management
- I/O functions

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

1.6 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, **Collective Communications**, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes).
- Chapter 5, **Groups, Contexts, and Communicators**, 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 6, **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 7, **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 8, **Profiling Interface**, explains a simple name-shifting convention that any MPI implementation must support. One motivation for this is the ability to put performance profiling calls into MPI without the need for access to the MPI source code. The name shift is merely an interface, it says nothing about how the actual profiling should be done and in fact, the name shift can be useful for other purposes.

- Annex A, **Language Bindings**, gives specific syntax in Fortran 77 and C, for all MPI functions, constants, and types.
- The **MPI Function Index** is a simple index showing the location of the precise definition of each MPI function, together with both C and Fortran bindings.

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 that speaks to users and 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 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:

- the call uses but does not update an argument marked IN,
- the call may update an argument marked OUT,
- the call both uses and updates an argument marked INOUT.

There is one special case — if an argument is a handle to an opaque object (these terms are defined in Section 2.4.1), and the object is updated by the procedure call, then the argument is marked OUT. It is marked this way even though the handle itself is not modified — we use the OUT attribute to denote that what the handle *references* is updated.

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.

A common occurrence for MPI functions is an argument that is used as IN by some processes and OUT by other processes. Such 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.

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 argument.

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

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

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

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

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

All MPI functions are first specified in the language-independent notation. Immediately below this, the ANSI C version of the function is shown, and below this, a version of the same function in Fortran 77.

2.3 Semantic Terms

When discussing MPI procedures the following semantic terms are used. The first two are usually applied to communication operations.

nonblocking If the procedure may return before the operation completes, and before the user is allowed to re-use resources (such as buffers) specified in the call.

blocking If return from the procedure indicates the user is allowed to re-use resources specified in the call.

local If completion of the procedure depends only on the local executing process. Such an operation does not require communication with another user process.

non-local If completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication occurring with another user process.

collective If all processes in a process group need to invoke the procedure.

2.4 Data Types

2.4.1 Opaque objects

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

In Fortran, all handles have type **INTEGER**. In C, a different handle type is defined for each category of objects. These should be types that support assignment and equality operators.

In Fortran, the handle can be an index to a table of opaque objects in system table; in C it can be such index or a pointer to the object. More bizarre possibilities exist.

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 a “null handle” value. MPI provides a “null handle” constant for each object type. Comparisons to this constant are used to test for validity of the handle.

A call to deallocate 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.

MPI calls do not change the value of handles, with the exception of calls that allocate and deallocate objects, and of the call **MPI_TYPE_COMMIT**, in Section 3.12.4.

A null handle argument is an erroneous IN argument in MPI calls, unless an exception is explicitly stated in the text that defines the function. Such exception is allowed for handles to request objects in Wait and Test calls (sections 3.7.3 and 3.7.5). Otherwise, a null handle can only be passed to a function that allocates a new object and returns a reference to it in the handle.

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. Such objects may not be destroyed.

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 separating of handles in user space, objects in system space, allows space-reclaiming, deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed. The specified design allows an object to be marked for deallocation, the user program can

then go out of scope, and the object itself still persists until any pending operations are complete.

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. The alternative would have been to allow handles to have been an arbitrary, opaque type. This would force the introduction of routines to do assignment and comparison, adding complexity, and was therefore ruled out. (*End of rationale.*)

Advice to users. A user may accidentally create a dangling reference by assigning to a handle the value of another handle, and then deallocating the object associated with these handles. Conversely, if a handle variable is deallocated before the associated object is freed, then the object becomes inaccessible (this may occur, for example, if the handle is a local variable within a subroutine, and the subroutine is exited before the associated object is deallocated). It is the user's responsibility to avoid adding or deleting references to opaque objects, except as a result of calls that allocate or deallocate such objects. (*End of advice to users.*)

Advice to implementors. The intended semantics of opaque objects is that each opaque object is separate from each other; each call to allocate such an object copies all the information required for the object. Implementations may avoid excessive copying by substituting referencing for copying. For example, a derived datatype may contain references to its components, rather than copies of its components; a call to `MPI_COMM_GROUP` may return a reference to the group associated with the communicator, rather than a copy of this group. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects such that the visible effect is as if the objects were copied. (*End of advice to implementors.*)

2.4.2 Array arguments

An MPI call may need an argument that is an array of opaque objects, or an array of handles. The array-of-handles is a regular array with entries that are handles to objects of the same type in consecutive locations in the array. Whenever such an array is used, an additional `len` argument is required to indicate the number of valid entries (unless this number can be derived otherwise). The valid entries are at the beginning of the array; `len` indicates how many of them there are, and need not be the entire size of the array. The same approach is followed for other array arguments.

2.4.3 State

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

2.4.4 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 can be queried using environmental inquiry functions (Chapter 7).

MPI also provides predefined named constant handles, such as `MPI_COMM_WORLD` which is a handle to an object that represents all processes available at start-up time and allowed to communicate with any of them.

All named constants, with the exception of `MPI_BOTTOM` in Fortran, can be used in initialization expressions or assignments. These constants do not change values during execution. Opaque objects accessed by constant handles are defined and do not change value between MPI initialization (`MPI_INIT()` call) and MPI completion (`MPI_FINALIZE()` call).

2.4.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran, the document uses `<type>` to represent a choice variable, for C, we use `(void *)`.

2.4.6 Addresses

Some MPI procedures use *address* arguments that represent an absolute address in the calling program. The datatype of such an argument is an integer of the size needed to hold any valid address in the execution environment.

2.5 Language Binding

This section defines the rules for MPI language binding in general and for Fortran 77 and ANSI C in particular. Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

It is expected that any Fortran 90 and C++ implementations use the Fortran 77 and ANSI C bindings, respectively. Although we consider it premature to define other bindings to Fortran 90 and C++, the current bindings are designed to encourage, rather than discourage, experimentation with better bindings that might be adopted later.

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.

There are several important language binding issues not addressed by this standard. This standard does not discuss the interoperability of message passing between languages. It is fully expected that many implementations will have such features, and that such features are a sign of the quality of the implementation.

2.5.1 Fortran 77 Binding Issues

All MPI names have an `MPI_` prefix, and all characters are capitals. Programs must not declare variables or functions with names beginning with the prefix, `MPI_`. This is mandated

```

1      double precision a
2      integer b
3      ...
4      call MPI_send(a,...)
5      call MPI_send(b,...)
6
7

```

Figure 2.1: An example of calling a routine with mismatched formal and actual arguments.

to avoid possible name collisions.

All MPI Fortran subroutines have a return code in the last argument. A few MPI operations are functions, which do not have the return code argument. The return code value for successful completion is `MPI_SUCCESS`. Other error codes are implementation dependent; see Chapter 7.

Handles are represented in Fortran as `INTEGERS`. Binary-valued variables are of type `LOGICAL`.

Array arguments are indexed from one.

Unless explicitly stated, the MPI F77 binding is consistent with ANSI standard Fortran 77. There are several points where this standard diverges from the ANSI Fortran 77 standard. These exceptions are consistent with common practice in the Fortran community. In particular:

- MPI identifiers are limited to thirty, not six, significant characters.
- MPI identifiers may contain underscores after the first character.
- An MPI subroutine with a choice argument may be called with different argument types. An example is shown in Figure 2.1. This violates the letter of the Fortran standard, but such a violation is common practice. An alternative would be to have a separate version of `MPI_SEND` for each data type.
- Although not required, it is strongly suggested that named MPI constants (`PARAMETERS`) be provided in an include file, called `mpif.h`. On systems that do not support include files, the implementation should specify the values of named constants.
- Vendors are encouraged to provide type declarations in the `mpif.h` file on Fortran systems that support user-defined types. One should define, if possible, the type `MPI_ADDRESS`, which is an `INTEGER` of the size needed to hold an address in the execution environment. On systems where type definition is not supported, it is up to the user to use an `INTEGER` of the right kind to represent addresses (i.e., `INTEGER*4` on a 32 bit machine, `INTEGER*8` on a 64 bit machine, etc.).

All MPI named constants can be used wherever an entity declared with the `PARAMETER` attribute can be used in Fortran. There is one exception to this rule: the MPI constant `MPI_BOTTOM` (section 3.12.2) can only be used as a buffer argument.

2.5.2 C Binding Issues

We use the ANSI C declaration format. All MPI names have an `MPI_` prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after

the prefix. Programs must not declare variables or functions with names beginning with the prefix, `MPI_`. This is mandated to avoid possible name collisions.

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. A few C functions do not return values, so that they can be implemented as macros.

Type declarations are provided for handles to each category of opaque objects. Either a pointer or an integer type is used.

Array arguments are indexed from zero.

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

Choice arguments are pointers of type `void*`.

Address arguments are of MPI defined type `MPI_Aint`. This is defined to be an int of the size needed to hold any valid address on the target architecture.

All named MPI constants can be used in initialization expressions or assignments like C constants.

2.6 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible. This document specifies the behavior of a parallel program assuming that only MPI calls are used for communication. The interaction of an MPI program with other possible means of communication (e.g., shared memory) is not specified.

MPI does not specify the execution model for each process. A process can be sequential, or can be multi-threaded, with threads possibly executing concurrently. Care has been taken to make MPI “thread-safe,” by avoiding the use of implicit state. The desired interaction of MPI with threads is that concurrent threads be all allowed to execute MPI calls, and calls be reentrant; a blocking MPI call blocks only the invoking thread, allowing the scheduling of another thread.

MPI does not provide mechanisms to specify the initial allocation of processes to an MPI computation and their binding to physical processors. It is expected that vendors will provide mechanisms to do so either at load time or at run time. Such mechanisms will allow the specification of the initial number of required processes, the code to be executed by each initial process, and the allocation of processes to processors. Also, the current proposal does not provide for dynamic creation or deletion of processes during program execution (the total number of processes is fixed), although it is intended to be consistent with such extensions. Finally, we always identify processes according to their relative rank in a group, that is, consecutive integers in the range `0..groupsize-1`.

2.7 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other

error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures. The error handling facilities described in section 7.2 can be used to restrict the scope of an unrecoverable error, or design error recovery at the application level.

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

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. In certain circumstances, when the MPI function may complete several distinct operations, and therefore may generate several independent errors, the MPI function may return multiple error codes. By default, an error detected during the execution of the MPI library causes the parallel computation to abort. 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 7.2.

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

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

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

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

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 nonconforming behavior.

2.8 Implementation issues

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

2.8.1 Independence of Basic Runtime Routines

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

Note that this in no way prevents the creation of library routines that provide parallel services whose operation is collective. However, the following program is expected to complete in an ANSI C environment regardless of the size of `MPI_COMM_WORLD` (assuming that I/O is available at the executing nodes).

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

The corresponding Fortran 77 program is 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).

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

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

2.8.2 Interaction with signals in POSIX

MPI does not specify either the interaction of processes with signals, in a UNIX environment, or with other events that do not relate to MPI communication. That is, signals are not significant from the view point of MPI, and implementors should attempt to implement MPI so that signals are transparent: an MPI call suspended by a signal should resume and

complete after the signal is handled. Generally, the state of a computation that is visible or significant from the view-point of MPI should only be affected by MPI calls.

The intent of MPI to be thread and signal safe has a number of subtle effects. For example, on Unix systems, a catchable signal such as SIGALRM (an alarm signal) must not cause an MPI routine to behave differently than it would have in the absence of the signal. Of course, if the signal handler issues MPI calls or changes the environment in which the MPI routine is operating (for example, consuming all available memory space), the MPI routine should behave as appropriate for that situation (in particular, in this case, the behavior should be the same as for a multithreaded MPI implementation).

A second effect is that a signal handler that performs MPI calls must not interfere with the operation of MPI. For example, an MPI receive of any type that occurs within a signal handler must not cause erroneous behavior by the MPI implementation. Note that an implementation is permitted to prohibit the use of MPI calls from within a signal handler, and is not required to detect such use.

It is highly desirable that MPI not use SIGALRM, SIGFPE, or SIGIO. An implementation is *required* to clearly document all of the signals that the MPI implementation uses; a good place for this information is a Unix ‘man’ page on MPI.

2.9 Examples

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

Chapter 3

Point-to-Point Communication

3.1 Introduction

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.

```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)    /* code for process zero */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else                /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

In 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 from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable **message** in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition,

the send operation associates an **envelope** with the message. This envelope specifies the message destination and contains distinguishing information that can be used by the **receive** operation to select a particular message. The last three parameters of the send operation specify the envelope for the message sent.

Process one (**myrank** = 1) receives this message with the **receive** operation **MPI_RECV**. The message to be received is selected according to the value of its envelope, and the message data is stored into the **receive buffer**. In the example above, the receive buffer consists of the storage containing the string **message** in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.

The next sections describe the blocking send and receive operations. We discuss send, receive, blocking communication semantics, type matching requirements, type conversion in heterogeneous environments, and more general communication modes. Nonblocking communication is addressed next, followed by channel-like constructs and send-receive operations. We then 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.

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)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (nonnegative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

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

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

```
<type> BUF(*)
```

```
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

The blocking semantics of this call are described in Sec. 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 below.

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	

Possible values for this argument for C and the corresponding C types are listed below.

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

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 3.13.

MPI requires support of the datatypes listed above, which match the basic datatypes of Fortran 77 and ANSI C. Additional MPI datatypes should be provided if the host language

has additional data types: `MPI_LONG_LONG_INT`, for (64 bit) C integers declared to be of type `longlong int`; `MPI_DOUBLE_COMPLEX` for double precision complex in Fortran declared to be of type `DOUBLE COMPLEX`; `MPI_REAL2`, `MPI_REAL4` and `MPI_REAL8` for Fortran reals, declared to be of type `REAL*2`, `REAL*4` and `REAL*8`, respectively; `MPI_INTEGER1`, `MPI_INTEGER2` and `MPI_INTEGER4` for Fortran integers, declared to be of type `INTEGER*1`, `INTEGER*2` and `INTEGER*4`, respectively; etc.

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

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 7. 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 5; below is a brief summary of their usage.

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

The communicator also specifies the set of processes that share this communication context. This **process group** is ordered and processes are identified by their rank within this group. Thus, the range of valid values for **dest** is `0, ... , n-1`, where `n` is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 5.)

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.

Users may define new communicators, as explained in Chapter 5. Communicators provide an important encapsulation mechanism for libraries and modules. They allow modules to have their own disjoint communication universe and their own process 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.*)

3.2.4 Blocking receive

The syntax of the blocking receive operation is given below.

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

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
IERROR
```

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

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

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

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

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

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

The selection of a message by a receive operation is governed by the value of the message envelope. A message can be received by a receive operation if its envelope matches the **source**, **tag** and **comm** values specified by the receive operation. The receiver may specify a wildcard `MPI_ANY_SOURCE` value for **source**, and/or a wildcard `MPI_ANY_TAG` value for **tag**, indicating that any source and/or tag are acceptable. It cannot specify a 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 $\{0, \dots, n-1\} \cup \{\text{MPI_ANY_SOURCE}\}$, where n is the number of processes in this group.

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

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

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 MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, **status** is a structure that contains three fields named **MPI_SOURCE**, **MPI_TAG**, and **MPI_ERROR**; the structure may contain additional fields. Thus, **status.MPI_SOURCE**, **status.MPI_TAG** and **status.MPI_ERROR** contain the source, tag, and error code, respectively, of the received message.

In Fortran, **status** is an array of **INTEGER**s 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.

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 (integer)

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

```
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. (We shall later see, in Section 3.12.5, that **MPI_GET_COUNT** may return, in certain situations, the value **MPI_UNDEFINED**.)

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

The **datatype** argument is passed to **MPI_GET_COUNT** so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to **MPI_PROBE**. (*End of rationale.*)

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

3.3 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 Sec. 3.13, 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 Sec. 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 3.13. The type `MPI_BYTE` allows one to transfer the binary value of a byte in memory unchanged.

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

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

The following examples illustrate the first two cases.

Example 3.1 Sender and receiver specify matching types.

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

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

Example 3.2 Sender and receiver do not specify matching types.

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

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

Example 3.3 Sender and receiver specify communication of untyped values.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is correct, irrespective of the type and size of **a** and **b** (unless this results in an out of bound memory access).

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.*)

1 Type MPI_CHARACTER

2 The type MPI_CHARACTER matches one character of a Fortran variable of type CHARACTER,
3 rather than the entire character string stored in the variable. Fortran variables of type
4 CHARACTER or substrings are transferred as if they were arrays of characters. This is
5 illustrated in the example below.
6

7 **Example 3.4** Transfer of Fortran CHARACTERs.

```
8
9 CHARACTER*10 a
10 CHARACTER*10 b
11
12 CALL MPI_COMM_RANK(comm, rank, ierr)
13 IF(rank.EQ.0) THEN
14     CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
15 ELSE
16     CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
17 END IF
18
```

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

21
22 *Rationale.* The alternative choice would be for MPI_CHARACTER to match a char-
23 acter of arbitrary length. This runs into problems.

24 A Fortran character variable is a constant length string, with no special termination
25 symbol. There is no fixed convention on how to represent characters, and how to store
26 their length. Some compilers pass a character argument to a routine as a pair of argu-
27 ments, one holding the address of the string and the other holding the length of string.
28 Consider the case of an MPI communication call that is passed a communication buffer
29 with type defined by a derived datatype (Section 3.12). If this communicator buffer
30 contains variables of type CHARACTER then the information on their length will not be
31 passed to the MPI routine.

32 This problem forces us to provide explicit information on character length with the
33 MPI call. One could add a length parameter to the type MPI_CHARACTER, but this
34 does not add much convenience and the same functionality can be achieved by defining
35 a suitable derived datatype. (*End of rationale.*)
36

37 *Advice to implementors.* Some compilers pass Fortran CHARACTER arguments as a
38 structure with a length and a pointer to the actual string. In such an environment,
39 the MPI call needs to dereference the pointer in order to reach the string. (*End of*
40 *advice to implementors.*)
41

42 3.3.2 Data conversion

43
44 One of the goals of MPI is to support parallel computations across heterogeneous environ-
45 nments. Communication in a heterogeneous environment may require data conversions. We
46 use the following terminology.

47 **type conversion** changes the datatype of a value, e.g., by rounding a REAL to an INTEGER.
48

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

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

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

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

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

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

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

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 does not require support for inter-language communication. The behavior of a program is undefined if messages are sent by a C process and received by a Fortran process, or vice-versa.

Rationale. MPI does not handle inter-language communication because there are no agreed standards for the correspondence between C types and Fortran types. Therefore, MPI programs that mix languages would not port. (*End of rationale.*)

Advice to implementors. MPI implementors may want to support inter-language communication by allowing Fortran programs to use “C MPI types,” such as `MPLINT`, `MPLCHAR`, etc., and allowing C programs to use Fortran types. (*End of advice to implementors.*)

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

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

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

There are three additional communication modes.

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

A send that uses the **synchronous** mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is **non-local**.

A send that uses the **ready** communication mode may be started *only* if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation that is otherwise required 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 can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

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.

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

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

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

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*)
```

```
1      INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
2      Send in buffered mode.
```

```
3
```

```
4
```

```
5
```

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

```
6
```

```
IN      buf                      initial address of send buffer (choice)
```

```
7
```

```
IN      count                    number of elements in send buffer (integer)
```

```
8
```

```
IN      datatype                 datatype of each send buffer element (handle)
```

```
9
```

```
IN      dest                     rank of destination (integer)
```

```
10
```

```
IN      tag                      message tag (integer)
```

```
11
```

```
IN      comm                     communicator (handle)
```

```
12
```

```
13
```

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
16      int tag, MPI_Comm comm)
```

```
17
```

```
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

```
18
```

```
<type> BUF(*)
```

```
19
```

```
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
20
```

```
Send in synchronous mode.
```

```
21
```

```
22
```

```
23
```

```
24
```

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

```
25
```

```
IN      buf                      initial address of send buffer (choice)
```

```
26
```

```
IN      count                    number of elements in send buffer (integer)
```

```
27
```

```
IN      datatype                 datatype of each send buffer element (handle)
```

```
28
```

```
IN      dest                     rank of destination (integer)
```

```
29
```

```
IN      tag                      message tag (integer)
```

```
30
```

```
IN      comm                     communicator (handle)
```

```
31
```

```
32
```

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
35      int tag, MPI_Comm comm)
```

```
36
```

```
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

```
37
```

```
<type> BUF(*)
```

```
38
```

```
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
39
```

```
Send in ready mode.
```

```
40
```

```
41
```

There is only one receive operation, which can match any of the send modes. The receive operation described in the last section is **blocking**: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

```
42
```

```
43
```

```
44
```

In a multi-threaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. In such a case it is the user's responsibility not to access or modify a

```
45
```

```
46
```

```
47
```

```
48
```

communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.

Rationale. We prohibit read accesses to a send buffer while it is being used, even though the send operation is not supposed to alter the content of this buffer. This may seem more stringent than necessary, but the additional restriction causes little loss of functionality and allows better performance on some systems — consider the case where data transfer is done by a DMA engine that is not cache-coherent with the main processor. (*End of rationale.*)

Advice to implementors. Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation.

It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal his or her preference for blocking the sender until a matching receive occurs by using the synchronous send mode.

A possible communication protocol for the various communication modes is outlined below.

ready send: The message is sent as soon as possible.

synchronous send: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.

standard send: First protocol may be used for short messages, and second protocol for long messages.

buffered send: The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).

Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.

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.

A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, many (most?) users expect some buffering.

In a multi-threaded 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. (*End of advice to implementors.*)

3.5 Semantics of point-to-point communication

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

Order Messages are *non-overtaking*: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of

sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard `MPI_ANY_SOURCE` is not used in receives. (Some of the calls described later, such as `MPI_CANCEL` or `MPI_WAITANY`, are additional sources of nondeterminism.)

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

Example 3.5 An example of non-overtaking messages.

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

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

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

Example 3.6 An example of two, intertwined matching pairs.

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

Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At that point, a matching pair of send and receive operation is enabled, and both operations

must complete. Process one next invokes its second receive call, which will be satisfied by the buffered message. Note that process one received the messages in the reverse order they were sent.

Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multi-threaded process. Then it is possible that messages that match this 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 signalled 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
    ! rank.EQ.1
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
```



```

1      CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
2  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 attempt to exchange messages.

```

8  CALL MPI_COMM_RANK(comm, rank, ierr)
9  IF (rank.EQ.0) THEN
10     CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
11     CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
12 ELSE    ! rank.EQ.1
13     CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
14     CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
15 END IF

```

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

Example 3.9 An exchange that relies on buffering.

```

24 CALL MPI_COMM_RANK(comm, rank, ierr)
25 IF (rank.EQ.0) THEN
26     CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
27     CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
28 ELSE    ! rank.EQ.1
29     CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
30     CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
31 END IF

```

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

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

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

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

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

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 (integer)

int MPI_Buffer_attach(void* buffer, int size)

MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)

<type> BUFFER(*)
INTEGER SIZE, IERROR

Provides to MPI a buffer in the user’s memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time.

MPI_BUFFER_DETACH(buffer_addr, size)

OUT	buffer_addr	initial buffer address (choice)
OUT	size	buffer size, in bytes (integer)

int MPI_Buffer_detach(void* buffer_addr, int* size)

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)

<type> BUFFER_ADDR(*)
INTEGER SIZE, IERROR

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This operation will block until all messages currently in the

buffer have been transmitted. Upon return of this function, the user may reuse or deallocate the space taken by the buffer.

Example 3.10 Calls to attach and detach buffers.

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

Advice to users. Even though the C functions `MPI_Buffer_attach` and `MPI_Buffer_detach` both have a first argument of type `void*`, these arguments are used differently: A pointer to the buffer is passed to `MPI_Buffer_attach`; the address of the pointer is passed to `MPI_Buffer_detach`, so that this call can return the pointer value. (*End of advice to users.*)

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

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

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

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

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

3.6.1 Model implementation of buffered mode

The model implementation uses the packing and unpacking functions described in Section 3.13 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 3.13). The MPI constant `MPI_BSEND_OVERHEAD` provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).
- Find the next contiguous empty space of *n* bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; `MPI_PACK` is used to pack data.
- Post nonblocking send (standard mode) for packed data.
- Return

3.7 Nonblocking communication

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

of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

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

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

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

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

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

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

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

Nonblocking sends in the buffered and ready modes have a more limited impact. A nonblocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of nonblocking sends is advantageous in these cases only if data copying can be concurrent 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 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.

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

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

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
```

```
<type> BUF(*)
```

```
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Start a standard mode, nonblocking send.

```
1 MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)
```

```
2     IN      buf                initial address of send buffer (choice)
3     IN      count              number of elements in send buffer (integer)
4     IN      datatype            datatype of each send buffer element (handle)
5     IN      dest                rank of destination (integer)
6     IN      tag                 message tag (integer)
7     IN      comm                communicator (handle)
8     IN      request             communication request (handle)
9
10    OUT     request
```

```
12 int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
13               int tag, MPI_Comm comm, MPI_Request *request)
```

```
15 MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
```

```
16     <type> BUF(*)
```

```
17     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
18     Start a buffered mode, nonblocking send.
```

```
21 MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
```

```
22     IN      buf                initial address of send buffer (choice)
23     IN      count              number of elements in send buffer (integer)
24     IN      datatype            datatype of each send buffer element (handle)
25     IN      dest                rank of destination (integer)
26     IN      tag                 message tag (integer)
27     IN      comm                communicator (handle)
28     IN      request             communication request (handle)
29
30    OUT     request
```

```
32 int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
33               int tag, MPI_Comm comm, MPI_Request *request)
```

```
35 MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
```

```
36     <type> BUF(*)
```

```
37     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
38     Start a synchronous mode, nonblocking send.
```

```
40
41
42
43
44
45
46
47
48
```

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)			1
IN	buf	initial address of send buffer (choice)	2
IN	count	number of elements in send buffer (integer)	3
IN	datatype	datatype of each send buffer element (handle)	4
IN	dest	rank of destination (integer)	5
IN	tag	message tag (integer)	6
IN	comm	communicator (handle)	7
OUT	request	communication request (handle)	8
			9
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,			10
int tag, MPI_Comm comm, MPI_Request *request)			11
			12
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)			13
<type> BUF(*)			14
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR			15
Start a ready mode nonblocking send.			16
			17
			18
			19
			20
MPI_IRECV (buf, count, datatype, source, tag, comm, request)			21
OUT	buf	initial address of receive buffer (choice)	22
IN	count	number of elements in receive buffer (integer)	23
IN	datatype	datatype of each receive buffer element (handle)	24
IN	source	rank of source (integer)	25
IN	tag	message tag (integer)	26
IN	comm	communicator (handle)	27
OUT	request	communication request (handle)	28
			29
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,			30
int tag, MPI_Comm comm, MPI_Request *request)			31
			32
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)			33
<type> BUF(*)			34
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR			35
Start a nonblocking receive.			36
			37
			38
			39
			40
			41
			42
			43
			44
			45
			46
			47
			48

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. A **empty** status is a status which is set to return `tag = MPI_ANY_TAG`, `source = MPI_ANY_SOURCE`, and is also internally configured so that calls to `MPI_GET_COUNT` and `MPI_GET_ELEMENTS` return `count = 0`. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

`MPI_WAIT(request, status)`

INOUT	request	request (handle)
OUT	status	status object (Status)

`int MPI_Wait(MPI_Request *request, MPI_Status *status)`

`MPI_WAIT(REQUEST, STATUS, IERROR)`

`INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR`

A call to `MPI_WAIT` returns when the operation identified by **request** is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to `MPI_WAIT` and the request handle is set to `MPI_REQUEST_NULL`. `MPI_WAIT` is a non-local operation.

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

One is allowed to call `MPI_WAIT` with a null or inactive **request** argument. In this case the operation returns immediately with empty **status**.

Advice to users. Successful return of `MPI_WAIT` after a `MPI_BSEND` 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 Sec. 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 multi-threaded 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)

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

`MPI_TEST(REQUEST, FLAG, STATUS, IERROR)`

LOGICAL FLAG

INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

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 communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is set to `MPI_REQUEST_NULL`. The call returns `flag = false`, otherwise. 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.*)

Rationale. The function `MPI_TEST` returns with `flag = true` exactly in those situations where the function `MPI_WAIT` returns; both functions return in such case the same value in `status`. Thus, a blocking Wait can be easily replaced by a nonblocking Test. (*End of rationale.*)

Example 3.11 Simple usage of nonblocking operations and `MPI_WAIT`.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
```

```
IF(rank.EQ.0) THEN
```

```
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
```

```

1      **** do some computation to mask latency ****
2      CALL MPI_WAIT(request, status, ierr)
3  ELSE
4      CALL MPI_IRecv(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
5      **** do some computation to mask latency ****
6      CALL MPI_WAIT(request, status, ierr)
7  END IF

```

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

MPI_REQUEST_FREE(request)

INOUT request communication request (handle)

```
int MPI_Request_free(MPI_Request *request)
```

```
MPI_REQUEST_FREE(REQUEST, IERROR)
```

```
INTEGER REQUEST, IERROR
```

Mark the request object for deallocation and set **request** to **MPI_REQUEST_NULL**. An ongoing communication that is associated with the request will be allowed to complete. The request will be deallocated only after its completion.

Rationale. The **MPI_REQUEST_FREE** mechanism is provided for reasons of performance and convenience on the sending side. (*End of rationale.*)

Advice to users. Once a request is freed by a call to **MPI_REQUEST_FREE**, it is not possible to check for the successful completion of the associated communication with calls to **MPI_WAIT** or **MPI_TEST**. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user — such an error must be treated as fatal. Questions arise as to how one knows when the operations have completed when using **MPI_REQUEST_FREE**. Depending on the program logic, there may be other ways in which the program knows that certain operations have completed and this makes usage of **MPI_REQUEST_FREE** practical. For example, an active send request could be freed when the logic of the program is such that the receiver sends a reply to the message sent — the arrival of the reply informs the sender that the send has completed and the send buffer can be reused. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (*End of advice to users.*)

Example 3.12 An example using **MPI_REQUEST_FREE**.

```

43 CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
44 IF(rank.EQ.0) THEN
45     DO i=1, n
46         CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, req, ierr)
47         CALL MPI_REQUEST_FREE(req, ierr)
48     END DO

```

```

        CALL MPI_Irecv(inval, 1, MPI_REAL, 1, 0, req, ierr)
        CALL MPI_WAIT(req, status, ierr)
    END DO
ELSE    ! rank.EQ.1
    CALL MPI_Irecv(inval, 1, MPI_REAL, 0, 0, req, ierr)
    CALL MPI_WAIT(req, status)
    DO I=1, n-1
        CALL MPI_Isend(outval, 1, MPI_REAL, 0, 0, req, ierr)
        CALL MPI_REQUEST_FREE(req, ierr)
        CALL MPI_Irecv(inval, 1, MPI_REAL, 0, 0, req, ierr)
        CALL MPI_WAIT(req, status, ierr)
    END DO
    CALL MPI_Isend(outval, 1, MPI_REAL, 0, 0, req, ierr)
    CALL MPI_WAIT(req, status)
END IF

```

3.7.4 Semantics of Nonblocking Communications

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

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

Example 3.13 Message ordering for nonblocking operations.

```

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
    CALL MPI_Isend(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
    CALL MPI_Isend(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE    ! rank.EQ.1
    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)
CALL MPI_WAIT(r2,status)

```

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

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

Example 3.14 An illustration of progress semantics.

```

1  CALL MPI_COMM_RANK(comm, rank, ierr)
2  IF (RANK.EQ.0) THEN
3      CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
4      CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
5  ELSE    ! rank.EQ.1
6      CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
7      CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, ierr)
8      CALL MPI_WAIT(r, status, ierr)
9  END IF

```

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

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

3.7.5 Multiple Completions

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

MPI_WAITANY (count, array_of_requests, index, status)

IN	count	list length (integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	index	index of handle for operation that completed (integer)
OUT	status	status object (Status)

```

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
               MPI_Status *status)

```

```

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 communication. (The array is indexed from zero in C, and from

one in Fortran.) If the request was allocated by a nonblocking communication operation, then it 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 a 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	<code>count</code>	list length (integer)
INOUT	<code>array_of_requests</code>	array of requests (array of handles)
OUT	<code>index</code>	index of operation that completed, or <code>MPI_UNDEFINED</code> if none completed (integer)
OUT	<code>flag</code>	true if one of the operations is complete (logical)
OUT	<code>status</code>	status object (Status)

```
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
               int *flag, MPI_Status *status)
```

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

Tests for completion of either one or none of the operations associated with active handles. In the former case, it returns `flag = true`, returns in `index` the index of this request in the array, and returns in `status` the status of that operation; if the request was allocated by a nonblocking communication call then the request is deallocated and the handle is set to `MPI_REQUEST_NULL`. (The array is indexed from zero in C, and from one in Fortran.) In the latter case (no operation completed), it returns `flag = false`, returns a value of `MPI_UNDEFINED` in `index` and `status` is undefined.

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

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

Rationale. The function `MPI_TESTANY` returns with `flag = true` exactly in those situations where the function `MPI_WAITANY` returns; both functions return in that case the same values in the remaining parameters. Thus, a blocking `MPI_WAITANY`

can be easily replaced by a nonblocking `MPI_TESTANY`. The same relation holds for the other pairs of Wait and Test functions defined in this section. (*End of rationale.*)

`MPI_WAITALL(count, array_of_requests, array_of_statuses)`

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

```
int MPI_Waitall(int count, MPI_Request *array_of_requests,
                MPI_Status *array_of_statuses)
```

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

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The *i*-th entry in `array_of_statuses` is set to the return status of the *i*-th operation. Requests that were created by nonblocking communication operations 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)

```
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
                MPI_Status *array_of_statuses)
```

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

Returns **flag** = **true** if all communications associated with active handles in the array have completed (this includes the case where no handle in the list is active). In this case, each status entry that corresponds to an active handle request is set to the status of the corresponding communication; if the request was allocated by a nonblocking communication call then it is deallocated, and the handle is 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 as errors in **MPI_WAITALL**.

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN	incount	length of array_of_requests (integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	outcount	number of completed requests (integer)
OUT	array_of_indices	array of indices of operations that completed (array of integers)
OUT	array_of_statuses	array of status objects for operations that completed (array of Status)

```
int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount,
                 int *array_of_indices, MPI_Status *array_of_statuses)
```

```
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
              ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

Waits until at least one of the operations associated with active handles in the list have completed. Returns in **outcount** the number of requests from the list **array_of_requests** that have completed. Returns in the first **outcount** locations of the array **array_of_indices** the indices of these operations (index within the array **array_of_requests**; the array is indexed

from zero in C and from one in Fortran). Returns in the first **outcount** locations of the array **array_of_status** the status for these completed operations. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to **MPI_REQUEST_NULL**.

If the list contains no active handles, then the call returns immediately with **outcount** = **MPI_UNDEFINED**.

When one or more of the communications completed by **MPI_WAITSSOME** 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_TESTSSOME(**incount**, **array_of_requests**, **outcount**, **array_of_indices**, **array_of_statuses**)

IN	incount	length of array_of_requests (integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	outcount	number of completed requests (integer)
OUT	array_of_indices	array of indices of operations that completed (array of integers)
OUT	array_of_statuses	array of status objects for operations that completed (array of Status)

```
int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount,
                 int *array_of_indices, MPI_Status *array_of_statuses)
```

```
MPI_TESTSSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
               ARRAY_OF_STATUSES, IERROR)
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

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

MPI_TESTSSOME is a local operation, which returns immediately, whereas **MPI_WAITSSOME** will block until a communication completes, if it was passed a list that contains at least one active handle. Both calls fulfill a **fairness** requirement: If a request for a receive repeatedly appears in a list of requests passed to **MPI_WAITSSOME** or **MPI_TESTSSOME**, and a matching send has been posted, then the receive will eventually succeed, unless the send is satisfied by another receive; and similarly for send requests.

Errors that occur during the execution of **MPI_TESTSSOME** are handled as for **MPI_WAITSSOME**.

Advice to users. The use of **MPI_TESTSSOME** is likely to be more efficient than the use of **MPI_TESTANY**. The former returns information on all completed communications, with the latter, a new call is required for each communication that completes.

A server with multiple clients can use `MPI_WAITSSOME` so as not to starve any client. Clients send messages to the server with service requests. The server calls `MPI_WAITSSOME` with one receive request for each client, and then handles all receives that completed. If a call to `MPI_WAITANY` is used instead, then one client could starve while requests from another client always sneak in first. (*End of advice to users.*)

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

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

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank > 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)
  END DO
  DO WHILE(.TRUE.)
    CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
    CALL DO_SERVICE(a(1,index)) ! handle one message
    CALL MPI_Irecv(a(1, index), n, MPI_REAL, index, tag,
                  comm, request_list(index), ierr)
  END DO
END IF
```

Example 3.16 Same code, using `MPI_WAITSSOME`.

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank > 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, requests(i), ierr)
  END DO
```

```

1      DO WHILE(.TRUE.)
2          CALL MPI_WAITSOME(size, request_list, numdone,
3                          indices, statuses, ierr)
4          DO i=1, numdone
5              CALL DO_SERVICE(a(1, indices(i)))
6              CALL MPI_Irecv(a(1, indices(i)), n, MPI_REAL, 0, tag,
7                          comm, requests(indices(i)), ierr)
8          END DO
9      END DO
10 END IF

```

3.8 Probe and Cancel

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

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

`MPI_Iprobe(source, tag, comm, flag, status)`

IN	source	source rank, or <code>MPI_ANY_SOURCE</code> (integer)
IN	tag	tag value or <code>MPI_ANY_TAG</code> (integer)
IN	comm	communicator (handle)
OUT	flag	(logical)
OUT	status	status object (Status)

```

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
               MPI_Status *status)

```

`MPI_Iprobe(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)`

LOGICAL FLAG

INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

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

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

A subsequent receive executed with the same context, and the source and tag returned in `status` by `MPI_PROBE` will receive the message that was matched by the probe, if no other intervening receive occurs after the probe. If the receiving process is multi-threaded, 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.

`MPI_PROBE(source, tag, comm, status)`

IN	<code>source</code>	source rank, or <code>MPI_ANY_SOURCE</code> (integer)
IN	<code>tag</code>	tag value, or <code>MPI_ANY_TAG</code> (integer)
IN	<code>comm</code>	communicator (handle)
OUT	<code>status</code>	status object (Status)

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

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

```
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

`MPI_PROBE` behaves like `MPI_IPROBE` except that it is a blocking call that returns only after a matching message has been found.

The MPI implementation of `MPI_PROBE` and `MPI_IPROBE` needs to guarantee progress: if a call to `MPI_PROBE` has been issued by a process, and a send that matches the probe 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 thread at the probing process). Similarly, if a process busy waits with `MPI_IPROBE` and a matching message has been issued, then the call to `MPI_IPROBE` will eventually return `flag = true` unless the message is received by another concurrent receive operation.

Example 3.17 Use blocking probe to wait for an incoming message.

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

```

1         END IF
2     END DO
3 END IF

```

Each message is received with the right type.

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

```

8     CALL MPI_COMM_RANK(comm, rank, ierr)
9     IF (rank.EQ.0) THEN
10        CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
11    ELSE IF(rank.EQ.1) THEN
12        CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
13    ELSE
14        DO i=1, 2
15            CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
16                          comm, status, ierr)
17            IF (status(MPI_SOURCE) = 0) THEN
18100                CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
19                          0, status, ierr)
20            ELSE
21200                CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
22                          0, status, ierr)
23            END IF
24        END DO
25    END IF
26

```

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

Advice to 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.*)

Returns `flag = false`, otherwise. If a receive operation might be canceled then one should call `MPI_TEST_CANCELLED` first, to check whether the operation was canceled, before checking on the other fields of the return status.

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

Advice to implementors. If a send operation uses an “eager” protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement `MPI_CANCEL`, this is still a local operation, since its completion does not depend on the code executed by other processes. If processing is required on another process, this should be transparent to the application (hence the need for an interrupt and an interrupt handler). (*End of advice to implementors.*)

3.9 Persistent communication requests

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

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

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

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

```
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
                  int tag, MPI_Comm comm, MPI_Request *request)
```

```

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

```

Creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.

```

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

```

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

```

int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm, MPI_Request *request)

```

```

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

```

Creates a persistent communication request for a buffered mode send.

```

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

```

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

```

int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm, MPI_Request *request)

```

```

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

```

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


```
1 MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)
```

2	IN	buf	initial address of send buffer (choice)
3	IN	count	number of elements sent (integer)
4	IN	datatype	type of each element (handle)
5	IN	dest	rank of destination (integer)
6	IN	tag	message tag (integer)
7	IN	comm	communicator (handle)
8	IN	request	communication request (handle)
9	OUT		
10			
11			

```
12 int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
13                   int tag, MPI_Comm comm, MPI_Request *request)
14
```

```
15 MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
16 <type> BUF(*)
17 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
18
```

19 Creates a persistent communication object for a ready mode send operation.

```
20
21 MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)
```

22	OUT	buf	initial address of receive buffer (choice)
23	IN	count	number of elements received (integer)
24	IN	datatype	type of each element (handle)
25	IN	source	rank of source or MPI_ANY_SOURCE (integer)
26	IN	tag	message tag or MPI_ANY_TAG (integer)
27	IN	comm	communicator (handle)
28	IN	request	communication request (handle)
29	OUT		
30			
31			

```
32
33 int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
34                  int tag, MPI_Comm comm, MPI_Request *request)
35
```

```
36 MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
37 <type> BUF(*)
38 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
39
```

40 Creates a persistent communication request for a receive operation. The argument **buf** is marked as **OUT** because the user gives permission to write on the receive buffer by passing the argument to **MPI_RECV_INIT**.

41 A persistent communication request is inactive after it was created — no active communication is attached to the request.

42 A communication (send or receive) that uses a persistent request is initiated by the function **MPI_START**.

43
44
45
46
47
48

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.

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 6 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

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

`MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)`

IN	sendbuf	initial address of send buffer (choice)
IN	sendcount	number of elements in send buffer (integer)
IN	sendtype	type of elements in send buffer (handle)
IN	dest	rank of destination (integer)
IN	sendtag	send tag (integer)
OUT	recvbuf	initial address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	type of elements in receive buffer (handle)
IN	source	rank of source (integer)
IN	recvtag	receive tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                int dest, int sendtag, void *recvbuf, int recvcount,
                MPI_Datatype recvtype, int source, MPI_Datatype recvtag,
                MPI_Comm comm, MPI_Status *status)

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
```

```

<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
SOURCE, RECV TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

```

Execute a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

INOUT	buf	initial address of send and receive buffer (choice)
IN	count	number of elements in send and receive buffer (integer)
IN	datatype	type of elements in send and receive buffer (handle)
IN	dest	rank of destination (integer)
IN	sendtag	send message tag (integer)
IN	source	rank of source (integer)
IN	recvtag	receive message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```

int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
    int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
    MPI_Status *status)

```

```

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.

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.

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`.

3.12 Derived datatypes

Up to here, all point to point communication have involved only contiguous buffers containing a sequence of elements of the same type. 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 back at the receiver site. This has the disadvantage of requiring additional memory-to-memory copy operations at both sites, even when the communication subsystem has scatter-gather capabilities. Instead, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

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

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

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

- A sequence of basic datatypes
- A sequence of integer (byte) displacements

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

Let

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

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

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

be the associated type signature. This type map, together with a base address *buf*, specifies a communication buffer: the communication buffer that consists of n entries, where the

i -th entry is at address $buf + disp_i$ and has type $type_i$. A message assembled from such a communication buffer will consist of n values, of the types defined by $Typesig$.

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 associated with `datatype`.

General datatypes can be used in all send and receive operations. We discuss, in Sec. 3.12.5, the case where the second argument `count` has value > 1 .

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

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

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

then

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + sizeof(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{3.1}$$

If $type_i$ requires alignment to a byte address that is a multiple of k_i , then ϵ is the least nonnegative increment needed to round $extent(Typemap)$ to the next multiple of $\max_i k_i$. The complete definition of **extent** is given on page 71.

Example 3.19 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 3.12.3. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. (*End of rationale.*)

3.12.1 Datatype constructors

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

1 MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

2 IN count replication count (nonnegative integer)
 3 IN oldtype old datatype (handle)
 4 OUT newtype new datatype (handle)

5
 6
 7 int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
 8 MPI_Datatype *newtype)

9
 10 MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
 11 INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

12 **newtype** is the datatype obtained by concatenating **count** copies of **oldtype**. Concatenation is defined using *extent* as the size of the concatenated copies.

13
 14 **Example 3.20** Let **oldtype** have type map $\{(\text{double}, 0), (\text{char}, 8)\}$, with extent 16, and let
 15 **count** = 3. The type map of the datatype returned by **newtype** is

16 $\{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{char}, 24), (\text{double}, 32), (\text{char}, 40)\}$;

17 i.e., alternating **double** and **char** elements, with displacements 0, 8, 16, 24, 32, 40.
 18
 19
 20

21 In general, assume that the type map of **oldtype** is

22 $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$,

23
 24 with extent *ex*. Then **newtype** has a type map with **count** · **n** entries defined by:

25 $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex),$
 26 $\dots, (type_0, disp_0 + ex \cdot (\text{count} - 1)), \dots, (type_{n-1}, disp_{n-1} + ex \cdot (\text{count} - 1))\}$.
 27
 28
 29
 30

31 **Vector** The function MPI_TYPE_VECTOR is a more general constructor that allows replication 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.
 32
 33
 34
 35

36 MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

37 IN count number of blocks (nonnegative integer)
 38 IN blocklength number of elements in each block (nonnegative integer)
 39 IN stride number of elements between start of each block (integer)
 40 IN oldtype old datatype (handle)
 41 OUT newtype new datatype (handle)

42
 43
 44
 45
 46
 47 int MPI_Type_vector(int count, int blocklength, int stride,
 48 MPI_Datatype oldtype, MPI_Datatype *newtype)

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

Example 3.21 Assume, again, that `oldtype` has type map $\{(\text{double}, 0), (\text{char}, 8)\}$, with extent 16. A call to `MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype)` will create the datatype with type map,

$$\{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{char}, 24), (\text{double}, 32), (\text{char}, 40),$$

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

That is, two blocks with three copies each of the old type, with a stride of 4 elements ($4 \cdot 16$ bytes) between the blocks.

Example 3.22 A call to `MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype)` will create the datatype,

$$\{(\text{double}, 0), (\text{char}, 8), (\text{double}, -32), (\text{char}, -24), (\text{double}, -64), (\text{char}, -56)\}.$$

In general, assume that `oldtype` has type map,

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

with extent ex . Let `bl` be the `blocklength`. The newly created datatype has a type map with `count · bl · n` entries:

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

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

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

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

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

$$(type_0, disp_0 + stride \cdot (count - 1) \cdot ex), \dots,$$

$$(type_{n-1}, disp_{n-1} + stride \cdot (count - 1) \cdot ex), \dots,$$

$$(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots,$$

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

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, count, n, oldtype, newtype)`, n arbitrary.

Hvector The function `MPI_TYPE_HVECTOR` is identical to `MPI_TYPE_VECTOR`, except that **stride** is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Sec. 3.12.7. (H stands for “heterogeneous”).

`MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)`

IN	count	number of blocks (nonnegative integer)
IN	blocklength	number of elements in each block (nonnegative integer)
IN	stride	number of bytes between start of each block (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

Assume that **oldtype** has type map,

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

with extent ex . Let **bl** be the **blocklength**. The newly created datatype has a type map with $count \cdot bl \cdot n$ entries:

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

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

`MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)`

IN	<code>count</code>	number of blocks – also number of entries in <code>array_of_displacements</code> and <code>array_of_blocklengths</code> (non-negative integer)
IN	<code>array_of_blocklengths</code>	number of elements per block (array of nonnegative integers)
IN	<code>array_of_displacements</code>	displacement for each block, in multiples of <code>oldtype</code> extent (array of integer)
IN	<code>oldtype</code>	old datatype (handle)
OUT	<code>newtype</code>	new datatype (handle)

```
int MPI_Type_indexed(int count, int *array_of_blocklengths,
                    int *array_of_displacements, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

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

Example 3.23 Let `oldtype` have type map $\{(\text{double}, 0), (\text{char}, 8)\}$, with extent 16. Let $\mathbf{B} = (3, 1)$ and let $\mathbf{D} = (4, 0)$. A call to `MPI_TYPE_INDEXED(2, B, D, oldtype, newtype)` returns a datatype with type map,

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

$$(\text{double}, 0), (\text{char}, 8)\}.$$

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

In general, assume that `oldtype` has type map,

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

with extent ex . Let \mathbf{B} be the `array_of_blocklength` argument and \mathbf{D} be the `array_of_displacements` argument. The newly created datatype has $n \cdot \sum_{i=0}^{\text{count}-1} B[i]$ entries:

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

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

```

1      (type0, disp0 + D[count - 1] · ex), ..., (typen-1, dispn-1 + D[count - 1] · ex), ...,
2      (type0, disp0 + (D[count - 1] + B[count - 1] - 1) · ex), ...,
3      (typen-1, dispn-1 + (D[count - 1] + B[count - 1] - 1) · ex)).
4
5
6

```

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

```

9      D[j] = j · stride, j = 0, ..., count - 1,
10
11 and
12
13      B[j] = blocklength, j = 0, ..., count - 1.
14

```

Hindexed The function `MPI_TYPE_HINDEXED` is identical to `MPI_TYPE_INDEXED`, except that block displacements in `array_of_displacements` are specified in bytes, rather than in multiples of the `oldtype` extent.

```

19 MPI_TYPE_HINDEXED( count, array_of_blocklengths, array_of_displacements, oldtype, new-
20 type)
21

```

22	IN	count	number of blocks – also number of entries in
23			array_of_displacements and array_of_blocklengths (inte-
24			ger)
25	IN	array_of_blocklengths	number of elements in each block (array of nonnega-
26			tive integers)
27	IN	array_of_displacements	byte displacement of each block (array of integer)
28	IN	oldtype	old datatype (handle)
29			
30	OUT	newtype	new datatype (handle)

```

31
32 int MPI_Type_hindexed(int count, int *array_of_blocklengths,
33                       MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
34                       MPI_Datatype *newtype)
35
36 MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
37                   OLDTYPE, NEWTYPE, IERROR)
38 INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
39 OLDTYPE, NEWTYPE, IERROR
40

```

Assume that `oldtype` has type map,

```

43      {(type0, disp0), ..., (typen-1, dispn-1)},
44

```

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

```

48      {(type0, disp0 + D[0]), ..., (typen-1, dispn-1 + D[0]), ...,

```

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

Struct `MPI_TYPE_STRUCT` is the most general type constructor. It further generalizes the previous one in that it allows each block to consist of replications of different datatypes.

`MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)`

IN	<code>count</code>	number of blocks (integer) – also number of entries in arrays <code>array_of_types</code> , <code>array_of_displacements</code> and <code>array_of_blocklengths</code>
IN	<code>array_of_blocklength</code>	number of elements in each block (array of integer)
IN	<code>array_of_displacements</code>	byte displacement of each block (array of integer)
IN	<code>array_of_types</code>	type of elements in each block (array of handles to datatype objects)
OUT	<code>newtype</code>	new datatype (handle)

```

int MPI_Type_struct(int count, int *array_of_blocklengths,
                    MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
                    MPI_Datatype *newtype)

```

```

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
                 ARRAY_OF_TYPES, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
ARRAY_OF_TYPES(*), NEWTYPE, IERROR

```

Example 3.24 Let `type1` have type map,

$$\{(double, 0), (char, 8)\},$$

with extent 16. Let $B = (2, 1, 3)$, $D = (0, 16, 26)$, and $T = (MPI_FLOAT, type1, MPI_CHAR)$. Then a call to `MPI_TYPE_STRUCT(3, B, D, T, newtype)` returns a datatype with type map,

$$\{(float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27), (char, 28)\}.$$

That is, two copies of `MPI_FLOAT` starting at 0, followed by one copy of `type1` starting at 16, followed by three copies of `MPI_CHAR`, starting at 26. (We assume that a float occupies four bytes.)

In general, let T be the `array_of_types` argument, where $T[i]$ is a handle to,

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

with extent ex_i . Let B be the `array_of_blocklength` argument and D be the `array_of_displacements` argument. Let c be the `count` argument. Then the newly created datatype has a type map with $\sum_{i=0}^{c-1} B[i] \cdot n_i$ entries:

$$\begin{aligned} &\{(type_0^0, disp_0^0 + D[0]), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0]), \dots, \\ &(type_0^0, disp_0^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + D[0] + (B[0] - 1) \cdot ex_0), \dots, \\ &(type_0^{c-1}, disp_0^{c-1} + D[c-1]), \dots, (type_{n_{c-1}-1}^{c-1}, disp_{n_{c-1}-1}^{c-1} + D[c-1]), \dots, \\ &(type_0^{c-1}, disp_0^{c-1} + D[c-1] + (B[c-1] - 1) \cdot ex_{c-1}), \dots, \\ &(type_{n_{c-1}-1}^{c-1}, disp_{n_{c-1}-1}^{c-1} + D[c-1] + (B[c-1] - 1) \cdot ex_{c-1})\}. \end{aligned}$$

A call to `MPI_TYPE_HINDEXED(count, B, D, oldtype, newtype)` is equivalent to a call to `MPI_TYPE_STRUCT(count, B, D, T, newtype)`, where each entry of T is equal to `oldtype`.

3.12.2 Address and extent functions

The displacements in a general datatype are relative to some initial buffer address. **Absolute addresses** can be substituted for these displacements: we treat them as displacements relative to “address zero,” the start of the address space. This initial address zero is indicated by the constant `MPI_BOTTOM`. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the `buf` argument is passed the value `MPI_BOTTOM`.

The address of a location in memory can be found by invoking the function `MPI_ADDRESS`.

`MPI_ADDRESS(location, address)`

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

`int MPI_Address(void* location, MPI_Aint *address)`

`MPI_ADDRESS(LOCATION, ADDRESS, IERROR)`

<type> LOCATION(*)

INTEGER ADDRESS, IERROR

Returns the (byte) address of `location`.

Example 3.25 Using `MPI_ADDRESS` for an array.

```

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

```

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

The following auxiliary functions provide useful information on derived datatypes.

`MPI_TYPE_EXTENT(datatype, extent)`

IN	<code>datatype</code>	datatype (handle)
OUT	<code>extent</code>	datatype extent (integer)

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

```

MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR

```

Returns the extent of a datatype, where extent is as defined on page 71.

`MPI_TYPE_SIZE(datatype, size)`

IN	<code>datatype</code>	datatype (handle)
OUT	<code>size</code>	datatype size (integer)

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

```

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

```

`MPI_TYPE_SIZE` returns 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.

3.12.3 Lower-bound and upper-bound markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, and override the definition given on page 71. This allows one to define a datatype that has

“holes” at its beginning or its end, or a datatype with entries that extend above the upper bound or below the lower bound. Examples of such usage are provided in Sec. 3.12.7. Also, the user may want to override the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to override default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional “pseudo-datatypes,” `MPI_LB` and `MPI_UB`, that can be used, respectively, to mark the lower bound or the upper bound of a datatype. These pseudo-datatypes occupy no space ($extent(MPI_LB) = extent(MPI_UB) = 0$). They do not affect the size or count of a datatype, and do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

Example 3.26 Let $D = (-3, 0, 6)$; $T = (MPI_LB, MPI_INT, MPI_UB)$, and $B = (1, 1, 1)$. Then a call to `MPI_TYPE_STRUCT(3, B, D, T, type1)` creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the sequence $\{(lb, -3), (int, 0), (ub, 6)\}$. If this type is replicated twice by a call to `MPI_TYPE_CONTIGUOUS(2, type1, type2)` then the newly created type can be described by the sequence $\{(lb, -3), (int, 0), (int, 9), (ub, 15)\}$. (An entry of type `ub` can be deleted if there is another entry of type `ub` with a higher displacement; an entry of type `lb` can be deleted if there is another entry of type `lb` with a lower displacement.)

In general, if

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

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

$$lb(Typemap) = \begin{cases} \min_j disp_j & \text{if no entry has basic type } lb \\ \min_j \{disp_j \text{ such that } type_j = lb\} & \text{otherwise} \end{cases}$$

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

$$ub(Typemap) = \begin{cases} \max_j disp_j + sizeof(type_j) + \epsilon & \text{if no entry has basic type } ub \\ \max_j \{disp_j \text{ such that } type_j = ub\} & \text{otherwise} \end{cases}$$

Then

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

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

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

The two functions below can be used for finding the lower bound and the upper bound of a datatype.

`MPI_TYPE_LB(datatype, displacement)`

IN	datatype	datatype (handle)
OUT	displacement	displacement of lower bound from origin, in bytes (integer)

`int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)`

```

MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR

```

```

MPI_TYPE_UB( datatype, displacement)
    IN      datatype          datatype (handle)
    OUT     displacement      displacement of upper bound from origin, in bytes (integer)

```

```

int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)

```

```

MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR

```

3.12.4 Commit and free

A datatype object has to be **committed** before it can be used in a communication. A committed datatype can still be used as a argument in datatype constructors. There is no need to commit basic datatypes. They are “pre-committed.”

```

MPI_TYPE_COMMIT(datatype)
    INOUT  datatype          datatype that is committed (handle)

```

```

int MPI_Type_commit(MPI_Datatype *datatype)

```

```

MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

```

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

Advice to implementors. The system may “compile” at commit time an internal representation for the datatype that facilitates communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (*End of advice to implementors.*)

```

MPI_TYPE_FREE(datatype)
    INOUT  datatype          datatype that is freed (handle)

```

```

int MPI_Type_free(MPI_Datatype *datatype)

```

```

MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

```


Marks the datatype object associated with `datatype` for deallocation and sets `datatype` to `MPI_DATATYPE_NULL`. Any communication that is currently using this datatype will complete normally. Derived datatypes that were defined from the freed datatype are not affected.

Example 3.27 The following code fragment gives examples of using `MPI_TYPE_COMMIT`.

```

1  INTEGER type1, type2
2  CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
3      ! new type object created
4  CALL MPI_TYPE_COMMIT(type1, ierr)
5      ! now type1 can be used for communication
6  type2 = type1
7      ! type2 can be used for communication
8      ! (it is a handle to same object as type1)
9  CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
10      ! new uncommitted type object created
11 CALL MPI_TYPE_COMMIT(type1, ierr)
12      ! now type1 can be used anew for communication
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

```

Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather than 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.*)

3.12.5 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,

```

37 MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
38 MPI_TYPE_COMMIT(newtype)
39 MPI_SEND(buf, 1, newtype, dest, tag, comm).
40
41
42
43
44
45
46
47
48

```

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,

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

and extent `extent`. (Empty entries of “pseudo-type” `MPI_UB` and `MPI_LB` 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 $addr_{i,j} = \text{buf} + \text{extent} \cdot i + \text{disp}_j$ and has type $type_j$, for $i = 0, \dots, \text{count} - 1$ and $j = 0, \dots, n - 1$. These entries need not be contiguous, nor distinct; their order can be arbitrary.

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

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

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

with extent $extent$. (Again, empty entries of “pseudo-type” `MPI_UB` and `MPI_LB` 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 $\text{buf} + \text{extent} \cdot i + \text{disp}_j$ 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 3.28 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
...
CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
...
CALL MPI_SEND( a, 4, MPI_REAL, ...)
CALL MPI_SEND( a, 2, type2, ...)
CALL MPI_SEND( a, 1, type22, ...)
CALL MPI_SEND( a, 1, type4, ...)
...
CALL MPI_RECV( a, 4, MPI_REAL, ...)
CALL MPI_RECV( a, 2, type2, ...)
CALL MPI_RECV( a, 1, type22, ...)
CALL MPI_RECV( a, 1, type4, ...)
```

Each of the sends matches any of the receives.

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

A datatype may specify overlapping entries. If such a datatype is used in a receive operation, that is, if some part of the receive buffer is written more than once by the receive operation, then the call is erroneous.

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

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

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 $0 \leq k \leq \text{count} \cdot n$. The number of basic elements received can be retrieved from **status** using the query function **MPI_GET_ELEMENTS**.

MPI_GET_ELEMENTS(**status**, **datatype**, **count**)

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

int MPI_Get_elements(**MPI_Status** ***status**, **MPI_Datatype** **datatype**, **int** ***count**)

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

The previously defined function, **MPI_GET_COUNT** (Sec. 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 $0 \leq k \leq \text{count}$. If **MPI_GET_COUNT** returns k , then the number of basic elements received (and the value returned by **MPI_GET_ELEMENTS**) 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** returns the value **MPI_UNDEFINED**.

Example 3.29 Usage of **MPI_GET_COUNT** and **MPI_GET_ELEMENT**.

```

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

```

The function **MPI_GET_ELEMENTS** can also be used after a probe to find the number of elements in the probed message. Note that the two functions **MPI_GET_COUNT** and **MPI_GET_ELEMENTS** return the same values when they are used with basic datatypes.

Rationale. The extension given to the definition of **MPI_GET_COUNT** seems natural: one would expect this function to return the value of the **count** argument, when

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

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

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

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

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

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

Advice to implementors. There is no need to distinguish (absolute) addresses and (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.*)

Note that in Fortran, Fortran INTEGERS may be too small to contain an address (e.g., 32 bit INTEGERS on a machine with 64bit pointers). Because of this, in Fortran, implementations may restrict the use of absolute addresses to only part of the process memory, and restrict the use of relative displacements to subranges of the process memory where they are constrained by the size of Fortran INTEGERS.

3.12.7 Examples

The following examples illustrate the use of derived datatypes.

Example 3.30 Send and receive a section of a 3D array.

```

REAL a(100,100,100), e(9,9,9)
INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)

C      extract the section a(1:17:2, 3:11, 2:10)
C      and store it in e(:, :, :).

CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C      create datatype for a 1D section
CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)

C      create datatype for a 2D section
CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)

C      create datatype for the entire section
CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice, 1,
                      threeslice, ierr)

CALL MPI_TYPE_COMMIT( threeslice, ierr)
CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9,
                  MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)

```

Example 3.31 Copy the (strictly) lower triangular part of a matrix.

```

REAL a(100,100), b(100,100)
INTEGER disp(100), blocklen(100), ltype, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)

C    copy lower triangular part of array a
C    onto lower triangular part of array b

CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

C    compute start and size of each column
DO i=1, 100
    disp(i) = 100*(i-1) + i
    block(i) = 100-i
END DO

C    create datatype for lower triangular part
CALL MPI_TYPE_INDEXED( 100, block, disp, MPI_REAL, ltype, ierr)

CALL MPI_TYPE_COMMIT(ltype, ierr)
CALL MPI_SENDRECV( a, 1, ltype, myrank, 0, b, 1,
    ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)

```

Example 3.32 Transpose a matrix.

```

REAL a(100,100), b(100,100)
INTEGER row, xpose, sizeofreal, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)

C    transpose matrix a onto b

CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C    create datatype for one row
CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C    create datatype for matrix in row-major order
CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)

CALL MPI_TYPE_COMMIT( xpose, ierr)

C    send matrix in row-major order and receive in column major order
CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,
    MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)

```

Example 3.33 Another approach to the transpose problem:

```

REAL a(100,100), b(100,100)

```

```

1      INTEGER disp(2), blocklen(2), type(2), row, row1, sizeofreal
2      INTEGER myrank, ierr
3      INTEGER status(MPI_STATUS_SIZE)
4
5      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
6
7  C      transpose matrix a onto b
8
9      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)
10
11 C      create datatype for one row
12 CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
13
14 C      create datatype for one row, with the extent of one real number
15 disp(1) = 0
16 disp(2) = sizeofreal
17 type(1) = row
18 type(2) = MPI_UB
19 blocklen(1) = 1
20 blocklen(2) = 1
21 CALL MPI_TYPE_STRUCT( 2, blocklen, disp, type, row1, ierr)
22
23 CALL MPI_TYPE_COMMIT( row1, ierr)
24
25 C      send 100 rows and receive in column major order
26 CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
27                   MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
28

```

Example 3.34 We manipulate an array of structures.

```

29
30 struct Partstruct
31 {
32     int    class; /* particle class */
33     double d[6]; /* particle coordinates */
34     char   b[7]; /* some additional information */
35 };
36
37 struct Partstruct    particle[1000];
38
39 int                  i, dest, rank;
40 MPI_Comm             comm;
41
42
43 /* build datatype describing structure */
44
45 MPI_Datatype Particletype;
46 MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
47 int          blocklen[3] = {1, 6, 7};
48

```

```

MPI_Aint    disp[3];
int         base;

/* compute displacements of structure components */

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i <3; i++) disp[i] -= base;

MPI_Type_struct( 3, blocklen, disp, type, &Particletype);

/* If compiler does padding in mysterious ways,
the following may be safer */

MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
int          blocklen1[4] = {1, 6, 7, 1};
MPI_Aint     disp1[4];

/* compute displacements of structure components */

MPI_Address( particle, disp1);
MPI_Address( particle[0].d, disp1+1);
MPI_Address( particle[0].b, disp1+2);
MPI_Address( particle+1, disp1+3);
base = disp1[0];
for (i=0; i <4; i++) disp1[i] -= base;

/* build datatype describing structure */

MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);

/* 4.1:
send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);

/* 4.2:
send only the entries of class zero particles,
preceded by the number of such entries */

MPI_Datatype Zparticles; /* datatype describing all particles
                           with class zero (needs to be recomputed

```



```

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

```

```

/* 4.3:
send the first two coordinates of all entries */

MPI_Datatype Allpairs;      /* datatype for all pairs of coordinates */

MPI_Aint sizeofentry;

MPI_Type_extent( Particletype, &sizeofentry);

/* sizeofentry can also be computed by subtracting the address
of particle[0] from the address of particle[1] */

MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
MPI_Type_commit( &Allpairs);
MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);

/* an alternative solution to 4.3 */

MPI_Datatype Onepair;      /* datatype for one pair of coordinates, with
                           the extent of one particle entry */

MPI_Aint disp2[3];
MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int blocklen2[3] = {1, 2, 1};

MPI_Address( particle, disp2);
MPI_Address( particle[0].d, disp2+1);
MPI_Address( particle+1, disp2+2);
base = disp2[0];
for (i=0; i<2; i++) disp2[i] -= base;

MPI_Type_struct( 3, blocklen2, disp2, type2, &Onepair);
MPI_Type_commit( &Onepair);
MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);

```

Example 3.35 The same manipulations as in the previous example, but use absolute addresses in datatypes.

```

struct Partstruct
{
    int class;
    double d[6];
    char b[7];
};

struct Partstruct particle[1000];

```

```

1          /* build datatype describing first array entry */
2
3  MPI_Datatype Particletype;
4  MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
5  int          block[3] = {1, 6, 7};
6  MPI_Aint     disp[3];
7
8  MPI_Address( particle, disp);
9  MPI_Address( particle[0].d, disp+1);
10 MPI_Address( particle[0].b, disp+2);
11 MPI_Type_struct( 3, block, disp, type, &Particletype);
12
13 /* Particletype describes first array entry -- using absolute
14    addresses */
15
16          /* 5.1:
17             send the entire array */
18
19 MPI_Type_commit( &Particletype);
20 MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
21
22
23          /* 5.2:
24             send the entries of class zero,
25             preceded by the number of such entries */
26
27 MPI_Datatype Zparticles, Ztype;
28
29 MPI_Aint zdisp[1000]
30 int zblock[1000], i, j, k;
31 int zzblock[2] = {1,1};
32 MPI_Datatype zztype[2];
33 MPI_Aint     zzdisp[2];
34
35 j=0;
36 for (i=0; i < 1000; i++)
37     if (particle[i].index==0)
38     {
39         for (k=i+1; (k < 1000)&&(particle[k].index = 0) ; k++);
40         zdisp[j] = i;
41         zblock[j] = k-i;
42         j++;
43         i = k;
44     }
45 MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
46 /* Zparticles describe particles with class zero, using
47    their absolute addresses*/
48

```

```

/* prepend particle count */
MPI_Address(&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzbblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);

```

Example 3.36 Handling of unions.

```

union {
    int    ival;
    float  fval;
} u[1000]

int    utype;

/* All entries of u have identical type; variable
   utype keeps track of their current type */

MPI_Datatype    type[2];
int             blocklen[2] = {1,1};
MPI_Aint        disp[2];
MPI_Datatype    mpi_utype[2];
MPI_Aint        i,j;

/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */

MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-i;
type[1] = MPI_UB;

type[0] = MPI_INT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);

type[0] = MPI_FLOAT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);

for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);

/* actual communication */

MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);

```

3.13 Pack and unpack

Some existing communication libraries provide pack/unpack functions for sending noncontiguous data. In these, the user explicitly packs data into a contiguous buffer before sending it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are described in Section 3.12, allow one, in most cases, to avoid explicit packing and unpacking. The user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer. The pack/unpack routines are provided for compatibility with previous libraries. Also, they provide some functionality that is not otherwise available in MPI. For instance, a message can be received in several parts, where the receive operation done on a later part may depend on the content of a former part. Another use is that outgoing messages may be explicitly buffered in user supplied space, thus overriding the system buffering policy. Finally, the availability of pack and unpack operations facilitates the development of additional communication libraries layered on top of MPI.

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

IN	inbuf	input buffer start (choice)
IN	incount	number of input data items (integer)
IN	datatype	datatype of each input data item (handle)
OUT	outbuf	output buffer start (choice)
IN	outsize	output buffer size, in bytes (integer)
INOUT	position	current position in buffer, in bytes (integer)
IN	comm	communicator for packed message (handle)

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
            int outsize, int *position, MPI_Comm comm)
```

```
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
<type> INBUF(*), OUTBUF(*)
INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

Packs the message in the send buffer specified by **inbuf**, **incount**, **datatype** into the buffer space specified by **outbuf** and **outcount**. 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.

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)			1
IN	inbuf	input buffer start (choice)	2
IN	insize	size of input buffer, in bytes (integer)	3
INOUT	position	current position in bytes (integer)	4
OUT	outbuf	output buffer start (choice)	5
IN	outcount	number of items to be unpacked (integer)	6
IN	datatype	datatype of each output data item (handle)	7
IN	comm	communicator for packed message (handle)	8

```

int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
               int outcount, MPI_Datatype datatype, MPI_Comm comm)
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
           IERROR)
<type> INBUF(*), OUTBUF(*)
INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

```

Unpacks a message into the receive buffer specified by **outbuf**, **outcount**, **datatype** from the buffer space specified by **inbuf** and **insize**. The output buffer can be any communication buffer allowed in **MPI_RECV**. The input buffer is a contiguous storage area containing **insize** bytes, starting at address **inbuf**. The input value of **position** is the first location in the input buffer occupied by the packed message. **position** is incremented by the size of the packed message, so that the output value of **position** is the first location in the input buffer after the locations occupied by the message that was unpacked. **comm** is the communicator used to receive the packed message.

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

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

Several messages can be successively packed into one **packing unit**. This is effected by several successive **related** calls to **MPI_PACK**, where the first call provides **position = 0**, and each successive call inputs the value of **position** that was output by the previous call, and the same values for **outbuf**, **outcount** and **comm**. This packing unit now contains the

equivalent information that would have been stored in a message by one send call with a send buffer that is the “concatenation” of the individual send buffers.

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

A message sent with any type (including `MPI_PACKED`) can be received using the type `MPI_PACKED`. Such a message can then be unpacked by calls to `MPI_UNPACK`.

A packing unit (or a message created by a regular, “typed” send) can be unpacked into several successive messages. This is effected by several successive related calls to `MPI_UNPACK`, where the first call provides `position = 0`, and each successive call inputs the value of `position` that was output by the previous call, and the same values for `inbuf`, `insize` and `comm`.

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

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

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

`MPI_PACK_SIZE(incount, datatype, comm, size)`

IN	<code>incount</code>	count argument to packing call (integer)
IN	<code>datatype</code>	datatype argument to packing call (handle)
IN	<code>comm</code>	communicator argument to packing call (handle)
OUT	<code>size</code>	upper bound on size of packed message, in bytes (integer)

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
                  int *size)
```

```
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)`.

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 3.37 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);
    MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
    MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else /* RECEIVER CODE */
    MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD)
}
```

Example 3.38 A elaborate example.

```
int position, i;
float a[1000];
char buff[1000]

....

MPI_Comm_rank(MPI_Comm_world, &myrank);
if (myrank == 0)
{
    /* SENDER CODE */

    int len[2];
    MPI_Aint disp[2];
    MPI_Datatype type[2], newtype;

    /* build datatype for i followed by a[0]...a[i-1] */

    len[0] = 1;
    len[1] = i;
    MPI_Address( &i, disp);
    MPI_Address( a, disp+1);
```



```

1   type[0] = MPI_INT;
2   type[1] = MPI_FLOAT;
3   MPI_Type_struct( 2, len, disp, type, &newtype);
4   MPI_Type_commit( &newtype);
5
6   /* Pack i followed by a[0]...a[i-1]*/
7
8   position = 0;
9   MPI_Pack( MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);
10
11  /* Send */
12
13  MPI_Send( buff, position, MPI_PACKED, 1, 0,
14           MPI_COMM_WORLD)
15
16  /* *****
17   One can replace the last three lines with
18   MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
19   ***** */
20  }
21  else /* myrank == 1 */
22  {
23      /* RECEIVER CODE */
24
25      MPI_Status status;
26
27      /* Receive */
28
29      MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, &status);
30
31      /* Unpack i */
32
33      position = 0;
34      MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
35
36      /* Unpack a[0]...a[i-1] */
37      MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
38  }

```

Example 3.39 Each process sends a count, followed by count characters to the root; the root concatenate all characters into one string.

```

41
42  int count, gsize, counts[64], totalcount, k1, k2, k,
43      displs[64], position, concat_pos;
44  char chr[100], *lbuf, *rbuf, *cbuf;
45  ...
46  MPI_Comm_size(comm, &gsize);
47  MPI_Comm_rank(comm, &myrank);
48

```

```

    /* allocate local pack buffer */
    MPI_Pack_size(1, MPI_INT, comm, &k1);
    MPI_Pack_size(count, MPI_CHAR, &k2);
    k = k1+k2;
    lbuf = (char *)malloc(k);

    /* pack count, followed by count characters */
    position = 0;
    MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
    MPI_Pack(chr, count, MPI_CHAR, &lbuf, k, &position, comm);

    if (myrank != root)
        /* gather at root sizes of all packed messages */
        MPI_Gather(&position, 1, MPI_INT, NULL, NULL,
            NULL, root, comm);

        /* gather at root packed messages */
        MPI_Gatherv(&lbuf, position, MPI_PACKED, NULL,
            NULL, NULL, NULL, root, comm);

else { /* root code */
    /* gather sizes of all packed messages */
    MPI_Gather(&position, 1, MPI_INT, counts, 1,
        MPI_INT, root, comm);

    /* gather all packed messages */
    displs[0] = 0;
    for (i=1; i < gsize; i++)
        displs[i] = displs[i-1] + counts[i-1];
    totalcount = displs[gsz-1] + counts[gsz-1];
    rbuf = (char *)malloc(totalcount);
    cbuf = (char *)malloc(totalcount);
    MPI_Gatherv(lbuf, position, MPI_PACKED, rbuf,
        counts, displs, MPI_PACKED, root, comm);

    /* unpack all messages and concatenate strings */
    concat_pos = 0;
    for (i=0; i < gsize; i++) {
        position = 0;
        MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
            &position, &count, 1, MPI_INT, comm);
        MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
            &position, cbuf+concat_pos, count, MPI_CHAR, comm);
        concat_pos += count;
    }
    cbuf[concat_pos] = '\0';
}

```

Chapter 4

Collective Communication

4.1 Introduction and Overview

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

- Barrier synchronization across all group members (Sec. 4.3).
- Broadcast from one member to all members of a group (Sec. 4.4). This is shown in figure 4.1.
- Gather data from all group members to one member (Sec. 4.5). This is shown in figure 4.1.
- Scatter data from one member to all members of a group (Sec. 4.6). This is shown in figure 4.1.
- A variation on Gather where all members of the group receive the result (Sec. 4.7). This is shown as “allgather” in figure 4.1.
- Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all) (Sec. 4.8). This is shown as “alltoall” in figure 4.1.
- Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all group members and a variation where the result is returned to only one member (Sec. 4.9).
- A combined reduction and scatter operation (Sec. 4.10).
- Scan across all members of a group (also called prefix) (Sec. 4.11).

A collective operation is executed by having all processes in the group call the communication routine, with matching arguments. The syntax and semantics of the collective operations are defined to be consistent with the syntax and semantics of the point-to-point operations. Thus, general datatypes are allowed and must match between sending and receiving processes as specified in Chapter 3. One of the key arguments is a communicator that defines the group of participating processes and provides a context for the operation. Several collective routines such as broadcast and gather have a single originating or receiving process. Such processes are called the *root*. Some arguments in the collective functions

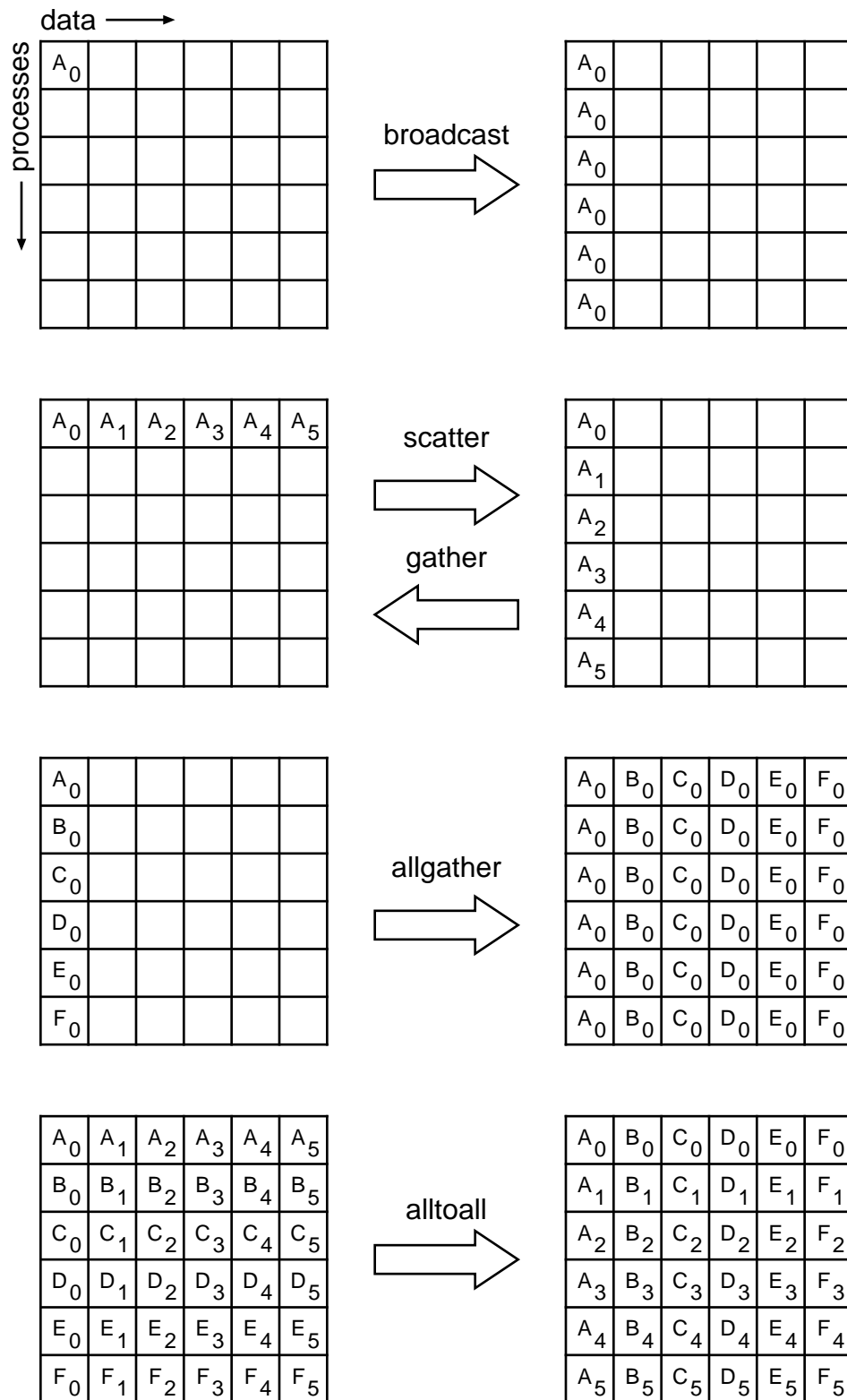


Figure 4.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0 , but after the broadcast all processes contain it.

are specified as “significant only at root,” and are ignored for all participants except the root. The reader is referred to Chapter 3 for information concerning communication buffers, general datatypes and type matching rules, and to Chapter 5 for information on how to define groups and create communicators.

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. Distinct type maps (the layout in memory, see Sec. 3.12) between sender and receiver are still allowed.

Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to access locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise indicated in the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function.

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. A more detailed discussion of correct use of collective routines is found in Sec. 4.12.

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

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

The collective operations do not accept a message tag argument. If future revisions of MPI define non-blocking collective functions, then tags (or a similar mechanism) will need to be added so as to allow the dis-ambiguation of multiple, pending, collective operations. (*End of rationale.*)

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

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 Sec. 4.12. (*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 must 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 Sec. 4.12. (*End of advice to implementors.*)

4.2 Communicator argument

The key concept of the collective functions is to have a “group” of participating processes. The routines do not have a group identifier as an explicit argument. Instead, there is a communicator argument. For the purposes of this chapter, a communicator can be thought of as a group identifier linked with a context. An inter-communicator, that is, a communicator that spans two groups, is *not* allowed as an argument to a collective function.

4.3 Barrier synchronization

`MPI_BARRIER(comm)`

IN `comm` communicator (handle)

`int MPI_Barrier(MPI_Comm comm)`

`MPI_BARRIER(COMM, IERROR)`

INTEGER `COMM`, `IERROR`

`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.

4.4 Broadcast

`MPI_BCAST(buffer, count, datatype, root, comm)`

INOUT `buffer` starting address of buffer (choice)

IN `count` number of entries in buffer (integer)

IN `datatype` data type of buffer (handle)

IN `root` rank of broadcast root (integer)

IN `comm` communicator (handle)

`int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`

`MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)`

<type> `BUFFER(*)`

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

`MPI_BCAST` broadcasts a message from the process with rank `root` to all processes of the group, itself included. It is called by all members of group using the same arguments for `comm`, `root`. On return, the contents of `root`’s communication buffer has been copied to all processes.

General, derived datatypes are allowed for `datatype`. The type signature of `count`, `datatype` on any process must be equal to the type signature of `count`, `datatype` at the root.

This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. `MPI_BCAST` and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

4.4.1 Example using `MPI_BCAST`

Example 4.1 Broadcast 100 ints from process 0 to every process in the group.

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

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

4.5 Gather

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

IN	<code>sendbuf</code>	starting address of send buffer (choice)
IN	<code>sendcount</code>	number of elements in send buffer (integer)
IN	<code>sendtype</code>	data type of send buffer elements (handle)
OUT	<code>recvbuf</code>	address of receive buffer (choice, significant only at root)
IN	<code>recvcount</code>	number of elements for any single receive (integer, significant only at root)
IN	<code>recvtype</code>	data type of recv buffer elements (significant only at root) (handle)
IN	<code>root</code>	rank of receiving process (integer)
IN	<code>comm</code>	communicator (handle)

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

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

Each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome

is *as if* each of the `n` processes in the group (including the root process) had executed a call to

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

and the root had executed `n` calls to

```
MPI_Recv(recvbuf + i * recvcount * extent(recvtype), recvcount, recvtype, i, ...),
```

where `extent(recvtype)` is the type extent obtained from a call to `MPI_Type_extent()`.

An alternative description is that the `n` messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to `MPI_RECV(recvbuf, recvcount*n, recvtype, ...)`.

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

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

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

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

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


```

1  MPI_GATHERV( sendbuf, sendcount, sendtype, recvbuf, recvcunts, displs, recvtype, root,
2  comm)

```

3	IN	sendbuf	starting address of send buffer (choice)
4	IN	sendcount	number of elements in send buffer (integer)
5	IN	sendtype	data type of send buffer elements (handle)
6	IN	sendtype	data type of send buffer elements (handle)
7	OUT	recvbuf	address of receive buffer (choice, significant only at root)
8			
9	IN	recvcunts	integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
10			
11			
12			
13	IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)
14			
15			
16			
17	IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
18			
19	IN	root	rank of receiving process (integer)
20			
21	IN	comm	communicator (handle)
22			

```

23  int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
24                void* recvbuf, int *recvcunts, int *displs,
25                MPI_Datatype recvtype, int root, MPI_Comm comm)

```

```

26  MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
27              RECVTYPE, ROOT, COMM, IERROR)
28  <type> SENDBUF(*), RECVBUF(*)
29  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
30  COMM, IERROR
31

```

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count of data from each process, since **recvcunts** is now an array. It also allows more flexibility as to where the data is placed on the root, by providing the new argument, **displs**.

The outcome is *as if* each process, including the root process, sends a message to the root,

```

37
38  MPI_Send(sendbuf, sendcount, sendtype, root, ...),

```

and the root executes **n** receives,

```

41  MPI_Recv(recvbuf + disp[i] * extent(recvtype), recvcunts[i], recvtype, i, ...).
42

```

Messages are placed in the receive buffer of the root process in rank order, that is, the data sent from process **j** is placed in the **j**th portion of the receive buffer **recvbuf** on process **root**. The **j**th portion of **recvbuf** begins at offset **displs[j]** elements (in terms of **recvtype**) into **recvbuf**.

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

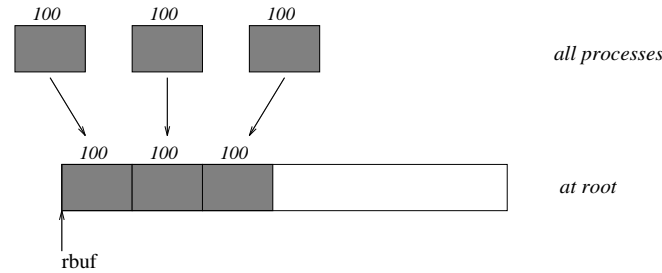


Figure 4.2: The root process gathers 100 ints from each process in the group.

The type signature implied by `sendcount`, `sendtype` on process `i` must be equal to the type signature implied by `recvcounts[i]`, `recvtype` at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 4.6.

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

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

4.5.1 Examples using MPI_GATHER, MPI_GATHERV

Example 4.2 Gather 100 ints from every process in group to root. See figure 4.2.

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

Example 4.3 Previous example modified – only the root allocates memory for the receive buffer.

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

Example 4.4 Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of `gsize*100` ints since type matching is defined pairwise between the root and each process in the gather.

```

1
2
3
4
5     MPI_Comm comm;
6     int gsize, sendarray[100];
7     int root, *rbuf;
8     MPI_Datatype rtype;
9     ...
10    MPI_Comm_size( comm, &gsize);
11    MPI_Type_contiguous( 100, MPI_INT, &rtype );
12    MPI_Type_commit( &rtype );
13    rbuf = (int *)malloc(gsize*100*sizeof(int));
14    MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);

```

Example 4.5 Now have each process send 100 ints to root, but place each set (of 100) *stride* ints apart at receiving end. Use `MPI_GATHERV` and the `displs` argument to achieve this effect. Assume *stride* \geq 100. See figure 4.3.

```

19
20    MPI_Comm comm;
21    int gsize, sendarray[100];
22    int root, *rbuf, stride;
23    int *displs, i, *rcounts;
24
25    ...
26
27    MPI_Comm_size( comm, &gsize);
28    rbuf = (int *)malloc(gsize*stride*sizeof(int));
29    displs = (int *)malloc(gsize*sizeof(int));
30    rcounts = (int *)malloc(gsize*sizeof(int));
31    for (i=0; i<gsize; ++i) {
32        displs[i] = i*stride;
33        rcounts[i] = 100;
34    }
35    MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
36                root, comm);

```

Note that the program is erroneous if *stride* $<$ 100.

Example 4.6 Same as Example 4.5 on the receiving side, but send the 100 ints from the 0th column of a 100 \times 150 int array, in C. See figure 4.4.

```

43    MPI_Comm comm;
44    int gsize, sendarray[100][150];
45    int root, *rbuf, stride;
46    MPI_Datatype stype;
47    int *displs, i, *rcounts;
48

```

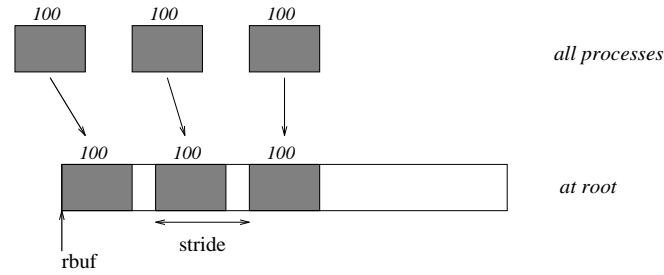


Figure 4.3: The root process gathers 100 ints from each process in the group, each set is placed `stride` ints apart.

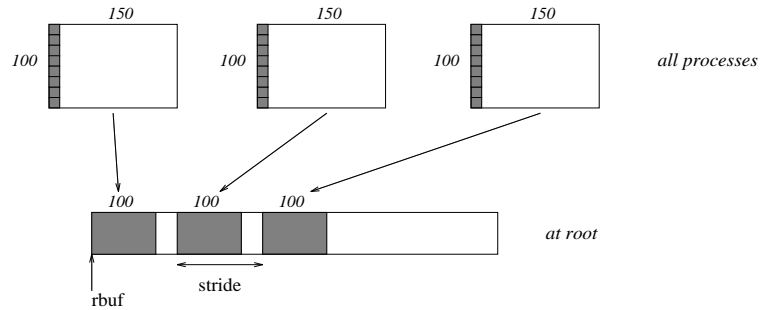


Figure 4.4: The root process gathers column 0 of a 100×150 C array, and each set is placed `stride` ints apart.

...

```

MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    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 4.7 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 4.5.

```

MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;

```

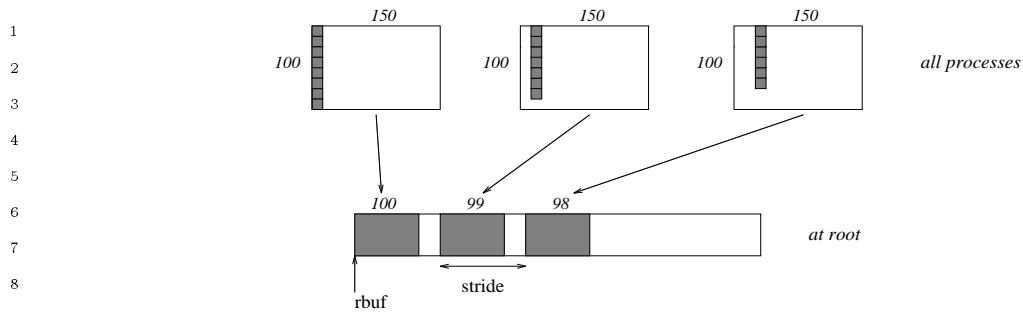


Figure 4.5: The root process gathers 100-i ints from column i of a 100×150 C array, and each set is placed `stride` ints apart.

```

13     int *displs,i,*rcounts;
14
15     ...
16
17     MPI_Comm_size( comm, &gsize);
18     MPI_Comm_rank( comm, &myrank );
19     rbuf = (int *)malloc(gsize*stride*sizeof(int));
20     displs = (int *)malloc(gsize*sizeof(int));
21     rcounts = (int *)malloc(gsize*sizeof(int));
22     for (i=0; i<gsize; ++i) {
23         displs[i] = i*stride;
24         rcounts[i] = 100-i;      /* note change from previous example */
25     }
26     /* Create datatype for the column we are sending
27      */
28     MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
29     MPI_Type_commit( &stype );
30     /* sptr is the address of start of "myrank" column
31      */
32     sptr = &sendarray[0][myrank];
33     MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
34     root, comm);
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```

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

Example 4.8 Same as Example 4.7, but done in a different way at the sending end. We create a datatype that causes the correct striding at the sending end so that that we read a column of a C array. A similar thing was done in Example 3.33, Section 3.12.7.

```

43     MPI_Comm comm;
44     int gsize,sendarray[100][150],*sptr;
45     int root, *rbuf, stride, myrank, disp[2], blocklen[2];
46     MPI_Datatype stype,type[2];
47     int *displs,i,*rcounts;
48

```

```

...
1
2
MPI_Comm_size( comm, &gsize);
3
MPI_Comm_rank( comm, &myrank );
4
rbuf = (int *)malloc(gsize*stride*sizeof(int));
5
displs = (int *)malloc(gsize*sizeof(int));
6
rcounts = (int *)malloc(gsize*sizeof(int));
7
for (i=0; i<gsize; ++i) {
8
    displs[i] = i*stride;
9
    rcounts[i] = 100-i;
10
}
11
/* Create datatype for one int, with extent of entire row
*/
12
disp[0] = 0;      disp[1] = 150*sizeof(int);
13
type[0] = MPI_INT; type[1] = MPI_UB;
14
blocklen[0] = 1;  blocklen[1] = 1;
15
MPI_Type_struct( 2, blocklen, disp, type, &stype );
16
MPI_Type_commit( &stype );
17
sptr = &sendarray[0][myrank];
18
MPI_Gatherv( sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
19
                                                    root, comm);
20
21
22

```

Example 4.9 Same as Example 4.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See figure 4.6.

```

23
24
25
MPI_Comm comm;
26
int gsize,sendarray[100][150],*sptr;
27
int root, *rbuf, *stride, myrank, bufsize;
28
MPI_Datatype stype;
29
int *displs,i,*rcounts,offset;
30
31
...
32
33
MPI_Comm_size( comm, &gsize);
34
MPI_Comm_rank( comm, &myrank );
35
36
stride = (int *)malloc(gsize*sizeof(int));
37
...
38
/* stride[i] for i = 0 to gsize-1 is set somehow
*/
39
40
41
/* set up displs and rcounts vectors first
*/
42
displs = (int *)malloc(gsize*sizeof(int));
43
rcounts = (int *)malloc(gsize*sizeof(int));
44
offset = 0;
45
for (i=0; i<gsize; ++i) {
46
    displs[i] = offset;
47
48

```

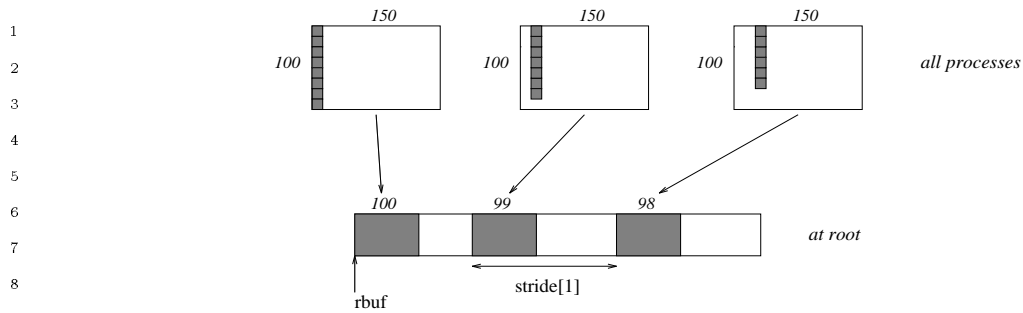


Figure 4.6: The root process gathers 100- i ints from column i of a 100×150 C array, and each set is placed `stride[i]` ints apart (a varying stride).

```

    offset += stride[i];
    rcounts[i] = 100-i;
}
/* the required buffer size for rbuf is now easily obtained
*/
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
/* Create datatype for the column we are sending
*/
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
              root, comm);

```

Example 4.10 Process i sends `num` ints from the i th column of a 100×150 int array, in C. The complicating factor is that the various values of `num` are not known to `root`, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

```

MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root,*rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,types[2];
int *displs,i,*rcounts,num;

...

MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );

/* First, gather nums to root
*/
rcounts = (int *)malloc(gsize*sizeof(int));
MPI_Gather( &num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);

```

```

/* root now has correct rcounts, using these we set displs[] so
 * that data is placed contiguously (or concatenated) at receive end
 */
displs = (int *)malloc(gsize*sizeof(int));
displs[0] = 0;
for (i=1; i<gsize; ++i) {
    displs[i] = displs[i-1]+rcounts[i-1];
}
/* And, create receive buffer
 */
rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
 *sizeof(int));
/* Create datatype for one int, with extent of entire row
 */
disp[0] = 0;      disp[1] = 150*sizeof(int);
type[0] = MPI_INT; type[1] = MPI_UB;
blocklen[0] = 1;  blocklen[1] = 1;
MPI_Type_struct( 2, blocklen, disp, type, &stype );
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
              root, comm);

```

4.6 Scatter

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

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcount	number of elements sent to each process (integer, significant only at root)
IN	sendtype	data type of send buffer elements (significant only at root) (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

```

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

```

```

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
            ROOT, COMM, IERROR)

```



```
1 <type> SENDBUF(*), RECVBUF(*)
```

```
2 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVMYPE, ROOT, COMM, IERROR
```

```
3 MPI_SCATTER is the inverse operation to MPI_GATHER.
```

```
4 The outcome is as if the root executed n send operations,
```

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

```
6 and each process executed a receive,
```

```
7 MPI_Recv(recvbuf, recvcoun, recvmype, i, ...).
```

```
8
9 An alternative description is that the root sends a message with MPI_Send(sendbuf,
10 sendcount·n, sendtype, ...). This message is split into n equal segments, the ith segment is
11 sent to the ith process in the group, and each process receives this message as above.
```

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

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

```
18 All arguments to the function are significant on process root, while on other processes,
19 only arguments recvbuf, recvcoun, recvmype, root, comm are significant. The arguments root
20 and comm must have identical values on all processes.
```

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

```
23
24 Rationale. Though not needed, the last restriction is imposed so as to achieve
25 symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write
26 restriction) is necessary. (End of rationale.)
```

```
27
28
29 MPI_SCATTERV( sendbuf, sendcounts, displs, sendtype, recvbuf, recvcoun, recvmype, root,
30 comm)
```

```
31
32 IN      sendbuf      address of send buffer (choice, significant only at root)
33 IN      sendcounts   integer array (of length group size) specifying the num-
34                      ber of elements to send to each processor
35 IN      displs       integer array (of length group size). Entry i specifies
36                      the displacement (relative to sendbuf from which to
37                      take the outgoing data to process i
38 IN      sendtype     data type of send buffer elements (handle)
39 OUT     recvbuf      address of receive buffer (choice)
40 IN      recvcoun     number of elements in receive buffer (integer)
41 IN      recvmype     data type of receive buffer elements (handle)
42 IN      root         rank of sending process (integer)
43 IN      comm         communicator (handle)
```

```
44
45
46
47 int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
48 MPI_Datatype sendtype, void* recvbuf, int recvcoun,
```

```

        MPI_Datatype recvtype, int root, MPI_Comm comm)
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
        RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
COMM, IERROR

```

`MPI_SCATTERV` is the inverse operation to `MPI_GATHERV`.

`MPI_SCATTERV` extends the functionality of `MPI_SCATTER` by allowing a varying 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 the new argument, `displs`.

The outcome is as if the root executed `n` send operations,

```
MPI_Send(sendbuf + displs[i] * extent(sendtype), sendcounts[i], sendtype, i, ...),
```

and each process executed a receive,

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

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

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

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

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

4.6.1 Examples using `MPI_SCATTER`, `MPI_SCATTERV`

Example 4.11 The reverse of Example 4.2. Scatter sets of 100 ints from the root to each process in the group. See figure 4.7.

```

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

```

Example 4.12 The reverse of Example 4.5. 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 4.8.

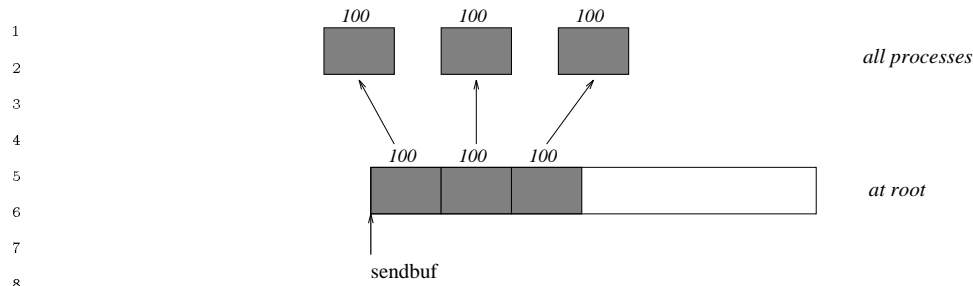


Figure 4.7: The root process scatters sets of 100 ints to each process in the group.

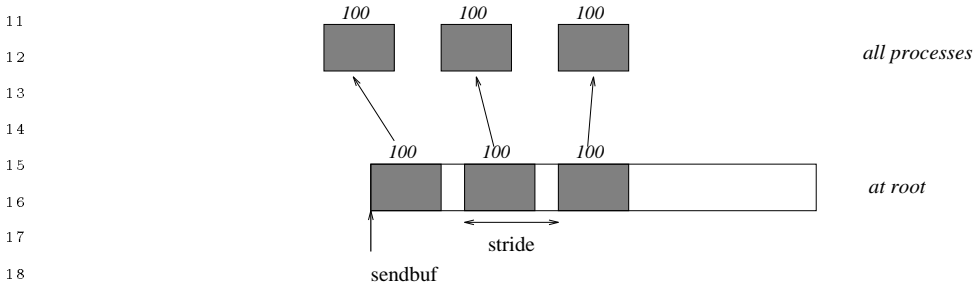


Figure 4.8: The root process scatters sets of 100 ints, moving by `stride` ints from send to send in the scatter.

```

23 MPI_Comm comm;
24 int gsize,*sendbuf;
25 int root, rbuf[100], i, *displs, *counts;
26
27 ...
28
29 MPI_Comm_size( comm, &gsize);
30 sendbuf = (int *)malloc(gsize*stride*sizeof(int));
31 ...
32 displs = (int *)malloc(gsize*sizeof(int));
33 counts = (int *)malloc(gsize*sizeof(int));
34 for (i=0; i<gsize; ++i) {
35     displs[i] = i*stride;
36     counts[i] = 100;
37 }
38 MPI_Scatterv( sendbuf, counts, displs, MPI_INT, rbuf, 100, MPI_INT,
39              root, comm);

```

Example 4.13 The reverse of Example 4.9. 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 4.9.

```

45 MPI_Comm comm;
46 int gsize,recvarray[100][150],*rptra;
47 int root, *sendbuf, myrank, bufsize, *stride;
48

```

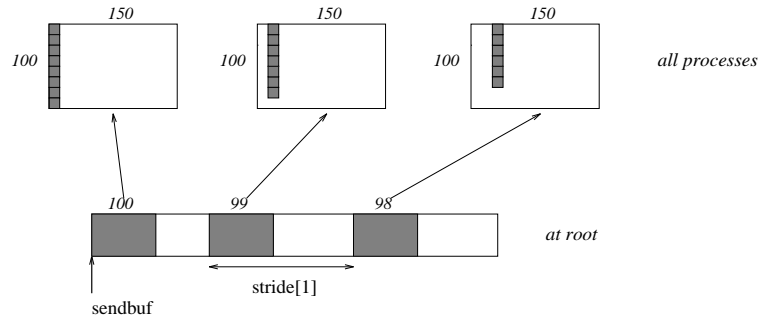


Figure 4.9: The root scatters blocks of 100- i ints into column i of a 100×150 C array. At the sending side, the blocks are `stride[i]` ints apart.

```

MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 * sendbuf comes from elsewhere
 */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit( &rtype );
rptr = &recvarray[0][myrank];
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
              root, comm);

```

4.7 Gather-to-all

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

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (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 (integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```

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

MPI_ALLGATHER can be thought of as **MPI_GATHER**, but where all processes receive the result, instead of just the root. The *j*th block of data sent from each process is received by every process and placed in the *j*th block of the buffer **recvbuf**.

The type signature associated with **sendcount**, **sendtype**, at a process must be equal to the type signature associated with **recvcount**, **recvtype** at any other process.

The outcome of a call to **MPI_ALLGATHER(...)** is as if all processes executed *n* calls to

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

for **root** = 0 , . . . , *n*-1. The rules for correct usage of **MPI_ALLGATHER** are easily found from the corresponding rules for **MPI_GATHER**.

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcoun- 1

IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcount	number of elements in send buffer (integer)	4
IN	sendtype	data type of send buffer elements (handle)	5
OUT	recvbuf	address of receive buffer (choice)	6
IN	recvcoun- 7	integer array (of length group size) containing the num- 8	9
		ber of elements that are received from each process 10	
IN	displs	integer array (of length group size). Entry <i>i</i> specifies 11	12
		the displacement (relative to <i>recvbuf</i>) at which to place 13	
		the incoming data from process <i>i</i>	
IN	recvtype	data type of receive buffer elements (handle)	14
IN	comm	communicator (handle)	15

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, 18
                  void* recvbuf, int *recvcoun- 19
                  MPI_Datatype recvtype, MPI_Comm comm) 20
```

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 21
                RECVTYPE, COMM, IERROR) 22
<type> SENDBUF(*), RECVBUF(*) 23
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, 24
IERROR 25
```

MPI_ALLGATHERV can be thought of as **MPI_GATHERV**, but where all processes receive 27
the result, instead of just the root. The *j*th block of data sent from each process is received 28
by every process and placed in the *j*th block of the buffer *recvbuf*. These blocks need not 29
all be the same size. 30

The type signature associated with *sendcount*, *sendtype*, at process *j* must be equal to 31
the type signature associated with *recvcoun- 32*

The outcome is as if all processes executed calls to 33

```
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcoun- 34
            recvtype, root, comm), 35
```

for *root* = 0 , ... , *n*-1. The rules for correct usage of **MPI_ALLGATHERV** are easily 37
found from the corresponding rules for **MPI_GATHERV**. 38

4.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV 40

Example 4.14 The all-gather version of Example 4.2. Using **MPI_ALLGATHER**, we will 41
gather 100 ints from every process in the group to every process. 42

```
MPI_Comm comm; 44
int gsize, sendarray[100]; 45
int *rbuf; 46
... 47
48
```

```

1      MPI_Comm_size( comm, &gsize);
2      rbuf = (int *)malloc(gsize*100*sizeof(int));
3      MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);

```

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

6

7 4.8 All-to-All Scatter/Gather

8

9

10

```

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

```

12	IN	sendbuf	starting address of send buffer (choice)
13	IN	sendcount	number of elements sent to each process (integer)
14	IN	sendtype	data type of send buffer elements (handle)
15	IN	sendtype	data type of send buffer elements (handle)
16	OUT	recvbuf	address of receive buffer (choice)
17	IN	recvcount	number of elements received from any process (integer)
18	IN	recvcount	number of elements received from any process (integer)
19	IN	recvtype	data type of receive buffer elements (handle)
20	IN	recvtype	data type of receive buffer elements (handle)
21	IN	comm	communicator (handle)
22			

```

23 int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
24                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
25                 MPI_Comm comm)

```

```

26
27 MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
28              COMM, IERROR)
29 <type> SENDBUF(*), RECVBUF(*)
30 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

```

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

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

38 The outcome is as if each process executed a send to each process (itself included) with
39 a call to,

```

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

```

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

```

44 MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount, i, ...).

```

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

48

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun-
ts, rdispls, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)	
IN	sendcounts	integer array equal to the group size specifying the number of elements to send to each processor	
IN	sdispls	integer array (of length group size). Entry <i>j</i> specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process <i>j</i>	
IN	sendtype	data type of send buffer elements (handle)	
OUT	recvbuf	address of receive buffer (choice)	
IN	recvcoun-	integer array equal to the group size specifying the number of elements that can be received from each processor	
IN	rdispls	integer array (of length group size). Entry <i>i</i> specifies the displacement (relative to recvbuf at which to place the incoming data from process <i>i</i>	
IN	recvtype	data type of receive buffer elements (handle)	
IN	comm	communicator (handle)	

```
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
                  MPI_Datatype sendtype, void* recvbuf, int *recvcoun-
                  ts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
               RDISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
RECVTYPE, COMM, IERROR
```

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**.

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

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

```
MPI_Send(sendbuf + displs[i] · extent(sendtype), sendcounts[i], sendtype, i, ...),
```

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

```
MPI_Recv(recvbuf + displs[i] · extent(recvtype), recvcoun-
ts[i], recvtype, i, ...).
```

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

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.*)

4.9 Global Reduction Operations

The functions in this section perform a global reduce operation (such as sum, max, logical AND, etc.) across all the members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction at one node, an all-reduce that returns this result at all nodes, and a scan (parallel prefix) operation. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

4.9.1 Reduce

`MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)`

IN	sendbuf	address of send buffer (choice)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	count	number of elements in send buffer (integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	reduce operation (handle)
IN	root	rank of root process (integer)
IN	comm	communicator (handle)

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

`MPI_REDUCE` combines the elements provided in the input buffer of each process in the group, using the operation `op`, and returns the combined value in the output buffer of the process with rank `root`. The input buffer is defined by the arguments `sendbuf`, `count` and `datatype`; the output buffer is defined by the arguments `recvbuf`, `count` and `datatype`; both have the same number of elements, with the same type. The routine is called by all

group members using the same arguments for `count`, `datatype`, `op`, `root` and `comm`. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is `MPI_MAX` and the send buffer contains two elements that are floating point numbers (`count = 2` and `datatype = MPI_FLOAT`), then `recvbuf(1) = global max(sendbuf(1))` and `recvbuf(2) = global max(sendbuf(2))`.

Sec. 4.9.2, lists the set of predefined operations provided by `MPI`. That section also enumerates the datatypes each operation can be applied to. 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 Sec. 4.9.4.

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

The `datatype` argument of `MPI_REDUCE` must be compatible with `op`. Predefined operators work only with the `MPI` types listed in Sec. 4.9.2 and Sec. 4.9.3. 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 4.9.4.

4.9.2 Predefined reduce operations

The following predefined operations are supplied for `MPI_REDUCE` and related functions `MPI_ALLREDUCE`, `MPI_REDUCE_SCATTER`, and `MPI_SCAN`. These operations are invoked by placing the following in `op`.

Name	Meaning
<code>MPI_MAX</code>	maximum
<code>MPI_MIN</code>	minimum
<code>MPI_SUM</code>	sum
<code>MPI_PROD</code>	product
<code>MPI_LAND</code>	logical and
<code>MPI_BAND</code>	bit-wise and
<code>MPI_LOR</code>	logical or
<code>MPI BOR</code>	bit-wise or
<code>MPI_LXOR</code>	logical xor

1	MPI_BXOR	bit-wise xor
2	MPI_MAXLOC	max value and location
3	MPI_MINLOC	min value and location

4 The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Sec.
5 4.9.3. For the other predefined operations, we enumerate below the allowed combinations
6 of **op** and **datatype** arguments. First, define groups of MPI basic datatypes in the following
7 way.

10	C integer:	MPI_INT, MPI_LONG, MPI_SHORT,
11		MPI_UNSIGNED_SHORT, MPI_UNSIGNED,
12		MPI_UNSIGNED_LONG
13	Fortran integer:	MPI_INTEGER
14	Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL,
15		MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE
16	Logical:	MPI_LOGICAL
17	Complex:	MPI_COMPLEX
18	Byte:	MPI_BYTE

19 Now, the valid datatypes for each option is specified below.

22	Op	Allowed Types
24	MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point
25	MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex
26	MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical
27	MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte

28 **Example 4.15** A routine that computes the dot product of two vectors that are distributed
29 across a group of processes and returns the answer at node zero.

```

31 SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
32 REAL a(m), b(m)           ! local slice of array
33 REAL c                    ! result (at node zero)
34 REAL sum
35 INTEGER m, comm, i, ierr
36
37 ! local sum
38 sum = 0.0
39 DO i = 1, m
40     sum = sum + a(i)*b(i)
41 END DO
42
43 ! global sum
44 CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
45 RETURN
46
```

47 **Example 4.16** A routine that computes the product of a vector and an array that are
48 distributed across a group of processes and returns the answer at node zero.

```

SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)      ! local slice of array
REAL c(n)               ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
    sum(j) = 0.0
    DO i = 1, m
        sum(j) = sum(j) + a(i)*b(i,j)
    END DO
END DO

! global sum
CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)

! return result at node zero (and garbage at the other nodes)
RETURN

```

4.9.3 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:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

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:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

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 $(u_0, 0), (u_1, 1), \dots, (u_{n-1}, n-1)$, 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 according to the first component of each pair, and ties are resolved according to the second component.

The reduce operation is defined to operate on arguments that consist of a pair: value and index. For both Fortran and C, types are provided to describe the pair. The potentially 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, and coercing the index to this type also. In C, the MPI-provided pair type has distinct types and the index is an `int`.

In order to use `MPI_MINLOC` and `MPI_MAXLOC` in a reduce operation, one must provide a **datatype** argument that represents a pair (value and index). MPI provides seven such predefined datatypes. The operations `MPI_MAXLOC` and `MPI_MINLOC` can be used with each of the following datatypes.

Fortran:

Name	Description
<code>MPI_2REAL</code>	pair of REALs
<code>MPI_2DOUBLE_PRECISION</code>	pair of DOUBLE PRECISION variables
<code>MPI_2INTEGER</code>	pair of INTEGERS

C:

Name	Description
<code>MPI_FLOAT_INT</code>	float and int
<code>MPI_DOUBLE_INT</code>	double and int
<code>MPI_LONG_INT</code>	long and int
<code>MPI_2INT</code>	pair of int
<code>MPI_SHORT_INT</code>	short and int
<code>MPI_LONG_DOUBLE_INT</code>	long double and int

The datatype `MPI_2REAL` is *as if* defined by the following (see Section 3.12).

```
MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)
```

Similar statements apply for `MPI_2INTEGER`, `MPI_2DOUBLE_PRECISION`, and `MPI_2INT`.

The datatype `MPI_FLOAT_INT` is *as if* defined by the following sequence of instructions.

```

type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
disp[1] = sizeof(float)
block[0] = 1
block[1] = 1
MPI_TYPE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)

```

Similar statements apply for MPI_LONG_INT and MPI_DOUBLE_INT.

Example 4.17 Each process has an array of 30 doubles, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

```

...
/* each process has an array of 30 double: ain[30]
*/
double ain[30], aout[30];
int ind[30];
struct {
    double val;
    int rank;
} in[30], out[30];
int i, myrank, root;

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
for (i=0; i<30; ++i) {
    in[i].val = ain[i];
    in[i].rank = myrank;
}
MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );
/* At this point, the answer resides on process root
*/
if (myrank == root) {
    /* read ranks out
    */
    for (i=0; i<30; ++i) {
        aout[i] = out[i].val;
        ind[i] = out[i].rank;
    }
}

```

Example 4.18 Same example, in Fortran.

```

...
! each process has an array of 30 double: ain(30)

DOUBLE PRECISION ain(30), aout(30)
INTEGER ind(30);
DOUBLE PRECISION in(2,30), out(2,30)

```

```

1      INTEGER i, myrank, root, ierr;
2
3      MPI_COMM_RANK(MPI_COMM_WORLD, myrank);
4      DO I=1, 30
5          in(1,i) = ain(i)
6          in(2,i) = myrank      ! myrank is coerced to a double
7      END DO
8
9      MPI_REDUCE( in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,
10                                     comm, ierr );
11      ! At this point, the answer resides on process root
12
13      IF (myrank .EQ. root) THEN
14          ! read ranks out
15          DO I= 1, 30
16              aout(i) = out(1,i)
17              ind(i) = out(2,i) ! rank is coerced back to an integer
18          END DO
19      END IF

```

Example 4.19 Each process has a non-empty array of values. Find the minimum global value, the rank of the process that holds it and its index on this process.

```

23      #define LEN    1000
24
25      float val[LEN];          /* local array of values */
26      int count;               /* local number of values */
27      int myrank, minrank, minindex;
28      float minval;
29
30      struct {
31          float value;
32          int index;
33      } in, out;
34
35          /* local minloc */
36      in.value = val[0];
37      in.index = 0;
38      for (i=1; i < count; i++)
39          if (in.value > val[i]) {
40              in.value = val[i];
41              in.index = i;
42          }
43
44          /* global minloc */
45      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
46      in.index = myrank*LEN + in.index;
47      MPI_Reduce( in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
48

```

```

    /* At this point, the answer resides on process root
    */
if (myrank == root) {
    /* read answer out
    */
    minval = out.value;
    minrank = out.index / LEN;
    minindex = out.index % LEN;
}

```

Rationale. The definition of `MPI_MINLOC` and `MPI_MAXLOC` given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of `MPI_MAXLOC` and `MPI_MINLOC`, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

4.9.4 User-Defined Operations

`MPI_OP_CREATE(function, commute, op)`

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

```
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
```

```
MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
```

```

EXTERNAL FUNCTION
LOGICAL COMMUTE
INTEGER OP, IERROR

```

`MPI_OP_CREATE` binds a user-defined global operation to an `op` handle that can subsequently be used in `MPI_REDUCE`, `MPI_ALLREDUCE`, `MPI_REDUCE_SCATTER`, and `MPI_SCAN`. The user-defined operation is assumed to be associative. If `commute = true`, then the operation should be both commutative and associative. If `commute = false`, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, taking advantage of the associativity of the operation. If `commute = true` then the order of evaluation can be changed, taking advantage of commutativity and associativity.

`function` is the user-defined function, which must have the following four arguments: `invec`, `inoutvec`, `len` and `datatype`.

The ANSI-C prototype for the function is the following.

```

typedef void MPI_User_function( void *invec, void *inoutvec, int *len,
                                MPI_Datatype *datatype);

```

The Fortran declaration of the user-defined function appears below.


```

1  FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
2  <type> INVEC(LEN), INOUTVEC(LEN)
3  INTEGER LEN, TYPE

```

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

Informally, we can think of **invec** and **inoutvec** as arrays of **len** elements that **function** 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 $\text{invec}[i] \circ \text{inoutvec}[i]$, for $i = 0, \dots, \text{count} - 1$, 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.

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

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

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

```

if (rank > 0) {
    RECV(tempbuf, count, datatype, rank-1,...)
    User_reduce( tempbuf, sendbuf, count, datatype)
}
if (rank < groupsize-1) {
    SEND( sendbuf, count, datatype, rank+1, ...)
}
/* answer now resides in process groupsize-1 ... now send to root
*/
if (rank == groupsize-1) {
    SEND( sendbuf, count, datatype, root, ...)
}
if (rank == root) {
    RECV(recvbuf, count, datatype, groupsize-1,...)
}

```

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 non-commutative operator defined by the function `User_reduce()`. A more efficient implementation is achieved by taking advantage of associativity and using a logarithmic tree reduction. Commutativity can be used to advantage, for those cases in which the `commute` argument to `MPI_OP_CREATE` is true. Also, the amount of temporary buffer required can be reduced, and communication can be pipelined with computation, by transferring and reducing the elements in chunks of size `len < count`.

The predefined reduce operations can be implemented as a library of user-defined operations. However, better performance might be achieved if `MPI_REDUCE` handles these functions as a special case. (*End of advice to implementors.*)

`MPI_OP_FREE(op)`

IN op operation (handle)

```
int MPI_op_free( MPI_Op *op)
```

```
MPI_OP_FREE( OP, IERROR)
```

```
INTEGER OP, IERROR
```

Marks a user-defined reduction operation for deallocation and sets `op` to `MPI_OP_NULL`.

Example of User-defined Reduce

It is time for an example of user-defined reduction.

Example 4.20 Compute the product of an array of complex numbers, in C.

```

1  typedef struct {
2      double real,imag;
3  } Complex;
4
5  /* the user-defined function
6   */
7  void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr )
8  {
9      int i;
10     Complex c;
11
12     for (i=0; i< *len; ++i) {
13         c.real = inout->real*in->real -
14             inout->imag*in->imag;
15         c.imag = inout->real*in->imag +
16             inout->imag*in->real;
17         *inout = c;
18         in++; inout++;
19     }
20 }
21
22 /* and, to call it...
23  */
24 ...
25
26     /* each process has an array of 100 Complexes
27     */
28     Complex a[100], answer[100];
29     MPI_Op myOp;
30     MPI_Datatype ctype;
31
32     /* explain to MPI how type Complex is defined
33     */
34     MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
35     MPI_Type_commit( &ctype );
36     /* create the complex-product user-op
37     */
38     MPI_Op_create( myProd, True, &myOp );
39
40     MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );
41
42     /* At this point, the answer, which consists of 100 Complexes,
43     * resides on process root
44     */
45
46
47
48

```

4.9.5 All-Reduce

MPI includes variants of each of the reduce operations where the result is returned to all processes in the group. MPI requires that all processes participating in these operations receive identical results.

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in send buffer (integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

Same as **MPI_REDUCE** except that the result appears in the receive buffer of all the group members.

Advice to implementors. The all-reduce operations can be implemented as a reduce, followed by a broadcast. However, a direct implementation can lead to better performance. (*End of advice to implementors.*)

Example 4.21 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 4.16).

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)      ! local slice of array
REAL c(n)              ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
    sum(j) = 0.0
    DO i = 1, m
        sum(j) = sum(j) + a(i)*b(i,j)
    END DO
END DO

! global sum
```

```

1  CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
2
3  ! return result at all nodes
4  RETURN
5

```

4.10 Reduce-Scatter

MPI includes variants of each of the reduce operations where the result is scattered to all processes in the group on return.

```

12 MPI_REDUCE_SCATTER( sendbuf, recvbuf, recvcunts, datatype, op, comm)

```

13	IN	sendbuf	starting address of send buffer (choice)
14	OUT	recvbuf	starting address of receive buffer (choice)
15			
16	IN	recvcunts	integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
17			
18			
19	IN	datatype	data type of elements of input buffer (handle)
20	IN	op	operation (handle)
21			
22	IN	comm	communicator (handle)
23			

```

24 int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcunts,
25                        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

```

```

26 MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
27                   IERROR)

```

```

28     <type> SENDBUF(*), RECVBUF(*)
29     INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
30

```

MPI_REDUCE_SCATTER first does an element-wise reduction on vector of **count** = $\sum_i \text{recvcunts}[i]$ elements in the send buffer defined by **sendbuf**, **count** and **datatype**. Next, the resulting vector of results is split into **n** disjoint segments, where **n** is the number of members in the group. Segment **i** contains **recvcunts**[**i**] elements. The **i**th segment is sent to process **i** and stored in the receive buffer defined by **recvbuf**, **recvcunts**[**i**] and **datatype**.

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

4.11 Scan

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

MPI_SCAN is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank *i*, the reduction of the values in the send buffers of processes with ranks $0, \dots, i$ (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers are as for **MPI_REDUCE**.

Rationale. We have defined an inclusive scan, that is, the prefix reduction on process *i* includes the data from process *i*. An alternative is to define scan in an exclusive manner, where the result on *i* only includes data up to *i*-1. Both definitions are useful. The latter has some advantages: the inclusive scan can always be computed from the exclusive scan with no additional communication; for non-invertible operations such as max and min, communication is required to compute the exclusive scan from the inclusive scan. There is, however, a complication with exclusive scan since one must define the “unit” element for the reduction in this case. That is, one must explicitly say what occurs for process 0. This was thought to be complex for user-defined operations and hence, the exclusive scan was dropped. (*End of rationale.*)

4.11.1 Example using MPI_SCAN

Example 4.22 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:

<i>values</i>	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
<i>logicals</i>	0	0	1	1	1	0	0	1
<i>result</i>	v_1	$v_1 + v_2$	v_3	$v_3 + v_4$	$v_3 + v_4 + v_5$	v_6	$v_6 + v_7$	v_8

The operator that produces this effect is,

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

where,

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

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

```

11 typedef struct {
12     double val;
13     int log;
14 } SegScanPair;
15
16 /* the user-defined function
17 */
18 void segScan( SegScanPair *in, SegScanPair *inout, int *len,
19 MPI_Datatype *dptr )
20 {
21     int i;
22     SegScanPair c;
23
24     for (i=0; i< *len; ++i) {
25         if ( in->log == inout->log )
26             c.val = in->val + inout->val;
27         else
28             c.val = inout->val;
29         c.log = inout->log;
30         *inout = c;
31         in++; inout++;
32     }
33 }
34

```

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

```

39 int i,base;
40 SeqScanPair a, answer;
41 MPI_Op myOp;
42 MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
43 MPI_Aint disp[2];
44 int blocklen[2] = { 1, 1};
45 MPI_Datatype sspair;
46
47 /* explain to MPI how type SegScanPair is defined
48 */

```

```

MPI_Address( a, disp);
MPI_Address( a.log, disp+1);
base = disp[0];
for (i=0; i<2; ++i) disp[i] -= base;
MPI_Type_struct( 2, blocklen, disp, type, &sspair );
MPI_Type_commit( &sspair );
/* create the segmented-scan user-op
*/
MPI_Op_create( segScan, False, &myOp );
...
MPI_Scan( a, answer, 1, spair, myOp, root, comm );

```

4.12 Correctness

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.

Example 4.23 The following is erroneous.

```

switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
        break;
    case 1:
        MPI_Bcast(buf2, count, type, 1, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}

```

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

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

Example 4.24 The following is erroneous.

```

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

```



```

1      MPI_Bcast(buf1, count, type, 2, comm2);
2      MPI_Bcast(buf2, count, type, 1, comm1);
3      break;
4  }

```

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

Collective operations must be executed in an order so that no cyclic dependences occur.

Example 4.25 The following is erroneous.

```

14  switch(rank) {
15      case 0:
16          MPI_Bcast(buf1, count, type, 0, comm);
17          MPI_Send(buf2, count, type, 1, tag, comm);
18          break;
19      case 1:
20          MPI_Recv(buf2, count, type, 0, tag, comm, status);
21          MPI_Bcast(buf1, count, type, 0, comm);
22          break;
23  }

```

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 4.26 A correct, but non-deterministic program.

```

37  switch(rank) {
38      case 0:
39          MPI_Bcast(buf1, count, type, 0, comm);
40          MPI_Send(buf2, count, type, 1, tag, comm);
41          break;
42      case 1:
43          MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
44          MPI_Bcast(buf1, count, type, 0, comm);
45          MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
46          break;
47      case 2:
48          MPI_Send(buf2, count, type, 1, tag, comm);

```

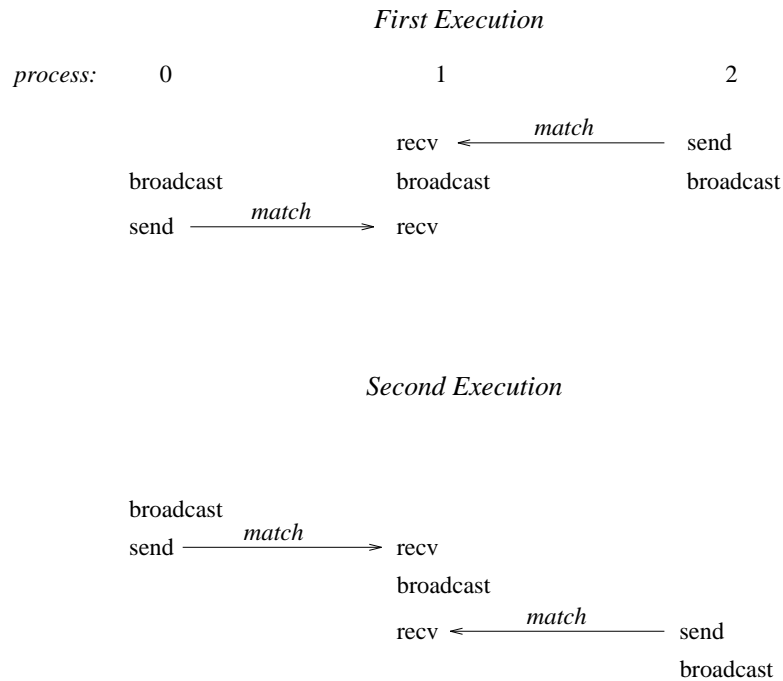


Figure 4.10: A race condition causes non-deterministic matching of sends and receives. One cannot rely on synchronization from a broadcast to make the program deterministic.

```

MPI_Bcast(buf1, count, type, 0, comm);
break;
}

```

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

Two possible executions of this program, with different matchings of sends and receives, are illustrated in figure 4.10. Note that the second execution has the peculiar effect that a send executed after the broadcast is received at another node before the broadcast. This example illustrates the fact that one should not rely on collective communication functions to have particular synchronization effects. A program that works correctly only when the first execution occurs (only when broadcast is synchronizing) is erroneous.

Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication call at a process. In these situations, it is the user's responsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.

Advice to implementors. Assume that broadcast is implemented using point-to-point MPI communication. Suppose the following two rules are followed.

1. All receives specify their source explicitly (no wildcards).
2. Each process sends all messages that pertain to one collective call before sending any message that pertain to a subsequent collective call.

1 Then, messages belonging to successive broadcasts cannot be confused, as the order
2 of point-to-point messages is preserved.

3 It is the implementor's responsibility to ensure that point-to-point messages are not
4 confused with collective messages. One way to accomplish this is, whenever a commu-
5 nicator is created, to also create a "hidden communicator" for collective communica-
6 tion. One could achieve a similar effect more cheaply, for example, by using a hidden
7 tag or context bit to indicate whether the communicator is used for point-to-point or
8 collective communication. (*End of advice to implementors.*)

Chapter 5

Groups, Contexts, and Communicators

5.1 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 [26] and [3] for further information on writing libraries in MPI, using the features described in this chapter.

5.1.1 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.

5.1.2 MPI's Support for Libraries

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

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

Communicators (see [16, 24, 27]) encapsulate all of these ideas in order to provide the appropriate scope for all communication operations in MPI. Communicators are divided into two kinds: intra-communicators for operations within a single group of processes, and inter-communicators, for point-to-point communication between two groups of processes.

Caching. Communicators (see below) provide a “caching” mechanism that allows one to associate new attributes with communicators, on a par with MPI built-in features. This can be used by advanced users to adorn communicators further, and by MPI to implement some communicator functions. For example, the virtual-topology functions described in Chapter 6 are likely to be supported this way.

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.

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

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

- 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 6 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 current 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. There is no general-purpose collective communication on inter-communicators, so contexts are used just to isolate point-to-point communication.
- 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 communication in an related manner to intra-communicators. Users who do not need inter-communication in their applications can safely ignore this extension. Users who need collective operations via inter-communicators must layer it on top of MPI. Users who require inter-communication between overlapping groups must also layer this capability on top of MPI.

5.2 Basic Concepts

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

5.2.1 Groups

A **group** is an ordered set of process identifiers (henceforth processes); processes are implementation-dependent objects. Each process in a group is associated with an integer **rank**. Ranks are contiguous and start from zero. Groups are represented by opaque **group objects**, and hence cannot be directly transferred from one process to another. A group is used within a communicator to describe the participants in a communication “universe” and to rank such participants (thus giving them unique names within that “universe” 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 process-address-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.*)

5.2.2 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-to-point operations. Contexts are not explicit MPI objects; they appear only as part of the realization of communicators (below).

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). Communicator-generating functions use a collective communication to agree on a new group-wide unique context.

Analogously, in inter-communication (which is strictly point-to-point 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.*)

5.2.3 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 6), communicators may also “cache” additional information (see section 5.7). 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.

5.2.4 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` 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 computation are available after MPI is initialized. For this case, `MPI_COMM_WORLD` is a communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynamically join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, `MPI_COMM_WORLD` is a communicator incorporating all processes with which the joining process can immediately communicate. Therefore, `MPI_COMM_WORLD` may simultaneously have different values in 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 `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.

5.3 Group Management

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

5.3.1 Group Accessors

`MPI_GROUP_SIZE(group, size)`

IN	group	group (handle)
OUT	size	number of processes in the group (integer)

`int MPI_Group_size(MPI_Group group, int *size)`

`MPI_GROUP_SIZE(GROUP, SIZE, IERROR)`
`INTEGER GROUP, SIZE, IERROR`

`MPI_GROUP_RANK(group, rank)`

IN	group	group (handle)
OUT	rank	rank of the calling process in group, or <code>MPI_UNDEFINED</code> if the process is not a member (integer)

`int MPI_Group_rank(MPI_Group group, int *rank)`

`MPI_GROUP_RANK(GROUP, RANK, IERROR)`
`INTEGER GROUP, RANK, IERROR`

MPI_GROUP_TRANSLATE_RANKS (*group1*, *n*, *ranks1*, *group2*, *ranks2*)

IN	group1	group1 (handle)
IN	n	number of ranks in <i>ranks1</i> and <i>ranks2</i> arrays (integer)
IN	ranks1	array of zero or more valid ranks in <i>group1</i>
IN	group2	group2 (handle)
OUT	ranks2	array of corresponding ranks in <i>group2</i> , MPI_UNDE- FINED when no correspondence exists.

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
                             MPI_Group group2, int *ranks2)
```

```
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
```

This function is important for determining the relative numbering of the same processes in two different groups. For instance, if one knows the ranks of certain processes in the group of **MPI_COMM_WORLD**, one might want to know their ranks in a subset of that group.

MPI_GROUP_COMPARE(*group1*, *group2*, *result*)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	result	result (integer)

```
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)
```

```
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR
```

MPI_IDENT results if the group members and group order is exactly the same in both groups. This happens for instance if **group1** and **group2** are the same handle. **MPI_SIMILAR** results if the group members are the same but the order is different. **MPI_UNEQUAL** results otherwise.

5.3.2 Group Constructors

Group constructors are used to subset and superset existing groups. These constructors construct new groups from existing groups. These are local operations, and distinct groups may be defined on different processes; a process may also define a group that does not include itself. Consistent definitions are required when groups are used as arguments in communicator-building functions. MPI does not provide a mechanism to build a group from scratch, but only from other, previously defined groups. The base group, upon which all other groups are defined, is the group associated with the initial communicator **MPI_COMM_WORLD** (accessible through the function **MPI_COMM_GROUP**).

Rationale. In what follows, there is no group duplication function analogous to **MPI_COMM_DUP**, defined later in this chapter. There is no need for a group duplicator. A group, once created, can have several references to it by making copies of

the handle. The following constructors address the need for subsets and supersets of existing groups. (*End of rationale.*)

Advice to implementors. Each group constructor behaves as if it returned a new group object. When this new group is a copy of an existing group, then one can avoid creating such new objects, using a reference-count mechanism. (*End of advice to implementors.*)

MPI_COMM_GROUP(comm, group)

IN	comm	communicator (handle)
OUT	group	group corresponding to comm (handle)

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

MPI_COMM_GROUP(COMM, GROUP, IERROR)
INTEGER COMM, GROUP, IERROR

MPI_COMM_GROUP returns in **group** a handle to the group of **comm**.

MPI_GROUP_UNION(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	union group (handle)

int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	intersection group (handle)

**int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)**

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	newgroup	difference group (handle)

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
                        MPI_Group *newgroup)
```

```
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

The set-like operations are defined as follows:

union All elements of the first group (**group1**), followed by all elements of second group (**group2**) not in first.

intersect all elements of the first group that are also in the second group, ordered as in first group.

difference all elements of the first group that are not in the second group, ordered as in the first group.

Note that for these operations the order of processes in the output group is determined primarily by order in the first group (if possible) and then, if necessary, by order in the second group. Neither union nor intersection are commutative, but both are associative.

The new group can be empty, that is, equal to **MPI_GROUP_EMPTY**.

MPI_GROUP_INCL(group, n, ranks, newgroup)

IN	group	group (handle)
IN	n	number of elements in array ranks (and size of newgroup) (integer)
IN	ranks	ranks of processes in group to appear in newgroup (array of integers)
OUT	newgroup	new group derived from above, in the order defined by ranks (handle)

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
```

```
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

The function **MPI_GROUP_INCL** creates a group **newgroup** that consists of the **n** processes in **group** with ranks **rank[0]**, ..., **rank[n-1]**; the process with rank **i** in **newgroup** is the process with rank **ranks[i]** in **group**. Each of the **n** elements of **ranks** must be a valid rank in **group** and all elements must be distinct, or else the program is erroneous. If **n = 0**, then **newgroup** is **MPI_GROUP_EMPTY**. This function can, for instance, be used to reorder the elements of a group. See also **MPI_GROUP_COMPARE**.

1 MPI_GROUP_EXCL(group, n, ranks, newgroup)

2	IN	group	group (handle)
3	IN	n	number of elements in array ranks (integer)
4	IN	ranks	array of integer ranks in group not to appear in newgroup
5			group
6			
7	OUT	newgroup	new group derived from above, preserving the order defined by group (handle)
8			

10 int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

11 MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)

12 INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

14 The function MPI_GROUP_EXCL creates a group of processes **newgroup** that is obtained by deleting from **group** those processes with ranks **ranks[0] ... ranks[n-1]**. The ordering of processes in **newgroup** is identical to the ordering in **group**. Each of the **n** elements of **ranks** must be a valid rank in **group** and all elements must be distinct; otherwise, the program is erroneous. If **n** = 0, then **newgroup** is identical to **group**.

21 MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)

22	IN	group	group (handle)
23	IN	n	number of triplets in array ranges (integer)
24	IN	ranges	an array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be included in newgroup
25			
26			
27			
28	OUT	newgroup	new group derived from above, in the order defined by ranges (handle)
29			

31 int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
32 MPI_Group *newgroup)

34 MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)

35 INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

36 If **ranges** consist of the triplets

37 $(first_1, last_1, stride_1), \dots, (first_n, last_n, stride_n)$

39 then **newgroup** consists of the sequence of processes in **group** with ranks

41 $first_1, first_1 + stride_1, \dots, first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, \dots$

42 $first_n, first_n + stride_n, \dots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$

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

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

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

IN	group	group (handle)
IN	n	number of elements in array ranks (integer)
IN	ranges	a one-dimensional array of integer triplets of the form (first rank, last rank, stride), indicating the ranks in group of processes to be excluded from the output group <code>newgroup</code> .
OUT	newgroup	new group derived from above, preserving the order in <code>group</code> (handle)

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
                        MPI_Group *newgroup)
```

```
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 `ranges` replaced by the triplet `(i,i,1)` in the argument `ranges`.

Advice to users. The range operations do not explicitly enumerate ranks, and therefore are more scalable if implemented efficiently. Hence, we recommend MPI programmers to use them whenever 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.*)

5.3.3 Group Destructors

`MPI_GROUP_FREE(group)`

INOUT	group	group (handle)
-------	--------------	----------------

```
int MPI_Group_free(MPI_Group *group)
```

```

1 MPI_GROUP_FREE(GROUP, IERROR)
2     INTEGER GROUP, IERROR

```

This operation marks a group object for deallocation. The handle **group** is set to `MPI_GROUP_NULL` by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to `MPI_COMM_CREATE` and `MPI_COMM_DUP`, and decremented for each call to `MPI_GROUP_FREE` or `MPI_COMM_FREE`; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

5.4 Communicator Management

This section describes the manipulation of communicators in **MPI**. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (*End of advice to implementors.*)

5.4.1 Communicator Accessors

The following are all local operations.

```

28 MPI_COMM_SIZE(comm, size)

```

IN	comm	communicator (handle)
OUT	size	number of processes in the group of comm (integer)

```

33 int MPI_Comm_size(MPI_Comm comm, int *size)

```

```

35 MPI_COMM_SIZE(COMM, SIZE, IERROR)
36     INTEGER COMM, SIZE, IERROR

```

Rationale. This function is equivalent to accessing the communicator's group with `MPI_COMM_GROUP` (see above), computing the size using `MPI_GROUP_SIZE`, and then freeing the temporary group via `MPI_GROUP_FREE`. However, this function is so commonly used, that this shortcut was introduced. (*End of rationale.*)

Advice to users. This function indicates the number of processes involved in a communicator. For `MPI_COMM_WORLD`, it indicates the total number of processes available (for this version of **MPI**, there is no standard way to change the number of processes once initialization has taken place).

This call is often used with the next call to determine the amount of concurrency available for a specific library or program. The following call, `MPI_COMM_RANK`

indicates the rank of the process that calls it in the range from 0...size-1, where size is the return value of MPI_COMM_SIZE. (*End of advice to users.*)

MPI_COMM_RANK(comm, rank)

IN	comm	communicator (handle)
OUT	rank	rank of the calling process in group of comm (integer)

int MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI_COMM_RANK(COMM, RANK, IERROR)

INTEGER COMM, RANK, IERROR

Rationale. This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK, and then freeing the temporary group via MPI_GROUP_FREE. However, this function is so commonly used, that this shortcut was introduced. (*End of rationale.*)

Advice to users. This function gives the rank of the process in the particular communicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.

Many programs will be written with the master-slave model, where one process (such as the rank-zero process) will play a supervisory role, and the other processes will serve as compute nodes. In this framework, the two preceding calls are useful for determining the roles of the various processes of a communicator. (*End of advice to users.*)

MPI_COMM_COMPARE(comm1, comm2, result)

IN	comm1	first communicator (handle)
IN	comm2	second communicator (handle)
OUT	result	result (integer)

int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)

INTEGER COMM1, COMM2, RESULT, IERROR

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

5.4.2 Communicator Constructors

The following are collective functions that are invoked by all processes in the group associated with comm.

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.*)

```
MPI_COMM_DUP(comm, newcomm)
```

```
IN      comm      communicator (handle)
```

```
OUT     newcomm    copy of comm (handle)
```

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

```
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
```

```
INTEGER COMM, NEWCOMM, IERROR
```

MPI_COMM_DUP Duplicates the existing communicator `comm` 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 communicator. Returns in `newcomm` a new communicator with the same group, any copied cached information, but a new context (see section 5.7.1).

Advice to users. This operation is used to provide a parallel library call with a duplicate communication space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see chapter 6). This call is valid even if there are pending point-to-point communications involving the communicator `comm`. A typical call might involve a **MPI_COMM_DUP** at the beginning of the parallel call, and an **MPI_COMM_FREE** of that duplicated communicator at the end of the call. Other models of communicator management are also possible.

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

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

```
MPI_COMM_CREATE(comm, group, newcomm)
```

```
IN      comm      communicator (handle)
```

```
IN      group      Group, which is a subset of the group of comm (handle)
```

```
OUT     newcomm    new communicator (handle)
```

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

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

INTEGER COMM, GROUP, NEWCOMM, IERROR

This function creates a new communicator **newcomm** with communication group defined by **group** and a new context. No cached information propagates from **comm** to **newcomm**. The function returns **MPI_COMM_NULL** to processes that are not in **group**. The call is erroneous if not all **group** arguments have the same value, or if **group** is not a subset of the group associated with **comm**. Note that the call is to be executed by all processes in **comm**, even if they do not belong to the new group. This call applies only to intra-communicators.

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 sometimes to 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) further to subdivide a computation into parallel sub-computations. A more general service is provided by **MPI_COMM_SPLIT**, below. (*End of advice to users.*)

Advice to implementors. Since all processes calling **MPI_COMM_DUP** or **MPI_COMM_CREATE** provide the same **group** argument, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

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

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assignment (integer)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

```

1 MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
2     INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

```

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

A call to **MPI_COMM_CREATE(comm, group, newcomm)** is equivalent to a call to **MPI_COMM_SPLIT(comm, color, key, newcomm)**, where all members of **group** provide **color = 0** and **key = rank** in **group**, and all processes that are not members of **group** provide **color = MPI_UNDEFINED**. The function **MPI_COMM_SPLIT** allows more general partitioning of a group into one or more subgroups with optional reordering. This call applies only intra-communicators.

The value of **color** must be nonnegative.

Advice to users. This is an extremely powerful mechanism for dividing a single communicating group of processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra.

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. (In general, they will have different ranks.)

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

Rationale. **color** is restricted to be nonnegative, so as not to conflict with the value assigned to **MPI_UNDEFINED**. (*End of rationale.*)

5.4.3 Communicator Destructors

```

43 MPI_COMM_FREE(comm)

```

```

45     INOUT    comm                communicator to be destroyed (handle)

```

```

47 int MPI_Comm_free(MPI_Comm *comm)

```

```
MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR
```

This collective operation marks the communication object for deallocation. The handle is set to `MPI_COMM_NULL`. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see section 5.7) are called in arbitrary order.

Advice to implementors. A reference-count mechanism may be used: the reference count is incremented by each call to `MPI_COMM_DUP`, and decremented by each call to `MPI_COMM_FREE`. The object is ultimately deallocated when the count reaches zero.

Though collective, it is anticipated that this operation will normally be implemented to be local, though the debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

5.5 Motivating Examples

5.5.1 Current Practice #1

Example #1a:

```
main(int argc, char **argv)
{
    int me, size;
    ...
    MPI_Init ( &argc, &argv );
    MPI_Comm_rank (MPI_COMM_WORLD, &me);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    (void)printf ("Process %d size %d\n", me, size);
    ...
    MPI_Finalize();
}
```

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

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

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

    MPI_Comm_rank(MPI_COMM_WORLD, &me);    /* local */
}
```

```

1      MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
2
3      if((me % 2) == 0)
4      {
5          /* send unless highest-numbered process */
6          if((me + 1) < size)
7              MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
8      }
9      else
10         MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD);
11
12         ...
13         MPI_Finalize();
14     }

```

Example #1b schematically illustrates message exchanges between “even” and “odd” processes in the “all” communicator.

5.5.2 Current Practice #2

```

20     main(int argc, char **argv)
21     {
22         int me, count;
23         void *data;
24         ...
25
26         MPI_Init(&argc, &argv);
27         MPI_Comm_rank(MPI_COMM_WORLD, &me);
28
29         if(me == 0)
30         {
31             /* get input, create buffer ‘data’ */
32             ...
33         }
34
35         MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
36
37         ...
38
39         MPI_Finalize();
40     }

```

This example illustrates the use of a collective communication.

5.5.3 (Approximate) Current Practice #3

```

46     main(int argc, char **argv)
47     {
48         int me, count, count2;

```

```

void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
MPI_Group MPI_GROUP_WORLD, grprem;
MPI_Comm commslave;
static int ranks[] = {0};
...
MPI_Init(&argc, &argv);
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */

MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem); /* local */
MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);

if(me != 0)
{
    /* compute on slave */
    ...
    MPI_Reduce(send_buf, recv_buff, count, MPI_INT, MPI_SUM, 1, commslave);
    ...
}
/* zero falls through immediately to this reduce, others do later... */
MPI_Reduce(send_buf2, recv_buff2, count2,
           MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Comm_free(&commslave);
MPI_Group_free(&MPI_GROUP_WORLD);
MPI_Group_free(&grprem);
MPI_Finalize();
}

```

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

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

5.5.4 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
#define SOME_COUNT 50

main(int argc, char **argv)
{

```

```

1      int me;
2      MPI_Request request[2];
3      MPI_Status status[2];
4      MPI_Group MPI_GROUP_WORLD, subgroup;
5      int ranks[] = {2, 4, 6, 8};
6      MPI_Comm the_comm;
7      ...
8      MPI_Init(&argc, &argv);
9      MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
10
11     MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
12     MPI_Group_rank(subgroup, &me); /* local */
13
14     MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
15
16     if(me != MPI_UNDEFINED)
17     {
18         MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
19                 the_comm, request);
20         MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
21                 the_comm, request+1);
22     }
23
24     for(i = 0; i < SOME_COUNT, i++)
25         MPI_Reduce(..., the_comm);
26     MPI_Waitall(2, request, status);
27
28     MPI_Comm_free(&the_comm);
29     MPI_Group_free(&MPI_GROUP_WORLD);
30     MPI_Group_free(&subgroup);
31     MPI_Finalize();
32 }

```

5.5.5 Library Example #1

The main program:

```

37     main(int argc, char **argv)
38     {
39         int done = 0;
40         user_lib_t *libh_a, *libh_b;
41         void *dataset1, *dataset2;
42         ...
43         MPI_Init(&argc, &argv);
44         ...
45         init_user_lib(MPI_COMM_WORLD, &libh_a);
46         init_user_lib(MPI_COMM_WORLD, &libh_b);
47         ...
48         user_start_op(libh_a, dataset1);

```

```

user_start_op(libh_b, dataset2);
...
while(!done)
{
    /* work */
    ...
    MPI_Reduce(..., MPI_COMM_WORLD);
    ...
    /* see if done */
    ...
}
user_end_op(libh_a);
user_end_op(libh_b);

uninit_user_lib(libh_a);
uninit_user_lib(libh_b);
MPI_Finalize();
}

```

The user library initialization code:

```

void init_user_lib(MPI_Comm comm, user_lib_t **handle)
{
    user_lib_t *save;

    user_lib_initsave(&save); /* local */
    MPI_Comm_dup(comm, &(save -> comm));

    /* other inits */
    ...

    *handle = save;
}

```

User start-up code:

```

void user_start_op(user_lib_t *handle, void *data)
{
    MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
    MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
}

```

User communication clean-up code:

```

void user_end_op(user_lib_t *handle)
{
    MPI_Status *status;
    MPI_Wait(handle -> isend_handle, status);
    MPI_Wait(handle -> irecv_handle, status);
}

```


User object clean-up code:

```

void uninit_user_lib(user_lib_t *handle)
{
    MPI_Comm_free(&(handle -> comm));
    free(handle);
}

```

5.5.6 Library Example #2

The main program:

```

main(int argc, char **argv)
{
    int ma, mb;
    MPI_Group MPI_GROUP_WORLD, group_a, group_b;
    MPI_Comm comm_a, comm_b;

    static int list_a[] = {0, 1};
    #if defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
    static int list_b[] = {0, 2, 3};
    #else /* EXAMPLE_2A */
    static int list_b[] = {0, 2};
    #endif

    int size_list_a = sizeof(list_a)/sizeof(int);
    int size_list_b = sizeof(list_b)/sizeof(int);

    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

    MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
    MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);

    MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
    MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);

    if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
    if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_b, &mb);

    if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);

    if(comm_b != MPI_COMM_NULL)
    {
        lib_call(comm_b);
        lib_call(comm_b);
    }
}

```

```

    }
    if(comm_a != MPI_COMM_NULL)
        MPI_Comm_free(&comm_a);
    if(comm_b != MPI_COMM_NULL)
        MPI_Comm_free(&comm_b);
    MPI_Group_free(&group_a);
    MPI_Group_free(&group_b);
    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Finalize();
}

```

The library:

```

void lib_call(MPI_Comm comm)
{
    int me, done = 0;
    MPI_Comm_rank(comm, &me);
    if(me == 0)
        while(!done)
        {
            MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm);
            ...
        }
    else
    {
        /* work */
        MPI_Send(..., 0, ARBITRARY_TAG, comm);
        ....
    }
}
#ifdef EXAMPLE_2C
    /* include (resp, exclude) for safety (resp, no safety): */
    MPI_Barrier(comm);
#endif
}

```

The above example is really three examples, depending on whether or not one includes rank 3 in `list_b`, and whether or not a synchronize is included in `lib_call`. This example illustrates that, despite contexts, subsequent calls to `lib_call` with the same context need not be safe from one another (colloquially, “back-masking”). Safety is realized if the `MPI_Barrier` is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back masking.

Algorithms like “reduce” and “allreduce” have strong enough source selectivity properties so that they are inherently okay (no backmasking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [28]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity — deleting either feature removes the guarantee that backmasking cannot be required.

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

All of the foregoing is a supposition of “collective calls” implemented with point-to-point operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also section 5.8.

5.6 Inter-Communication

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All point-to-point communication described thus far has involved communication between processes that are members of the same group. This type of communication is called “intra-communication” and the communicator used is called an “intra-communicator,” as we have noted earlier in the chapter.

In modular and multi-disciplinary applications, different process groups execute distinct modules and processes within different modules communicate with one another in a pipeline or a more general module graph. In these applications, the most natural way for a process to specify a target process is by the rank of the target process within the target group. In applications that contain internal user-level servers, each server may be a process group that provides services to one or more clients, and each client may be a process group that uses the services of one or more servers. It is again most natural to specify the target process by rank within the target group in these applications. This type of communication is called “inter-communication” and the communicator used is called an “inter-communicator,” as introduced earlier.

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the “local group,” that is, the sender in a send and the receiver in a receive. The group containing the target process is called the “remote group,” that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (**communicator, rank**) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking and require that the local and remote groups be disjoint in order to avoid deadlock.

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point communication is the same for both inter- and 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.

- An inter-communicator cannot be used for collective communication.
- 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_COMM_CREATE`).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

group
send_context
receive_context
source

For inter-communicators, **group** describes the remote group, and **source** is the rank of the process in the local group. For intra-communicators, **group** is the communicator group (remote=local), **source** is the rank of the process in this group, and **send_context** and **receive_context** are identical. A group is represented by a rank-to-absolute-address translation table.

The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process **P** in group \mathcal{P} , which has an inter-communicator $C_{\mathcal{P}}$, and a process **Q** in group \mathcal{Q} , which has an inter-communicator $C_{\mathcal{Q}}$. Then

- $C_{\mathcal{P}}.\text{group}$ describes the group \mathcal{Q} and $C_{\mathcal{Q}}.\text{group}$ describes the group \mathcal{P} .
- $C_{\mathcal{P}}.\text{send_context} = C_{\mathcal{Q}}.\text{receive_context}$ and the context is unique in \mathcal{Q} ;
 $C_{\mathcal{P}}.\text{receive_context} = C_{\mathcal{Q}}.\text{send_context}$ and this context is unique in \mathcal{P} .
- $C_{\mathcal{P}}.\text{source}$ is rank of **P** in \mathcal{P} and $C_{\mathcal{Q}}.\text{source}$ is rank of **Q** in \mathcal{Q} .

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.

The same algorithm is appropriate for intra-communicators as well.

In order to support inter-communicator accessors and constructors, it is necessary to supplement this model with additional structures, that store information about the local communication group, and additional safe contexts. (*End of advice to implementors.*)

5.6.1 Inter-communicator Accessors

MPI_COMM_TEST_INTER(comm, flag)

IN **comm** communicator (handle)
OUT **flag** (logical)

int MPI_Comm_test_inter(MPI_Comm comm, int *flag)

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)

INTEGER COMM, IERROR

LOGICAL FLAG

This local routine allows the calling process to determine if a communicator is an inter-communicator 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_* Function Behavior (in Inter-Communication Mode)	
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

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** and **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 inter-communicator:

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**
 (integer)

int MPI_Comm_remote_size(MPI_Comm comm, int *size)

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)

```

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

```

MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR

```

Rationale. Symmetric access to both the local and remote groups of an inter-communicator is important, so this function, as well as `MPI_COMM_REMOTE_SIZE` have been provided. (*End of rationale.*)

5.6.2 Inter-communicator Operations

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 merging 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 (the two leaders could be the same process). Furthermore, members of each group know the rank of their leader.

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

In standard MPI implementations (with static process allocation at initialization), the `MPI_COMM_WORLD` communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. In dynamic MPI implementations, where, for example, a process may spawn new child processes during an MPI execution, the parent process may be the “bridge” between the old communication universe and the new communication world that includes the parent and its children.

The application topology functions described in chapter 6 do not apply to inter-communicators. Users that require this capability should utilize `MPI_INTERCOMM_MERGE` to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topology-oriented intra-communicator. Alternatively, it may be reasonable to devise one’s own application topology mechanisms for this case, without loss of generality.

```

1  MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag,
2  newintercomm)
3      IN      local_comm      local intra-communicator (handle)
4      IN      local_leader    rank of local group leader in local_comm (integer)
5      IN      peer_comm       "peer" communicator; significant only at the local_leader
6                               (handle)
7
8      IN      remote_leader    rank of remote group leader in peer_comm; significant
9                               only at the local_leader (integer)
10
11     IN      tag              "safe" tag (integer)
12
13     OUT     newintercomm      new inter-communicator (handle)

```

```

14  int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
15                          MPI_Comm peer_comm, int remote_leader, int tag,
16                          MPI_Comm *newintercomm)
17
18  MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
19                      NEWINTERCOMM, IERROR)
20  INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
21  NEWINTERCOMM, IERROR

```

This call creates an inter-communicator. It is collective over the union of the local and remote groups. Processes should provide identical `local_comm` and `local_leader` arguments within each group. Wildcards are not permitted for `remote_leader`, `local_leader`, and `tag`.

This call uses point-to-point communication with communicator `peer_comm`, and with tag `tag` between the leaders. Thus, care must be taken that there be no pending communication on `peer_comm` that could interfere with this communication.

Advice to users. We recommend using a dedicated peer communicator, such as a duplicate of `MPI_COMM_WORLD`, to avoid trouble with peer communicators. (*End of advice to users.*)

```

34  MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)
35
36  IN      intercomm      Inter-Communicator (handle)
37  IN      high           (logical)
38
39  OUT     newintracomm    new intra-communicator (handle)

```

```

40
41  int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
42                          MPI_Comm *newintracomm)
43
44  MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
45  INTEGER INTERCOMM, INTRACOMM, IERROR
46  LOGICAL HIGH

```

This function creates an intra-communicator from the union of the two groups that are associated with `intercomm`. All processes should provide the same `high` value within each

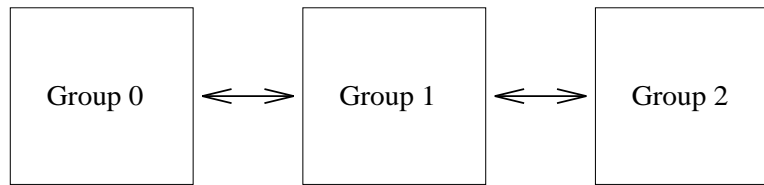


Figure 5.1: Three-group pipeline.

of the two groups. If processes in one group provided the value `high = false` and processes in the other group provided the value `high = true` then the union orders the “low” group before the “high” group. If all processes provided the same `high` argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

Advice to implementors. The implementation of `MPI_INTERCOMM_MERGE`, `MPI_COMM_FREE` and `MPI_COMM_DUP` are similar to the implementation of `MPI_INTERCOMM_CREATE`, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)

5.6.3 Inter-Communication Examples

Example 1: Three-Group “Pipeline”

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1 inter-communicator.

```

main(int argc, char **argv)
{
    MPI_Comm    myComm;          /* intra-communicator of local sub-group */
    MPI_Comm    myFirstComm;     /* inter-communicator */
    MPI_Comm    mySecondComm;   /* second inter-communicator (group 1 only) */
    int membershipKey;
    int rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* User code must generate membershipKey in the range [0, 1, 2] */
    membershipKey = rank % 3;

    /* Build intra-communicator for local sub-group */
    MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

    /* Build inter-communicators. Tags are hard-coded. */
    if (membershipKey == 0)
    {
        /* Group 0 communicates with group 1. */
        MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,

```

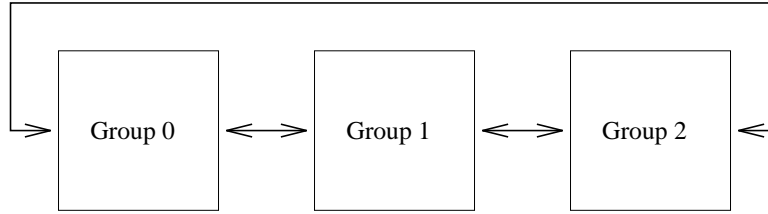



Figure 5.2: Three-group ring.

```

1
2
3
4
5
6
7
8
9
10         1, &myFirstComm);
11     }
12     else if (membershipKey == 1)
13     {
14         /* Group 1 communicates with groups 0 and 2. */
15         MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
16                               1, &myFirstComm);
17         MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
18                               12, &mySecondComm);
19     }
20     else if (membershipKey == 2)
21     {
22         /* Group 2 communicates with group 1. */
23         MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
24                               12, &myFirstComm);
25     }
26
27     /* Do work ... */
28
29     switch(membershipKey) /* free communicators appropriately */
30     {
31     case 1:
32         MPI_Comm_free(&mySecondComm);
33     case 0:
34     case 2:
35         MPI_Comm_free(&myFirstComm);
36         break;
37     }
38
39     MPI_Finalize();
40 }

```

Example 2: Three-Group “Ring”

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

```

44     main(int argc, char **argv)
45     {
46         MPI_Comm    myComm;      /* intra-communicator of local sub-group */
47         MPI_Comm    myFirstComm; /* inter-communicators */
48

```

```

MPI_Comm  mySecondComm;
MPI_Status status;
int membershipKey;
int rank;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...

/* User code must generate membershipKey in the range [0, 1, 2] */
membershipKey = rank % 3;

/* Build intra-communicator for local sub-group */
MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

/* Build inter-communicators. Tags are hard-coded. */
if (membershipKey == 0)
{
    /* Group 0 communicates with groups 1 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          2, &mySecondComm);
}
else if (membershipKey == 1)
{
    /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm);
}
else if (membershipKey == 2)
{
    /* Group 2 communicates with groups 0 and 1. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          2, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          12, &mySecondComm);
}

/* Do some work ... */

/* Then free communicators before terminating... */
MPI_Comm_free(&myFirstComm);
MPI_Comm_free(&mySecondComm);
MPI_Comm_free(&myComm);
MPI_Finalize();
}

```

Example 3: Building Name Service for Intercommunication

The following procedures exemplify the process by which a user could create name service for building intercommunicators via a rendezvous involving a server communicator, and a tag name selected by both groups.

After all MPI processes execute `MPI_INIT`, every process calls the example function, `Init_server()`, defined below. Then, if the `new_world` returned is `NULL`, the process getting `NULL` is required to implement a server function, in a reactive loop, `Do_server()`. Everyone else just does their prescribed computation, using `new_world` as the new effective “global” communicator. One designated process calls `Undo_Server()` to get rid of the server when it is not needed any longer.

Features of this approach include:

- Support for multiple name servers
- Ability to scope the name servers to specific processes
- Ability to make such servers come and go as desired.

```
#define INIT_SERVER_TAG_1 666
#define UNDO_SERVER_TAG_1 777

static int server_key_val;

/* for attribute management for server_comm, copy callback: */
void handle_copy_fn(MPI_Comm *oldcomm, int *keyval, void *extra_state,
void *attribute_val_in, void **attribute_val_out, int *flag)
{
    /* copy the handle */
    *attribute_val_out = attribute_val_in;
    *flag = 1; /* indicate that copy to happen */
}

int Init_server(peer_comm, rank_of_server, server_comm, new_world)
MPI_Comm peer_comm;
int rank_of_server;
MPI_Comm *server_comm;
MPI_Comm *new_world; /* new effective world, sans server */
{
    MPI_Comm temp_comm, lone_comm;
    MPI_Group peer_group, temp_group;
    int rank_in_peer_comm, size, color, key = 0;
    int peer_leader, peer_leader_rank_in_temp_comm;

    MPI_Comm_rank(peer_comm, &rank_in_peer_comm);
    MPI_Comm_size(peer_comm, &size);

    if ((size < 2) || (0 > rank_of_server) || (rank_of_server >= size))
        return (MPI_ERR_OTHER);
```

```

1
2  /* create two communicators, by splitting peer_comm
3     into the server process, and everyone else */
4
5  peer_leader = (rank_of_server + 1) % size;  /* arbitrary choice */
6
7  if ((color = (rank_in_peer_comm == rank_of_server)))
8  {
9      MPI_Comm_split(peer_comm, color, key, &lone_comm);
10
11      MPI_Intercomm_create(lone_comm, 0, peer_comm, peer_leader,
12                          INIT_SERVER_TAG_1, server_comm);
13
14      MPI_Comm_free(&lone_comm);
15      *new_world = MPI_COMM_NULL;
16  }
17  else
18  {
19      MPI_Comm_Split(peer_comm, color, key, &temp_comm);
20
21      MPI_Comm_group(peer_comm, &peer_group);
22      MPI_Comm_group(temp_comm, &temp_group);
23      MPI_Group_translate_ranks(peer_group, 1, &peer_leader,
24                              temp_group, &peer_leader_rank_in_temp_comm);
25
26      MPI_Intercomm_create(temp_comm, peer_leader_rank_in_temp_comm,
27                          peer_comm, rank_of_server,
28                          INIT_SERVER_TAG_1, server_comm);
29
30      /* attach new_world communication attribute to server_comm: */
31
32      /* CRITICAL SECTION FOR MULTITHREADING */
33      if(server_keyval == MPI_KEYVAL_INVALID)
34      {
35          /* acquire the process-local name for the server keyval */
36          MPI_keyval_create(handle_copy_fn, NULL,
37                          &server_keyval, NULL);
38      }
39
40      *new_world = temp_comm;
41
42      /* Cache handle of intra-communicator on inter-communicator: */
43      MPI_Attr_put(server_comm, server_keyval, (void *)(*new_world));
44  }
45
46  return (MPI_SUCCESS);
47
48  }

```

The actual server process would commit to running the following code:

```

1      The actual server process would commit to running the following code:
2
3      int Do_server(server_comm)
4      MPI_Comm server_comm;
5      {
6          void init_queue();
7          int en_queue(), de_queue(); /* keep triplets of integers
8                                     for later matching (fns not shown) */
9
10         MPI_Comm comm;
11         MPI_Status status;
12         int client_tag, client_source;
13         int client_rank_in_new_world, pairs_rank_in_new_world;
14         int buffer[10], count = 1;
15
16         void *queue;
17         init_queue(&queue);
18
19         for (;;)
20         {
21             MPI_Recv(buffer, count, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
22                     server_comm, &status); /* accept from any client */
23
24             /* determine client: */
25             client_tag = status.MPI_TAG;
26             client_source = status.MPI_SOURCE;
27             client_rank_in_new_world = buffer[0];
28
29             if (client_tag == UNDO_SERVER_TAG_1)          /* client that
30                                                         terminates server */
31             {
32                 while (de_queue(queue, MPI_ANY_TAG, &pairs_rank_in_new_world,
33                             &pairs_rank_in_server))
34                     ;
35
36                 MPI_Intercomm_free(&server_comm);
37                 break;
38             }
39
40             if (de_queue(queue, client_tag, &pairs_rank_in_new_world,
41                             &pairs_rank_in_server))
42             {
43                 /* matched pair with same tag, tell them
44                  about each other! */
45                 buffer[0] = pairs_rank_in_new_world;
46                 MPI_Send(buffer, 1, MPI_INT, client_src, client_tag,
47                         server_comm);
48

```

```

        buffer[0] = client_rank_in_new_world;
        MPI_Send(buffer, 1, MPI_INT, pairs_rank_in_server, client_tag,
                 server_comm);
    }
    else
        en_queue(queue, client_tag, client_source,
                 client_rank_in_new_world);
}
}

```

A particular process would be responsible for ending the server when it is no longer needed. Its call to `Undo_server` would terminate server function.

```

int Undo_server(server_comm)    /* example client that ends server */
MPI_Comm *server_comm;
{
    int buffer = 0;
    MPI_Send(&buffer, 1, MPI_INT, 0, UNDO_SERVER_TAG_1, *server_comm);
    MPI_Intercomm_free(server_comm);
}

```

The following is a blocking name-service for inter-communication, with same semantic restrictions as `MPI_Intercomm_create`, but simplified syntax. It uses the functionality just defined to create the name service.

```

int Intercomm_name_create(local_comm, server_comm, tag, comm)
MPI_Comm local_comm, server_comm;
int tag;
MPI_Comm *comm;
{
    int error;
    int found;    /* attribute acquisition mgmt for new_world */
                 /* comm in server_comm */

    void *val;

    MPI_Comm new_world;

    int buffer[10], rank;
    int local_leader = 0;

    MPI_Attr_get(server_comm, server_keyval, &val, &found);
    new_world = (MPI_Comm)val; /* retrieve cached handle */

    MPI_Comm_rank(server_comm, &rank); /* rank in local group */

    if (rank == local_leader)
    {

```

```

1      buffer[0] = rank;
2      MPI_Send(&buffer, 1, MPI_INT, 0, tag, server_comm);
3      MPI_Recv(&buffer, 1, MPI_INT, 0, tag, server_comm);
4  }
5
6      error = MPI_Intercomm_create(local_comm, local_leader, new_world,
7                                  buffer[0], tag, comm);
8
9      return(error);
10 }

```

5.7 Caching

MPI provides a “caching” facility that allows an application to attach arbitrary pieces of information, called **attributes**, to communicators. 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,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the communicator is freed and its handle subsequently reused by MPI.

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

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

5.7.1 Functionality

Attributes are attached to communicators. 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` (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 represents that attributes be stored by MPI opaquely within a communicator. 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 avoid problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)

The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators.

A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid access to attributes through the use of pointers in communicators (to find the attribute table) and cleverly chosen key values (to retrieve individual attributes). In light of the efficiency “hit” inherent in the minimal interface, the more complete interface defined here is seen to be superior. (*End of advice to implementors.*)

MPI provides the following services related to caching. They are all process local.

MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)

IN	copy_fn	Copy callback function for keyval
IN	delete_fn	Delete callback function for keyval
OUT	keyval	key value for future access (integer)
IN	extra_state	Extra state for callback functions

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
                     *delete_fn, int *keyval, void* extra_state)
```

```
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
EXTERNAL COPY_FN, DELETE_FN
INTEGER KEYVAL, EXTRA_STATE, IERROR
```

Generates a new attribute key. Keys are locally unique in a process, and opaque to user, though they are explicitly stored in integers. Once allocated, the key value can be used to associate attributes and access them on any locally defined communicator.

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:


```

1  typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
2                                void *extra_state, void *attribute_val_in,
3                                void *attribute_val_out, int *flag)

```

A Fortran declaration for such a function is as follows:

```

5  SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
6                                ATTRIBUTE_VAL_OUT, FLAG, IERR)
7                                INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
8                                ATTRIBUTE_VAL_OUT, IERR
9                                LOGICAL FLAG
10

```

The copy callback function is invoked for each key value in `oldcomm` in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns `flag = 0`, then the attribute is deleted in the duplicated communicator. Otherwise (`flag = 1`), the new attribute value is set to the value returned in `attribute_val_out`. The function returns `MPI_SUCCESS` on success and an error code on failure (in which case `MPI_COMM_DUP` will fail).

`copy_fn` may be specified as `MPI_NULL_COPY_FN` or `MPI_DUP_FN` from either C or FORTRAN; `MPI_NULL_COPY_FN` is a function that does nothing other than returning `flag = 0` and `MPI_SUCCESS`. `MPI_DUP_FN` is a simple-minded copy function that sets `flag = 1`, returns the value of `attribute_val_in` in `attribute_val_out`, and returns `MPI_SUCCESS`.

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 `copy_fn` is a callback deletion function, defined as follows. The `delete_fn` function is invoked when a communicator is deleted by `MPI_COMM_FREE` or when a call is made explicitly to `MPI_ATTR_DELETE`. `delete_fn` should be of type `MPI_Delete_function`, which is defined as follows:

```

42  typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
43                                void *attribute_val, void *extra_state);

```

A Fortran declaration for such a function is as follows:

```

45  SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
46                                INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
47

```

This function is called by `MPI_COMM_FREE`, `MPI_ATTR_DELETE`, and `MPI_ATTR_PUT` 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).

`delete_fn` may be specified as `MPI_NULL_DELETE_FN` from either C or FORTRAN; `MPI_NULL_DELETE_FN` is a function that does nothing, other than returning `MPI_SUCCESS`.

The special key value `MPI_KEYVAL_INVALID` is never returned by `MPI_KEYVAL_CREATE`. Therefore, it can be used for static initialization of key values.

`MPI_KEYVAL_FREE(keyval)`

INOUT	<code>keyval</code>	Frees the integer key value (integer)
-------	---------------------	---------------------------------------

`int MPI_Keyval_free(int *keyval)`

`MPI_KEYVAL_FREE(KEYVAL, IERROR)`

INTEGER KEYVAL, IERROR

Frees an extant attribute key. This function sets the value of `keyval` to `MPI_KEYVAL_INVALID`. Note that it is not erroneous to free an attribute key that is in use, because the actual free does not transpire until after all references (in other communicators on the process) to the key have been freed. These references need to be explicitly freed by the program, either via calls to `MPI_ATTR_DELETE` that free one attribute instance, or by calls to `MPI_COMM_FREE` that free all attribute instances associated with the freed communicator.

`MPI_ATTR_PUT(comm, keyval, attribute_val)`

IN	<code>comm</code>	communicator to which attribute will be attached (handle)
IN	<code>keyval</code>	key value, as returned by <code>MPI_KEYVAL_CREATE</code> (integer)
IN	<code>attribute_val</code>	attribute value

`int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)`

`MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)`

INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

This function stores the stipulated attribute value `attribute_val` for subsequent retrieval by `MPI_ATTR_GET`. If the value is already present, then the outcome is as if `MPI_ATTR_DELETE` was first called to delete the previous value (and the callback function `delete_fn` was executed), and a new value was next stored. The call is erroneous if there is no key with value `keyval`; in particular `MPI_KEYVAL_INVALID` is an erroneous key value. The call will fail if the `delete_fn` function returned an error code other than `MPI_SUCCESS`.

```
1 MPI_ATTR_GET(comm, keyval, attribute_val, flag)
```

```
2     IN      comm      communicator to which attribute is attached (handle)
3     IN      keyval    key value (integer)
4
5     OUT     attribute_val attribute value, unless flag = false
6     OUT     flag      true if an attribute value was extracted; false if no
7                     attribute is associated with the key
```

```
9 int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
```

```
10 MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
```

```
11     INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
```

```
12     LOGICAL FLAG
```

14 Retrieves attribute value by key. The call is erroneous if there is no key with value
15 **keyval**. On the other hand, the call is correct if the key value exists, but no attribute is
16 attached on **comm** for that key; in such case, the call returns **flag = false**. In particular
17 **MPI_KEYVAL_INVALID** is an erroneous key value.

19 *Advice to users.* The call to **MPI_Attr_put** passes in **attribute_val** the *value* of the
20 attribute; the call to **MPI_Attr_get** passes in **attribute_val** the *address* of the the location
21 where the attribute value is to be returned. Thus, if the attribute value itself is a
22 pointer of type **void***, the the actual **attribute_val** parameter to **MPI_Attr_put** will be
23 of type **void*** and the actual **attribute_val** parameter to **MPI_Attr_put** will be of type
24 **void****. (*End of advice to users.*)

26 *Rationale.* The use of a formal parameter **attribute_val** or type **void*** (rather than
27 **void****) avoids the messy type casting that would be needed if the attribute value is
28 declared with a type other than **void***. (*End of rationale.*)

```
31 MPI_ATTR_DELETE(comm, keyval)
```

```
33     IN      comm      communicator to which attribute is attached (handle)
34     IN      keyval    The key value of the deleted attribute (integer)
```

```
36 int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

```
38 MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
```

```
39     INTEGER COMM, KEYVAL, IERROR
```

41 Delete attribute from cache by key. This function invokes the attribute delete function
42 **delete_fn** specified when the **keyval** was created. The call will fail if the **delete_fn** function
43 returns an error code other than **MPI_SUCCESS**.

44 Whenever a communicator is replicated using the function **MPI_COMM_DUP**, all call-
45 back copy functions for attributes that are currently set are invoked (in arbitrary order).
46 Whenever a communicator is deleted using the function **MPI_COMM_FREE** all callback
47 delete functions for attributes that are currently set are invoked.

5.7.2 Attributes Example

Advice to users. This example shows how to write a collective communication operation that uses caching to be more efficient after the first call. The coding style assumes that MPI function results return only error statuses. (*End of advice to users.*)

```

/* key for this module's stuff: */
static int gop_key = MPI_KEYVAL_INVALID;

typedef struct
{
    int ref_count;          /* reference count */
    /* other stuff, whatever else we want */
} gop_stuff_type;

Efficient_Collective_Op (comm, ...)
MPI_Comm comm;
{
    gop_stuff_type *gop_stuff;
    MPI_Group      group;
    int            foundflag;

    MPI_Comm_group(comm, &group);

    if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
    {
        if ( ! MPI_keyval_create( gop_stuff_copier,
                                gop_stuff_destructor,
                                &gop_key, (void *)0));

        /* get the key while assigning its copy and delete callback
           behavior. */

        MPI_Abort (comm, 99);
    }

    MPI_Attr_get (comm, gop_key, &gop_stuff, &foundflag);
    if (foundflag)
    { /* This module has executed in this group before.
       We will use the cached information */
    }
    else
    { /* This is a group that we have not yet cached anything in.
       We will now do so.
       */

        /* First, allocate storage for the stuff we want,
           and initialize the reference count */

        gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));

```

```

1      if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
2
3      gop_stuff -> ref_count = 1;
4
5      /* Second, fill in *gop_stuff with whatever we want.
6         This part isn't shown here */
7
8      /* Third, store gop_stuff as the attribute value */
9      MPI_Attr_put ( comm, gop_key, gop_stuff);
10     }
11     /* Then, in any case, use contents of *gop_stuff
12        to do the global op ... */
13 }
14
15 /* The following routine is called by MPI when a group is freed */
16
17 gop_stuff_destructor (comm, keyval, gop_stuff, extra)
18 MPI_Comm comm;
19 int keyval;
20 gop_stuff_type *gop_stuff;
21 void *extra;
22 {
23     if (keyval != gop_key) { /* abort -- programming error */ }
24
25     /* The group's being freed removes one reference to gop_stuff */
26     gop_stuff -> ref_count -= 1;
27
28     /* If no references remain, then free the storage */
29     if (gop_stuff -> ref_count == 0) {
30         free((void *)gop_stuff);
31     }
32 }
33
34 /* The following routine is called by MPI when a group is copied */
35 gop_stuff_copier (comm, keyval, extra, gop_stuff_in, gop_stuff_out, flag)
36 MPI_Comm comm;
37 int keyval;
38 gop_stuff_type *gop_stuff_in, *gop_stuff_out;
39 void *extra;
40 {
41     if (keyval != gop_key) { /* abort -- programming error */ }
42
43     /* The new group adds one reference to this gop_stuff */
44     gop_stuff_out -> ref_count += 1;
45     gop_stuff_out = gop_stuff_in;
46 }
47
48

```

5.8 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

5.8.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that communicator must be free of side effects throughout execution of the subprogram: there should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work “safely.” For libraries so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communications will interfere. Since we permit good implementations to create new communicators without synchronization (such as by preallocated contexts on communicators), this does not impose a significant overhead.

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.

5.8.2 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.

1 In such a case, a new communicator needs to be dynamically allocated for each new
2 invocation of a parallel procedure. The allocation is done by the caller. A new communicator
3 can be generated by a call to `MPI_COMM_DUP`, if the callee execution group is identical to
4 the caller execution group, or by a call to `MPI_COMM_SPLIT` if the caller execution group
5 is split into several subgroups executing distinct parallel routines. The new communicator
6 is passed as an argument to the invoked routine.

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

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

- 14 • messages sent before a procedure call (or before a return from the procedure) are also
15 received before the matching call (or return) at the receiving end;
- 16 • messages are always selected by source (no use is made of `MPI_ANY_SOURCE`).

19 The General case

20
21 In the general case, there may be multiple concurrently active invocations of the same
22 parallel procedure within the same group; invocations may not be well-nested. A new
23 communicator needs to be created for each invocation. It is the user's responsibility to make
24 sure that, should two distinct parallel procedures be invoked concurrently on overlapping
25 sets of processes, then communicator creation be properly coordinated.

Chapter 6

Process Topologies

6.1 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 inter-communicators. 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 5, 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 proposed in this chapter deal only with machine-independent mapping.

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 [20]. 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 [10, 9].

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with tremendous benefits for program readability

and notational power in message-passing programming. (*End of rationale.*)

6.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes stand for the 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. Edges in the communication graph are not weighted, so that processes are either simply connected or not connected at all.

Rationale. Experience with similar techniques in PARMACS [5, 8] show that this information is usually sufficient for a good mapping. Additionally, a more precise specification is more difficult for the user to set up, and it would make the interface functions substantially more complicated. (*End of rationale.*)

Specifying the virtual topology in terms of a graph is sufficient for all applications. However, in many applications the graph structure is regular, and the detailed set-up of the graph would be inconvenient for the user and might be less efficient at run time. A large fraction of all parallel applications use process topologies like rings, two- or higher-dimensional grids, or tori. These structures are completely defined by the number of dimensions and the numbers of processes in each coordinate direction. Also, the mapping of grids and tori is generally an easier problem than that of general graphs. Thus, it is desirable to address these cases explicitly.

Process coordinates in a cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a (2×2) grid is as follows.

```
coord (0,0):  rank 0
coord (0,1):  rank 1
coord (1,0):  rank 2
coord (1,1):  rank 3
```

6.3 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 5.

6.4 Overview of the Functions

The functions `MPI_GRAPH_CREATE` and `MPI_CART_CREATE` are used to create general (graph) virtual topologies and cartesian topologies, respectively. These topology creation functions are collective. As with 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. A new communicator `comm_topol` is created that carries the topological structure as cached information (see Chapter 5). In analogy to function `MPI_COMM_CREATE`, no cached information propagates from `comm_old` to `comm_topol`.

`MPI_CART_CREATE` can be used to describe cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an n -dimensional hypercube is an n -dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function `MPI_DIMS_CREATE` can be used to compute a balanced distribution of processes among a given number of dimensions.

Rationale. Similar functions are contained in EXPRESS [22] and PARMACS. (*End of rationale.*)

The function `MPI_TOPO_TEST` can be used to inquire about the topology associated with a communicator. The topological information can be extracted from the communicator using the functions `MPI_GRAPHDIMS_GET` and `MPI_GRAPH_GET`, for general graphs, and `MPI_CARTDIM_GET` and `MPI_CART_GET`, for cartesian topologies. Several additional functions are provided to manipulate cartesian topologies: the functions `MPI_CART_RANK` and `MPI_CART_COORDS` translate cartesian coordinates into a group rank, and vice-versa; the function `MPI_CART_SUB` can be used to extract a cartesian subspace (analogous to `MPI_COMM_SPLIT`). The function `MPI_CART_SHIFT` provides the information needed to communicate with neighbors in a cartesian dimension. The two functions `MPI_GRAPH_NEIGHBORS_COUNT` and `MPI_GRAPH_NEIGHBORS` can be used to extract the neighbors of a node in a graph. The function `MPI_CART_SUB` is collective over the input communicator's group; all other functions are local.

Two additional functions, `MPI_GRAPH_MAP` and `MPI_CART_MAP` are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 5, they are sufficient to implement all other topology functions. Section 6.5.7 outlines such an implementation.

6.5 Topology Constructors

6.5.1 Cartesian Constructor

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

IN	comm_old	input communicator (handle)
IN	ndims	number of dimensions of cartesian grid (integer)
IN	dims	integer array of size ndims specifying the number of processes in each dimension
IN	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
IN	reorder	ranking may be reordered (true) or not (false) (logical)
OUT	comm_cart	communicator with new cartesian topology (handle)

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
                   int reorder, MPI_Comm *comm_cart)
```

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
                INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
                LOGICAL PERIODS(*), REORDER
```

MPI_CART_CREATE returns a handle to a new communicator to which the cartesian topology information is attached. If **reorder** = **false** then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the cartesian grid is smaller than the size of the group of **comm**, then some processes are returned **MPI_COMM_NULL**, in analogy to **MPI_COMM_SPLIT**. The call is erroneous if it specifies a grid that is larger than the group size.

6.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE

For cartesian topologies, the function **MPI_DIMS_CREATE** helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of **MPI_COMM_WORLD**'s group) into an *n*-dimensional topology.

MPI_DIMS_CREATE(nnodes, ndims, dims)

IN	nnodes	number of nodes in a grid (integer)
IN	ndims	number of cartesian dimensions (integer)
INOUT	dims	integer array of size ndims specifying the number of nodes in each dimension

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

```

MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR

```

The entries in the array **dims** are set to describe a cartesian grid with **ndims** dimensions and a total of **nnodes** nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array **dims**. If **dims[i]** is set to a positive number, the routine will not modify the number of nodes in dimension **i**; only those entries where **dims[i] = 0** are modified by the call.

Negative input values of **dims[i]** are erroneous. An error will occur if **nnodes** is not a multiple of $\prod_{i, \text{dims}[i] \neq 0} \text{dims}[i]$.

For **dims[i]** set by the call, **dims[i]** will be ordered in non-increasing order. Array **dims** is suitable for use as input to routine **MPI_CART_CREATE**. **MPI_DIMS_CREATE** is local.

Example 6.1

dims before call	function call	dims on return
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

6.5.3 General (Graph) Constructor

```

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

```

IN	comm_old	input communicator (handle)
IN	nnodes	number of nodes in graph (integer)
IN	index	array of integers describing node degrees (see below)
IN	edges	array of integers describing graph edges (see below)
IN	reorder	ranking may be reordered (true) or not (false) (logical)
OUT	comm_graph	communicator with graph topology added (handle)

```

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
    int reorder, MPI_Comm *comm_graph)

```

```

MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
    IERROR)

```

```

    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER

```

MPI_GRAPH_CREATE returns a handle to a new communicator to which the graph topology information is attached. If **reorder = false** then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the

processes. If the size, **nnodes**, of the graph is smaller than the size of the group of **comm**, then some processes are returned **MPI_COMM_NULL**, in analogy to **MPI_CART_CREATE** and **MPI_COMM_SPLIT**. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters **nnodes**, **index** and **edges** define the graph structure. **nnodes** is the number of nodes of the graph. The nodes are numbered from 0 to **nnodes**-1. The *i*th entry of array **index** stores the total number of neighbors of the first *i* graph nodes. The lists of neighbors of nodes 0, 1, ..., **nnodes**-1 are stored in consecutive locations in array **edges**. The array **edges** is a flattened representation of the edge lists. The total number of entries in **index** is **nnodes** and the total number of entries in **edges** is equal to the number of graph edges.

The definitions of the arguments **nnodes**, **index**, and **edges** are illustrated with the following simple example.

Example 6.2 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 $0 \leq j \leq \text{index}[0] - 1$ and the list of neighbors of node *i*, *i* > 0, is stored in **edges**[*j*], **index**[*i* - 1] ≤ *j* ≤ **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 $1 \leq j \leq \text{index}(1)$ and the list of neighbors of node *i*, *i* > 0, is stored in **edges**(*j*), **index**(*i*) + 1 ≤ *j* ≤ **index**(*i* + 1).

Advice to implementors. The following topology information is likely to be stored with a communicator:

- Type of topology (cartesian/graph),
- For a cartesian topology:
 1. **ndims** (number of dimensions),
 2. **dims** (numbers of processes per coordinate direction),
 3. **periods** (periodicity information),
 4. **own_position** (own position in grid, could also be computed from rank and **dims**)
- For a graph topology:

1. `index`,
 2. `edges`,
- which are the vectors defining the graph structure.

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array `index` simplifies access to the topology information. (*End of advice to implementors.*)

6.5.4 Topology inquiry functions

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.

`MPI_TOPO_TEST(comm, status)`

IN	<code>comm</code>	communicator (handle)
OUT	<code>status</code>	topology type of communicator <code>comm</code> (choice)

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

```
MPI_TOPO_TEST(COMM, STATUS, IERROR)
```

```
INTEGER COMM, STATUS, IERROR
```

The function `MPI_TOPO_TEST` returns the type of topology that is assigned to a communicator.

The output value `status` is one of the following:

<code>MPI_GRAPH</code>	graph topology
<code>MPI_CART</code>	cartesian topology
<code>MPI_UNDEFINED</code>	no topology

`MPI_GRAPHDIMS_GET(comm, nnodes, nedges)`

IN	<code>comm</code>	communicator for group with graph structure (handle)
OUT	<code>nnodes</code>	number of nodes in graph (integer) (same as number of processes in the group)
OUT	<code>nedges</code>	number of edges in graph (integer)

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
```

```
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
```

```
INTEGER COMM, NNODES, NEDGES, IERROR
```

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`.

```
1 MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)
```

2	IN	comm	communicator with graph structure (handle)
3	IN	maxindex	length of vector index in the calling program
4			(integer)
5	IN	maxedges	length of vector edges in the calling program
6			(integer)
7	OUT	index	array of integers containing the graph structure (for
8			details see the definition of MPI_GRAPH_CREATE)
9	OUT	edges	array of integers containing the graph structure

```
10
11
12
13 int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
14                  int *edges)
```

```
15 MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
16     INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
17
```

```
18
19 MPI_CARTDIM_GET(comm, ndims)
```

20	IN	comm	communicator with cartesian structure (handle)
21	OUT	ndims	number of dimensions of the cartesian structure (inte-
22			ger)

```
23
24
25 int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

```
26
27 MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
28     INTEGER COMM, NDIMS, IERROR
29
```

30 The functions MPI_CARTDIM_GET and MPI_CART_GET return the cartesian topology
31 information that was associated with a communicator by MPI_CART_CREATE.

```
32
33 MPI_CART_GET(comm, maxdims, dims, periods, coords)
```

34	IN	comm	communicator with cartesian structure (handle)
35	IN	maxdims	length of vectors dims , periods , and coords in the
36			calling program (integer)
37	OUT	dims	number of processes for each cartesian dimension (ar-
38			ray of integer)
39	OUT	periods	periodicity (true/false) for each cartesian dimension
40			(array of logical)
41	OUT	coords	coordinates of calling process in cartesian structure
42			(array of integer)

```
43
44
45
46 int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
47                  int *coords)
48
```

```

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)

```

```

MPI_CART_RANK(comm, coords, rank)

```

IN	comm	communicator with cartesian structure (handle)
IN	coords	integer array (of size ndims) specifying the cartesian coordinates of a process
OUT	rank	rank of specified process (integer)

```

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

```

```

MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR

```

For a process group with cartesian structure, the function `MPI_CART_RANK` translates the logical process coordinates to process ranks as they are used by the point-to-point routines.

For dimension `i` with `periods(i) = true`, if the coordinate, `coords(i)`, is out of range, that is, `coords(i) < 0` or `coords(i) ≥ dims(i)`, it is shifted back to the interval $0 \leq \text{coords}(i) < \text{dims}(i)$ automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

```

MPI_CART_COORDS(comm, rank, maxdims, coords)

```

IN	comm	communicator with cartesian structure (handle)
IN	rank	rank of a process within group of <code>comm</code> (integer)
IN	maxdims	length of vector <code>coord</code> in the calling program (integer)
OUT	coords	integer array (of size ndims) containing the cartesian coordinates of specified process (integer)

```

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

```

```

MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR

```

The inverse mapping, rank-to-coordinates translation is provided by `MPI_CART_COORDS`.

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

IN **comm** communicator with graph topology (handle)
 IN **rank** rank of process in group of **comm** (integer)
 OUT **nneighbors** number of neighbors of specified process (integer)

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
INTEGER COMM, RANK, NNEIGHBORS, IERROR

MPI_GRAPH_NEIGHBORS_COUNT and **MPI_GRAPH_NEIGHBORS** provide adjacency information for a general, graph topology.

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

IN **comm** communicator with graph topology (handle)
 IN **rank** rank of process in group of **comm** (integer)
 IN **maxneighbors** size of array **neighbors** (integer)
 OUT **neighbors** ranks of processes that are neighbors to specified process (array of integer)

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)

MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR

Example 6.3 Suppose that **comm** is a communicator with a shuffle-exchange topology. The group has 2^n members. Each process is labeled by a_1, \dots, a_n with $a_i \in \{0, 1\}$, and has three neighbors: $\text{exchange}(a_1, \dots, a_n) = a_1, \dots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), $\text{shuffle}(a_1, \dots, a_n) = a_2, \dots, a_n, a_1$, and $\text{unshuffle}(a_1, \dots, a_n) = a_n, a_1, \dots, a_{n-1}$. The graph adjacency list is illustrated below for $n = 3$.

node	exchange neighbors(1)	shuffle neighbors(2)	unshuffle 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.

```

C  assume: each process has stored a real number A.
C  extract neighborhood information
    CALL MPI_COMM_RANK(comm, myrank, ierr)
    CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
C  perform exchange permutation
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
+    neighbors(1), 0, comm, status, ierr)
C  perform shuffle permutation
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
+    neighbors(3), 0, comm, status, ierr)
C  perform unshuffle permutation
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
+    neighbors(2), 0, comm, status, ierr)

```

6.5.5 Cartesian Shift Coordinates

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

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

IN	<code>comm</code>	communicator with cartesian structure (handle)
IN	<code>direction</code>	coordinate dimension of shift (integer)
IN	<code>disp</code>	displacement (> 0: upwards shift, < 0: downwards shift) (integer)
OUT	<code>rank_source</code>	rank of source process (integer)
OUT	<code>rank_dest</code>	rank of destination process (integer)

```

int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
    int *rank_dest)

```

```

MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR

```

The `direction` argument indicates the dimension of the shift, i.e., the coordinate which value is modified by the shift. The coordinates are numbered from 0 to `ndims-1`, when `ndims` is the number of dimensions.

Depending on the periodicity of the cartesian group in the specified coordinate direction, `MPI_CART_SHIFT` provides the identifiers for a circular or an end-off shift. In the case of an end-off shift, the value `MPI_PROC_NULL` may be returned in `rank_source` or `rank_dest`, indicating that the source or the destination for the shift is out of range.

Example 6.4 The communicator, `comm`, has a two-dimensional, periodic, cartesian topology associated with it. A two-dimensional array of `REALs` is stored one element per process, in variable `A`. One wishes to skew this array, by shifting column `i` (vertically, i.e., along the column) by `i` steps.

```

.....
C find process rank
  CALL MPI_COMM_RANK(comm, rank, ierr)
C find cartesian coordinates
  CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
  CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
C skew array
  CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
+                           status, ierr)

```

Advice to users. In Fortran, the dimension indicated by `DIRECTION = i` has `DIMS(i+1)` nodes, where `DIMS` is the array that was used to create the grid. In C, the dimension indicated by `direction = i` is the dimension specified by `dims[i]`. (*End of advice to users.*)

6.5.6 Partitioning of Cartesian structures

`MPI_CART_SUB(comm, remain_dims, newcomm)`

IN	<code>comm</code>	communicator with cartesian structure (handle)
IN	<code>remain_dims</code>	the <code>i</code> th entry of <code>remain_dims</code> specifies whether the <code>i</code> th dimension is kept in the subgrid (true) or is dropped (false) (logical vector)
OUT	<code>newcomm</code>	communicator containing the subgrid that includes the calling process (handle)

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
```

```
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
```

```
  INTEGER COMM, NEWCOMM, IERROR
  LOGICAL REMAIN_DIMS(*)
```

If a cartesian topology has been created with `MPI_CART_CREATE`, the function `MPI_CART_SUB` can be used to partition the communicator group into subgroups that form lower-dimensional cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid cartesian topology. (This function is closely related to `MPI_COMM_SPLIT`.)

Example 6.5 Assume that `MPI_CART_CREATE(..., comm)` has defined a $(2 \times 3 \times 4)$ grid. Let `remain_dims = (true, false, true)`. Then a call to,

```
MPI_CART_SUB(comm, remain_dims, comm_new),
```

will create three communicators each with eight processes in a 2×4 cartesian topology. If `remain_dims = (false, false, true)` then the call to `MPI_CART_SUB(comm, remain_dims, comm_new)` will create six non-overlapping communicators, each with four processes, in a one-dimensional cartesian topology.

6.5.7 Low-level topology functions

The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, unless he or she is creating additional virtual topology capability other than that provided by MPI.

`MPI_CART_MAP(comm, ndims, dims, periods, newrank)`

IN	<code>comm</code>	input communicator (handle)
IN	<code>ndims</code>	number of dimensions of cartesian structure (integer)
IN	<code>dims</code>	integer array of size <code>ndims</code> specifying the number of processes in each coordinate direction
IN	<code>periods</code>	logical array of size <code>ndims</code> specifying the periodicity specification in each coordinate direction
OUT	<code>newrank</code>	reordered rank of the calling process; <code>MPI_UNDEFINED</code> if calling process does not belong to grid (integer)

```
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
                 int *newrank)
```

```
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
```

```
INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
```

```
LOGICAL PERIODS(*)
```

`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.

Advice to implementors. The function `MPI_CART_CREATE(comm, ndims, dims, periods, reorder, comm_cart)`, with `reorder = true` can be implemented by calling `MPI_CART_MAP(comm, ndims, dims, periods, newrank)`, then calling `MPI_COMM_SPLIT(comm, color, key, comm_cart)`, with `color = 0` if `newrank ≠ MPI_UNDEFINED`, `color = MPI_UNDEFINED` otherwise, and `key = newrank`.

The function `MPI_CART_SUB(comm, remain_dims, comm_new)` can be implemented by a call to `MPI_COMM_SPLIT(comm, color, key, comm_new)`, using a single number encoding of the lost dimensions as `color` and a single number encoding of the preserved dimensions as `key`.

All other cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (*End of advice to implementors.*)

The corresponding new function for general graph structures is as follows.

```

1  MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)
2      IN      comm      input communicator (handle)
3      IN      nnodes    number of graph nodes (integer)
4      IN      index     integer array specifying the graph structure, see
5                      MPI_GRAPH_CREATE
6
7      IN      edges     integer array specifying the graph structure
8
9      OUT     newrank    reordered rank of the calling process; MPI_UNDEFINED
10                      if the calling process does not belong to graph (inte-
11                      ger)
12
13  int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
14                  int *newrank)
15
16  MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
17      INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR

```

Advice to implementors. The function `MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph)`, with `reorder = true` can be implemented by calling `MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)`, then calling `MPI_COMM_SPLIT(comm, color, key, comm_graph)`, with `color = 0` if `newrank ≠ 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.*)

6.6 An Application Example

Example 6.6 The example in figure 6.1 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 all points owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the exchange subroutine might contain a call like `MPI_SEND(...,neigh_rank(1),...)` to send updated values to the left-hand neighbor (`i-1,j`).

```

integer ndims, num_neigh
logical reorder
parameter (ndims=2, num_neigh=4, reorder=.true.)
integer comm, comm_cart, dims(ndims), neigh_def(ndims), ierr
integer neigh_rank(num_neigh), own_position(ndims), i, j
logical periods(ndims)
real*8 u(0:101,0:101), f(0:101,0:101)
data dims / ndims * 0 /
comm = MPI_COMM_WORLD
C   Set process grid size and periodicity
call MPI_DIMS_CREATE(comm, ndims, dims,ierr)
periods(1) = .TRUE.
periods(2) = .TRUE.
C   Create a grid structure in WORLD group and inquire about own position
call MPI_CART_CREATE (comm, ndims, dims, periods, reorder, comm_cart,ierr)
call MPI_CART_GET (comm_cart, ndims, dims, periods, own_position,ierr)
C   Look up the ranks for the neighbors. Own process coordinates are (i,j).
C   Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1)
i = own_position(1)
j = own_position(2)
neigh_def(1) = i-1
neigh_def(2) = j
call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1),ierr)
neigh_def(1) = i+1
neigh_def(2) = j
call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2),ierr)
neigh_def(1) = i
neigh_def(2) = j-1
call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3),ierr)
neigh_def(1) = i
neigh_def(2) = j+1
call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4),ierr)
C   Initialize the grid functions and start the iteration
call init (u, f)
do 10 it=1,100
    call relax (u, f)
C   Exchange data with neighbor processes
    call exchange (u, comm_cart, neigh_rank, num_neigh)
10 continue
call output (u)
end

```

Figure 6.1: Set-up of process structure for two-dimensional parallel Poisson solver.

Chapter 7

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.

7.1 Implementation information

7.1.1 Environmental Inquiries

A set of attributes that describe the execution environment are attached to the communicator `MPI_COMM_WORLD` when MPI is initialized. The value of these attributes can be inquired by using the function `MPI_ATTR_GET` described in Chapter 5. It is erroneous to delete these attributes, free their keys, or change their values.

The list of predefined attribute keys include

`MPI_TAG_UB` Upper bound for tag value.

`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.

`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.)

These predefined attributes do not change value between MPI initialization (`MPI_INIT` and MPI completion (`MPI_FINALIZE`), and cannot be updated or deleted by users.

Advice to users. Note that in the C binding, the value returned by these attributes is a *pointer* to an `int` containing the requested value. (*End of advice to users.*)

The required parameter values are discussed in more detail below:

Tag values

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 legal value for `MPI_TAG_UB`.

The attribute `MPI_TAG_UB` has the same value on all processes of `MPI_COMM_WORLD`.

Host rank

The value returned for `MPI_HOST` gets the rank of the `HOST` process in the group associated with communicator `MPI_COMM_WORLD`, if there is such. `MPI_PROC_NULL` is returned if there is no host. MPI does not specify what it means for a process to be a `HOST`, nor does it require that a `HOST` exists.

The attribute `MPI_HOST` has the same value on all processes of `MPI_COMM_WORLD`.

IO rank

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 ANSI-C I/O operations are supported (e.g., `fopen`, `fprintf`, `lseek`).

If every process can provide language-standard I/O, then the value `MPI_ANY_SOURCE` will be returned. Otherwise, if the calling process can provide language-standard I/O, then its rank will be returned. Otherwise, if some process can provide language-standard I/O then the rank of one such process will be returned. The same value need not be returned by all processes. If no process can provide language-standard I/O, then the value `MPI_PROC_NULL` will be returned.

Advice to users. Note that input is not collective, and this attribute does *not* indicate which process can or does provide input. (*End of advice to users.*)

Clock synchronization

The value returned for `MPI_WTIME_IS_GLOBAL` is 1 if clocks at all processes in `MPI_COMM_WORLD` are synchronized, 0 otherwise. A collection of clocks is considered synchronized if explicit effort has been taken to synchronize them. The expectation is that the variation in time, as measured by calls to `MPI_WTIME`, will be less than one half the round-trip time for an MPI message of length zero. If time is measured at a process just before a send and at another process just after a matching receive, the second time should be always higher than the first one.

The attribute `MPI_WTIME_IS_GLOBAL` need not be present when the clocks are not synchronized (however, the attribute key `MPI_WTIME_IS_GLOBAL` is always valid). This attribute may be associated with communicators other than `MPI_COMM_WORLD`.

The attribute `MPI_WTIME_IS_GLOBAL` has the same value on all processes of `MPI_COMM_WORLD`.


```
1 MPI_GET_PROCESSOR_NAME( name, resultlen )
```

```
2     OUT      name                A unique specifier for the actual (as opposed to vir-
3                                     tual) node.
```

```
4     OUT      resultlen           Length (in printable characters) of the result returned
5                                     in name
6
```

```
7
8 int MPI_Get_processor_name(char *name, int *resultlen)
```

```
9 MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
```

```
10 CHARACTER*(*) NAME
```

```
11 INTEGER RESULTLEN, IERROR
12
```

13 This routine returns the name of the processor on which it was called at the moment
 14 of the call. The name is a character string for maximum flexibility. From this value it
 15 must be possible to identify a specific piece of hardware; possible values include “processor
 16 9 in rack 4 of mpp.cs.org” and “231” (where 231 is the actual processor number in the
 17 running homogeneous system). The argument **name** must represent storage that is at least
 18 MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up
 19 to this many characters into **name**.

20 The number of characters actually written is returned in the output argument, **resultlen**.

21
 22 *Rationale.* This function allows MPI implementations that do process migration
 23 to return the current processor. Note that nothing in MPI *requires* or defines pro-
 24 cess migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such
 25 an implementation. (*End of rationale.*)

26
 27 *Advice to users.* The user must provide at least MPI_MAX_PROCESSOR_NAME space
 28 to write the processor name — processor names can be this long. The user should
 29 examine the output argument, **resultlen**, to determine the actual length of the name.
 30 (*End of advice to users.*)

31
 32 The constant MPI_BSEND_OVERHEAD provides an upper bound on the fixed overhead
 33 per message buffered by a call to MPI_BSEND (see Section 3.6.1).

34 7.2 Error handling

35
 36 An MPI implementation cannot or may choose not to handle some errors that occur during
 37 MPI calls. These can include errors that generate exceptions or traps, such as floating point
 38 errors or access violations. The set of errors that are handled by MPI is implementation-
 39 dependent. Each such error generates an **MPI exception**.

40
 41 The above text takes precedence over any text on error handling within this document.
 42 Specifically, text that states that errors *will* be handled should be read as *may* be handled.

43 A user can associate an error handler with a communicator. The specified error han-
 44 dling routine will be used for any MPI exception that occurs during a call to MPI for a
 45 communication with this communicator. MPI calls that are not related to any communica-
 46 tor are considered to be attached to the communicator MPI_COMM_WORLD. The attachment
 47 of error handlers to communicators is purely local: different processes may attach different
 48 error handlers to the same communicator.

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.

Several predefined error handlers are available in MPI:

MPI_ERRORS_ARE_FATAL The handler, when called, causes the program to abort on all executing processes. This has the same effect as if `MPI_ABORT` was called by the process that invoked the handler.

MPI_ERRORS_RETURN The handler has no effect other than returning the error code to the user.

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

The error handler `MPI_ERRORS_ARE_FATAL` is associated by default with `MPI_COMM_WORLD` after initialization. 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, the state of MPI is undefined. That is, using a user-defined error handler, or `MPI_ERRORS_RETURN`, does *not* necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so.

Advice to implementors. A good quality implementation will, to the greatest possible, extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors. (*End of advice to implementors.*)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with communicators, and to test which error handler is associated with a communicator.

MPI_ERRHANDLER_CREATE(function, errhandler)

IN	function	user defined error handling procedure
OUT	errhandler	MPI error handler (handle)

```
int MPI_Errhandler_create(MPI_Handler_function *function,
                        MPI_Errhandler *errhandler)
```

MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)

EXTERNAL FUNCTION
 INTEGER ERRHANDLER, IERROR

Register the user routine **function** for use as an MPI exception handler. Returns in **errhandler** a handle to the registered exception handler.

In the C language, the user routine should be a C function of type `MPI_Handler_function`, which is defined as:

```
typedef void (MPI_Handler_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 “**stdargs**” 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.

Rationale. The variable argument list is provided because it provides an ANSI-standard hook for providing additional information to the error handler; without this hook, ANSI C prohibits additional arguments. (*End of rationale.*)

MPI_ERRHANDLER_SET(comm, errhandler)

IN	comm	communicator to set the error handler for (handle)
IN	errhandler	new MPI error handler for communicator (handle)

```
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
```

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
 INTEGER COMM, ERRHANDLER, IERROR

Associates the new error handler **errorhandler** with communicator **comm** at the calling process. Note that an error handler is always associated with the communicator.

MPI_ERRHANDLER_GET(comm, errhandler)

IN	comm	communicator to get the error handler from (handle)
OUT	errhandler	MPI error handler currently associated with communicator (handle)

```
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
```

MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
 INTEGER COMM, ERRHANDLER, IERROR

Returns in **errhandler** (a handle to) the error handler that is currently associated with communicator **comm**.

Example: A library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

MPI_ERRHANDLER_FREE(errhandler)

IN **errhandler** MPI error handler (handle)

int **MPI_Errhandler_free**(MPI_Errhandler *errhandler)

MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)

INTEGER ERRHANDLER, IERROR

Marks the error handler associated with **errhandler** for deallocation and sets **errhandler** to **MPI_ERRHANDLER_NULL**. The error handler will be deallocated after all communicators associated with it have been deallocated.

MPI_ERROR_STRING(errorcode, string, resultlen)

IN **errorcode** Error code returned by an MPI routine

OUT **string** Text that corresponds to the **errorcode**

OUT **resultlen** Length (in printable characters) of the result returned in **string**

int **MPI_Error_string**(int errorcode, char *string, int *resultlen)

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)

INTEGER ERRORCODE, RESULTLEN, IERROR

CHARACTER*(*) STRING

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.*)

7.3 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 include

1	<code>MPI_SUCCESS</code>	No error
2	<code>MPI_ERR_BUFFER</code>	Invalid buffer pointer
3	<code>MPI_ERR_COUNT</code>	Invalid count argument
4	<code>MPI_ERR_TYPE</code>	Invalid datatype argument
5	<code>MPI_ERR_TAG</code>	Invalid tag argument
6	<code>MPI_ERR_COMM</code>	Invalid communicator
7	<code>MPI_ERR_RANK</code>	Invalid rank
8	<code>MPI_ERR_REQUEST</code>	Invalid request (handle)
9	<code>MPI_ERR_ROOT</code>	Invalid root
10	<code>MPI_ERR_GROUP</code>	Invalid group
11	<code>MPI_ERR_OP</code>	Invalid operation
12	<code>MPI_ERR_TOPOLOGY</code>	Invalid topology
13	<code>MPI_ERR_DIMS</code>	Invalid dimension argument
14	<code>MPI_ERR_ARG</code>	Invalid argument of some other kind
15	<code>MPI_ERR_UNKNOWN</code>	Unknown error
16	<code>MPI_ERR_TRUNCATE</code>	Message truncated on receive
17	<code>MPI_ERR_OTHER</code>	Known error not in this list
18	<code>MPI_ERR_INTERN</code>	Internal MPI (implementation) error
19	<code>MPI_ERR_IN_STATUS</code>	Error code is in status
20	<code>MPI_ERR_PENDING</code>	Pending request
21	<code>MPI_ERR_LASTCODE</code>	Last error code

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 error codes satisfy,

$$0 = \text{MPI_SUCCESS} < \text{MPI_ERR_...} \leq \text{MPI_ERR_LASTCODE}.$$

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.*)

`MPI_ERROR_CLASS(errorcode, errorclass)`

IN	<code>errorcode</code>	Error code returned by an MPI routine
OUT	<code>errorclass</code>	Error class associated with <code>errorcode</code>

`int MPI_Error_class(int errorcode, int *errorclass)`

`MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)`
`INTEGER ERRORCODE, ERRORCLASS, IERROR`

The function `MPI_ERROR_CLASS` maps each standard error code (error class) onto itself.

7.4 Timers and synchronization

MPI defines a timer. A timer is specified even though it is not “message-passing,” because timing parallel programs is important in “performance debugging” and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers.

MPI_WTIME()

```
double MPI_Wtime(void)
```

```
DOUBLE PRECISION MPI_WTIME()
```

MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.

The “time in the past” is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not “ticks”), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```
{
    double starttime, endtime;
    starttime = MPI_Wtime();
    .... stuff to be timed ...
    endtime = MPI_Wtime();
    printf("That took %f seconds\n",endtime-starttime);
}
```

The times returned are local to the node that called them. There is no requirement that different nodes return “the same time.” (But see also the discussion of MPI_WTIME_IS_GLOBAL).

MPI_WTICK()

```
double MPI_Wtick(void)
```

```
DOUBLE PRECISION MPI_WTICK()
```

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} .

7.5 Startup

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from

the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine `MPI_INIT`.

`MPI_INIT()`

```
int MPI_Init(int *argc, char ***argv)
```

```
MPI_INIT(IERROR)
```

```
INTEGER IERROR
```

This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous (see `MPI_INITIALIZED`).

All MPI programs must contain a call to `MPI_INIT`; this routine must be called before any other MPI routine (apart from `MPI_INITIALIZED`) is called. The version for ANSI C accepts the `argc` and `argv` that are provided by the arguments to `main`:

```
int main(argc, argv)
```

```
int argc;
```

```
char **argv;
```

```
{
```

```
    MPI_Init(&argc, &argv);
```

```
    /* parse arguments */
```

```
    /* main program    */
```

```
    MPI_Finalize();    /* see below */
```

```
}
```

The Fortran version takes only `IERROR`.

An MPI implementation is free to require that the arguments in the C binding must be the arguments to `main`.

Rationale. The command line arguments are provided to `MPI_Init` to allow an MPI implementation to use them in initializing the MPI environment. They are passed by reference to allow an MPI implementation to *provide* them in environments where the command-line arguments are not provided to `main`. (*End of rationale.*)

`MPI_FINALIZE()`

```
int MPI_Finalize(void)
```

```
MPI_FINALIZE(IERROR)
```

```
INTEGER IERROR
```

This routine cleans up all MPI state. Once this routine is called, no MPI routine (even `MPI_INIT`) may be called. The user must ensure that all pending communications involving a process completes before the process calls `MPI_FINALIZE`.

`MPI_INITIALIZED(flag)`

OUT `flag` Flag is true if `MPI_INIT` has been called and false otherwise.

`int MPI_Initialized(int *flag)`

`MPI_INITIALIZED(FLAG, IERROR)`

LOGICAL FLAG

INTEGER IERROR

This routine may be used to determine whether `MPI_INIT` has been called. It is the *only* routine that may be called before `MPI_INIT` is called.

`MPI_ABORT(comm, errorcode)`

IN `comm` communicator of tasks to abort

IN `errorcode` error code to return to invoking environment

`int MPI_Abort(MPI_Comm comm, int errorcode)`

`MPI_ABORT(COMM, ERRORCODE, IERROR)`

INTEGER COMM, ERRORCODE, IERROR

This routine makes a “best attempt” to abort all tasks in the group of `comm`. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a `return errorcode` from the main program or an `abort(errorcode)`.

MPI implementations are required to define the behavior of `MPI_ABORT` at least for a `comm` of `MPI_COMM_WORLD`. MPI implementations may ignore the `comm` argument and act as if the `comm` was `MPI_COMM_WORLD`.

Rationale. The communicator argument is provided to allow for future extensions of MPI to environments with, for example, dynamic process management. In particular, it allows but does not require an MPI implementation to abort a subset of `MPI_COMM_WORLD`. (*End of rationale.*)

Chapter 8

Profiling Interface

8.1 Requirements

To meet the MPI profiling interface, an implementation of the MPI functions *must*

1. provide a mechanism through which all of the MPI defined functions may be accessed with a name shift. Thus all of the MPI functions (which normally start with the prefix “MPI_”) should also be accessible with the prefix “PMPI_”.
2. ensure that those MPI functions which are not replaced may still be linked into an executable image without causing name clashes.
3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economise by implementing it only for the lowest level routines.
4. where the implementation of different language bindings is done through a layered approach (e.g. the Fortran binding is a set of “wrapper” functions which call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

This is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

5. provide a no-op routine MPI_PCONTROL in the MPI library.

8.2 Discussion

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to

the source code which implements **MPI** on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

We believe that having such an interface is important if **MPI** is to be attractive to end users, since the availability of many different tools will be a significant factor in attracting users to the **MPI** standard.

The profiling interface is just that, an interface. It says *nothing* about the way in which it is used. There is therefore no attempt to lay down what information is collected through the interface, or how the collected information is saved, filtered, or displayed.

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as “internetworking” multiple **MPI** implementations. Since all that is defined is an interface, there is no objection to its being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which executable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the **MPI** profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this chapter is only present as justification and discussion of the logic for those requirements.

The examples below show one way in which an implementation could be constructed to meet the requirements on a Unix system (there are doubtless others which would be equally valid).

8.3 Logic of the design

Provided that an **MPI** implementation meets the requirements above, it is possible for the implementor of the profiling system to intercept all of the **MPI** calls which are made by the user program. She can then collect whatever information she requires before calling the underlying **MPI** implementation (through its name shifted entry points) to achieve the desired effects.

8.3.1 Miscellaneous control of profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at non-critical points in the calculation
- Adding user events to a trace file.

These requirements are met by use of the **MPI_PCONTROL**.

MPI_PCONTROL(level, ...)

IN level Profiling level

int **MPI_Pcontrol**(const int level, ...)

```

1 MPI_PCONTROL(level)
2     INTEGER LEVEL, ...

```

MPI libraries themselves make no use of this routine, and simply return immediately to the user code. However the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, we are unable to specify precisely the semantics which will be provided by calls to `MPI_PCONTROL`. This vagueness extends to the number of arguments to the function, and their datatypes.

However to provide some level of portability of user codes to different profiling libraries, we request the following meanings for certain values of `level`.

- `level==0` Profiling is disabled.
- `level==1` Profiling is enabled at a normal default level of detail.
- `level==2` Profile buffers are flushed. (This may be a no-op in some profilers).
- All other values of `level` have profile library defined effects and additional arguments.

We also request that the default state after `MPI_INIT` has been called is for profiling to be enabled at the normal default level. (i.e. as if `MPI_PCONTROL` had just been called with the argument 1). This allows users to link with a profiling library and obtain profile output without having to modify their source code at all.

The provision of `MPI_PCONTROL` as a no-op in the standard MPI library allows them to modify their source code to obtain more detailed profiling information, but still be able to link exactly the same code against the standard MPI library.

8.4 Examples

8.4.1 Profiler implementation

Suppose that the profiler wishes to accumulate the total amount of data sent by the `MPI_SEND` function, along with the total elapsed time spent in the function. This could trivially be achieved thus

```

34 static int totalBytes;
35 static double totalTime;
36
37 int MPI_SEND(void * buffer, const int count, MPI_Datatype datatype,
38             int dest, int tag, MPI_comm comm)
39 {
40     double tstart = MPI_Wtime();      /* Pass on all the arguments */
41     int extent;
42     int result    = PMPI_Send(buffer, count, datatype, dest, tag, comm);
43
44     MPI_Type_size(datatype, &extent); /* Compute size */
45     totalBytes += count*extent;
46
47     totalTime += MPI_Wtime() - tstart; /* and time */
48

```

```

    return result;
}

```

8.4.2 MPI library implementation

On a Unix system, in which the MPI library is implemented in C, then there are various possible options, of which two of the most obvious are presented here. Which is better depends on whether the linker and compiler support weak symbols.

Systems with weak symbols

If the compiler and linker support weak external symbols (e.g. Solaris 2.x, other system V.4 machines), then only a single library is required through the use of `#pragma weak` thus

```

#pragma weak MPI_Example = PMPI_Example

int PMPI_Example(/* appropriate args */)
{
    /* Useful content */
}

```

The effect of this `#pragma` is to define the external symbol `MPI_Example` as a weak definition. This means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library), however if no other definition exists, then the linker will use the weak definition.

Systems without weak symbols

In the absence of weak symbols then one possible solution would be to use the C macro pre-processor thus

```

#ifdef PROFILELIB
#   ifdef __STDC__
#       define FUNCTION(name) P##name
#   else
#       define FUNCTION(name) P/**/name
#   endif
#else
#   define FUNCTION(name) name
#endif

```

Each of the user visible functions in the library would then be declared thus

```

int FUNCTION(MPI_Example)(/* appropriate args */)
{
    /* Useful content */
}

```

The same source file can then be compiled to produce both versions of the library, depending on the state of the `PROFILELIB` macro symbol.

It is required that the standard `MPI` library be built in such a way that the inclusion of `MPI` functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those `MPI` functions which she wishes to intercept, references to any others being fulfilled by the normal `MPI` library. Therefore the link step can look something like this

```
% cc ... -lmyprof -lpmpi -lmpi
```

Here `libmyprof.a` contains the profiler functions which intercept some of the `MPI` functions. `libpmpi.a` contains the “name shifted” `MPI` functions, and `libmpi.a` contains the normal definitions of the `MPI` functions.

8.4.3 Complications

Multiple counting

Since parts of the `MPI` library may themselves be implemented using more basic `MPI` functions (e.g. a portable implementation of the collective operations implemented using point to point communications), there is potential for profiling functions to be called from within an `MPI` function which was called from a profiling function. This could lead to “double counting” of the time spent in the inner routine. Since this effect could actually be useful under some circumstances (e.g. it might allow one to answer the question “How much time is spent in the point to point routines when they’re called from collective functions ?”), we have decided not to enforce any restrictions on the author of the `MPI` library which would overcome this. Therefore the author of the profiling library should be aware of this problem, and guard against it herself. In a single threaded world this is easily achieved through use of a static variable in the profiling code which remembers if you are already inside a profiling routine. It becomes more complex in a multi-threaded environment (as does the meaning of the times recorded !)

Linker oddities

The Unix linker traditionally operates in one pass : the effect of this is that functions from libraries are only included in the image if they are needed at the time the library is scanned. When combined with weak symbols, or multiple definitions of the same function, this can cause odd (and unexpected) effects.

Consider, for instance, an implementation of `MPI` in which the Fortran binding is achieved by using wrapper functions on top of the C implementation. The author of the profile library then assumes that it is reasonable only to provide profile functions for the C binding, since Fortran will eventually call these, and the cost of the wrappers is assumed to be small. However, if the wrapper functions are not in the profiling library, then none of the profiled entry points will be undefined when the profiling library is called. Therefore none of the profiling code will be included in the image. When the standard `MPI` library is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of the `MPI` functions. The overall effect is that the code will link successfully, but will not be profiled.

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be aared out of the base library and into the profiling one.

8.5 Multiple levels of interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation which would allow multiple levels of call interception, however we were unable to construct an implementation of this which did not have the following disadvantages

- assuming a particular implementation language.
- imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we decided to accept the scheme outlined above.

Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function.

Unfortunately such an implementation may require more cooperation between the different profiling libraries than is required for the single level implementation detailed above.

Bibliography

- [1] 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.
- [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.
- [3] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OOO-SKI '94*, page in press, 1994.
- [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.
- [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.
- [6] R. Butler and E. Lusk. User’s guide to the p4 programming system. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992.
- [7] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: the p4 parallel programming system. *Journal of Parallel Computing*, 1994. to appear (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493).
- [8] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the parmacs message-passing library. *Parallel Computing, Special issue on message-passing interfaces*, to appear.
- [9] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole-routed networks for interprocessor communication in multicomputers. In *Proceedings of the 1990 Supercomputing Conference*, pages 647–656, 1990.
- [10] S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software)*, pages II-1 – II-4, 1991.
- [11] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–75, April 1993.

- [12] 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.
- [13] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. A model implementation of MPI. Technical report, Argonne National Laboratory, 1993.
- [14] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991.
- [15] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992.
- [16] D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991.
- [17] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, and Marc Snir. An efficient implementation of MPI. In *1994 International Conference on Parallel Processing*, 1994.
- [18] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, October 1990.
- [19] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.
- [20] O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989.
- [21] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990.
- [22] Parasoft Corporation, Pasadena, CA. *Express User's Guide*, version 3.2.5 edition, 1992.
- [23] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.
- [24] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990.
- [25] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992.
- [26] 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.
- [27] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 1994. (Invited Paper, to appear in Special Issue on Message Passing).

- [28] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Charles H. Still, Alvin P. Leung, and Manfred Morari. Zipcode: A Portable Communication Layer for High Performance Multicomputing. Technical Report UCRL-JC-106725 (revised 9/92, 12/93, 4/94), Lawrence Livermore National Laboratory, March 1991. To appear in *Concurrency: Practice & Experience*.
- [29] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992.

Annex A

Language Binding

A.1 Introduction

In this section we summarize the specific bindings for both Fortran and C. We present first the C bindings, then the Fortran bindings. Listings are alphabetical within chapter.

A.2 Defined Constants for C and Fortran

These are required defined constants, to be defined in the files `mpi.h` (for C) and `mpif.h` (for Fortran).

```
/* return codes (both C and Fortran) */
```

```
MPI_SUCCESS
```

```
MPI_ERR_BUFFER
```

```
MPI_ERR_COUNT
```

```
MPI_ERR_TYPE
```

```
MPI_ERR_TAG
```

```
MPI_ERR_COMM
```

```
MPI_ERR_RANK
```

```
MPI_ERR_REQUEST
```

```
MPI_ERR_ROOT
```

```
MPI_ERR_GROUP
```

```
MPI_ERR_OP
```

```
MPI_ERR_TOPOLOGY
```

```
MPI_ERR_DIMS
```

```
MPI_ERR_ARG
```

```
MPI_ERR_UNKNOWN
```

```
MPI_ERR_TRUNCATE
```

```
MPI_ERR_OTHER
```

```
MPI_ERR_INTERN
```

```
MPI_PENDING
```

```
MPI_ERR_IN_STATUS
```

```
MPI_ERR_LASTCODE
```

```
/* assorted constants (both C and Fortran) */
```

```
1  MPI_BOTTOM
2  MPI_PROC_NULL
3  MPI_ANY_SOURCE
4  MPI_ANY_TAG
5  MPI_UNDEFINED
6  MPI_BSEND_OVERHEAD
7  MPI_KEYVAL_INVALID
8
9  /* status size and reserved index values (Fortran) */
10 MPI_STATUS_SIZE
11 MPI_SOURCE
12 MPI_TAG
13 MPI_ERROR
14
15 /* Error-handling specifiers (C and Fortran) */
16 MPI_ERRORS_ARE_FATAL
17 MPI_ERRORS_RETURN
18
19 /* Maximum sizes for strings */
20 MPI_MAX_PROCESSOR_NAME
21 MPI_MAX_ERROR_STRING
22
23 /* elementary datatypes (C) */
24 MPI_CHAR
25 MPI_SHORT
26 MPI_INT
27 MPI_LONG
28 MPI_UNSIGNED_CHAR
29 MPI_UNSIGNED_SHORT
30 MPI_UNSIGNED
31 MPI_UNSIGNED_LONG
32 MPI_FLOAT
33 MPI_DOUBLE
34 MPI_LONG_DOUBLE
35 MPI_BYTE
36 MPI_PACKED
37
38
39
40 /* elementary datatypes (Fortran) */
41 MPI_INTEGER
42 MPI_REAL
43 MPI_DOUBLE_PRECISION
44 MPI_COMPLEX
45 MPI_DOUBLE_COMPLEX
46 MPI_LOGICAL
47 MPI_CHARACTER
48 MPI_BYTE
```

```

MPI_PACKED 1
2
/* datatypes for reduction functions (C) */ 3
MPI_FLOAT_INT 4
MPI_DOUBLE_INT 5
MPI_LONG_INT 6
MPI_2INT 7
MPI_SHORT_INT 8
MPI_LONG_DOUBLE_INT 9
10
/* datatypes for reduction functions (Fortran) */ 11
MPI_2REAL 12
MPI_2DOUBLE_PRECISION 13
MPI_2INTEGER 14
15
/* optional datatypes (Fortran) */ 16
MPI_INTEGER1 17
MPI_INTEGER2 18
MPI_INTEGER4 19
MPI_REAL2 20
MPI_REAL4 21
MPI_REAL8 22
23
/* optional datatypes (C) */ 24
MPI_LONG_LONG_INT 25
26
/* special datatypes for constructing derived datatypes 27
MPI_UB 28
MPI_LB 29
30
/* reserved communicators (C and Fortran) */ 31
MPI_COMM_WORLD 32
MPI_COMM_SELF 33
34
/* results of communicator and group comparisons */ 35
36
MPI_IDENT 37
MPI_CONGRUENT 38
MPI_SIMILAR 39
MPI_UNEQUAL 40
41
/* environmental inquiry keys (C and Fortran) */ 42
MPI_TAG_UB 43
MPI_IO 44
MPI_HOST 45
MPI_WTIME_IS_GLOBAL 46
47
/* collective operations (C and Fortran) */ 48

```

```

1  MPI_MAX
2  MPI_MIN
3  MPI_SUM
4  MPI_PROD
5  MPI_MAXLOC
6  MPI_MINLOC
7  MPI_BAND
8  MPI_BOR
9  MPI_BXOR
10 MPI_LAND
11 MPI_LOR
12 MPI_LXOR
13
14 /* Null handles */
15 MPI_GROUP_NULL
16 MPI_COMM_NULL
17 MPI_DATATYPE_NULL
18 MPI_REQUEST_NULL
19 MPI_OP_NULL
20 MPI_ERRHANDLER_NULL
21
22 /* Empty group */
23 MPI_GROUP_EMPTY
24
25 /* topologies (C and Fortran) */
26 MPI_GRAPH
27 MPI_CART
28
29
30     The following are defined C type definitions, also included in the file mpi.h.
31
32 /* opaque types (C) */
33 MPI_Aint
34 MPI_Status
35
36 /* handles to assorted structures (C) */
37 MPI_Group
38 MPI_Comm
39 MPI_Datatype
40 MPI_Request
41 MPI_Op
42
43 /* prototypes for user-defined functions (C) */
44 typedef int  MPI_Copy_function(MPI_Comm oldcomm, int keyval,
45                               void *extra_state, void *attribute_val_in,
46                               void *attribute_val_out, int *flag);
47 typedef int  MPI_Delete_function(MPI_Comm comm, int keyval,
48                                  void *attribute_val, void *extra_state)

```

```

typedef void MPI_Handler_function(MPI_Comm *, int *, ...);
typedef void MPI_User_function( void *invec, void *inoutvec, int *len,
                                MPI_Datatype *datatype);

```

For Fortran, here are examples of how each of the user-defined functions should be declared.

The user-function argument to `MPIOP_CREATE` should be declared like this:

```

FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, TYPE

```

The copy-function argument to `MPI_KEYVAL_CREATE` should be declared like this:

```

PROCEDURE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,
                        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
      ATTRIBUTE_VAL_OUT, IERR
LOGICAL FLAG

```

The delete-function argument to `MPI_KEYVAL_CREATE` should be declared like this:

```

PROCEDURE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR

```

A.3 C bindings for Point-to-Point Communication

These are presented here in the order of their appearance in the chapter.

```

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

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

int MPI_Buffer_attach( void* buffer, int size)

int MPI_Buffer_detach( void* buffer, int* size)

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

```

```
1  int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
2      int tag, MPI_Comm comm, MPI_Request *request)
3
4  int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
5      int tag, MPI_Comm comm, MPI_Request *request)
6
7  int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
8      int tag, MPI_Comm comm, MPI_Request *request)
9
10 int MPI_Wait(MPI_Request *request, MPI_Status *status)
11
12 int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
13
14 int MPI_Request_free(MPI_Request *request)
15
16 int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
17     MPI_Status *status)
18
19 int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
20     int *flag, MPI_Status *status)
21
22 int MPI_Waitall(int count, MPI_Request *array_of_requests,
23     MPI_Status *array_of_statuses)
24
25 int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
26     MPI_Status *array_of_statuses)
27
28 int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount,
29     int *array_of_indices, MPI_Status *array_of_statuses)
30
31 int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount,
32     int *array_of_indices, MPI_Status *array_of_statuses)
33
34 int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
35     MPI_Status *status)
36
37 int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
38
39 int MPI_Cancel(MPI_Request *request)
40
41 int MPI_Test_cancelled(MPI_Status *status, int *flag)
42
43 int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
44     int tag, MPI_Comm comm, MPI_Request *request)
45
46 int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
47     int tag, MPI_Comm comm, MPI_Request *request)
48
49 int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
50     int tag, MPI_Comm comm, MPI_Request *request)
51
52 int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
53     int tag, MPI_Comm comm, MPI_Request *request)
54
55 int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
56     int tag, MPI_Comm comm, MPI_Request *request)
```



```

1  int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
2                      int *size)
3
4

```

A.4 C Bindings for Collective Communication

```

6  int MPI_Barrier(MPI_Comm comm )
7
8  int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
9                MPI_Comm comm )
10
11 int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
12               void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
13               MPI_Comm comm)
14
15 int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
16                void* recvbuf, int *recvcounts, int *displs,
17                MPI_Datatype recvtype, int root, MPI_Comm comm)
18
19 int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
20                void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
21                MPI_Comm comm)
22
23 int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
24                 MPI_Datatype sendtype, void* recvbuf, int recvcount,
25                 MPI_Datatype recvtype, int root, MPI_Comm comm)
26
27 int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
28                  void* recvbuf, int recvcount, MPI_Datatype recvtype,
29                  MPI_Comm comm)
30
31 int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
32                   void* recvbuf, int *recvcounts, int *displs,
33                   MPI_Datatype recvtype, MPI_Comm comm)
34
35 int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
36                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
37                 MPI_Comm comm)
38
39 int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
40                  MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
41                  int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
42
43 int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
44               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
45
46 int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
47
48 int MPI_Op_free( MPI_Op *op)
49
50 int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
51                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
52
53 int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,

```

```

        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )

```

A.5 C Bindings for Groups, Contexts, and Communicators

```

int MPI_Group_size(MPI_Group group, int *size)
int MPI_Group_rank(MPI_Group group, int *rank)
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
        MPI_Group group2, int *ranks2)
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
        MPI_Group *newgroup)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
        MPI_Group *newgroup)
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
        MPI_Group *newgroup)
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
        MPI_Group *newgroup)
int MPI_Group_free(MPI_Group *group)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
int MPI_Comm_free(MPI_Comm *comm)
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)

```

```

1  int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
2                          MPI_Comm peer_comm, int remote_leader, int tag,
3                          MPI_Comm *newintercomm)
4
5  int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
6                          MPI_Comm *newintracomm)
7
8  int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
9                      *delete_fn, int *keyval, void* extra_state)
10
11 int MPI_Keyval_free(int *keyval)
12
13 int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
14
15 int MPI_Attr_get(MPI_Comm comm, int keyval, void* attribute_val, int *flag)
16
17 int MPI_Attr_delete(MPI_Comm comm, int keyval)

```

A.6 C Bindings for Process Topologies

```

18
19 int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
20                   int reorder, MPI_Comm *comm_cart)
21
22 int MPI_Dims_create(int nnodes, int ndims, int *dims)
23
24 int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
25                   int reorder, MPI_Comm *comm_graph)
26
27 int MPI_Topo_test(MPI_Comm comm, int *status)
28
29 int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
30
31 int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
32                  int *edges)
33
34 int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
35
36 int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
37                 int *coords)
38
39 int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
40
41 int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
42
43 int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
44
45 int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
46                        int *neighbors)
47
48 int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
49                  int *rank_dest)
50
51 int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
52
53 int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
54                 int *newrank)

```

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
                  int *newrank)
```

A.7 C bindings for Environmental Inquiry

```
int MPI_Get_processor_name(char *name, int *resultlen)
int MPI_Errhandler_create(MPI_Handler_function *function,
                          MPI_Errhandler *errhandler)
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
int MPI_Error_string(int errorcode, char *string, int *resultlen)
int MPI_Error_class(int errorcode, int *errorclass)
int double MPI_Wtime(void)
int double MPI_Wtick(void)
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize(void)
int MPI_Initialized(int *flag)
int MPI_Abort(MPI_Comm comm, int errorcode)
```

A.8 C Bindings for Profiling

```
int MPI_Pcontrol(const int level, ...)
```

A.9 Fortran Bindings for Point-to-Point Communication

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
```

```
1      INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
2      MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
3      <type> BUF(*)
4      INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
5
6      MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
7      <type> BUF(*)
8      INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
9
10     MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR)
11     <type> BUFFER(*)
12     INTEGER SIZE, IERROR
13
14     MPI_BUFFER_DETACH( BUFFER, SIZE, IERROR)
15     <type> BUFFER(*)
16     INTEGER SIZE, IERROR
17
18     MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
19     <type> BUF(*)
20     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
21
22     MPI_IRESEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
23     <type> BUF(*)
24     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
25
26     MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
27     <type> BUF(*)
28     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
29
30     MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
31     <type> BUF(*)
32     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
33
34     MPI_Irecv(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
35     <type> BUF(*)
36     INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
37
38     MPI_WAIT(REQUEST, STATUS, IERROR)
39     INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
40
41     MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
42     LOGICAL FLAG
43     INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
44
45     MPI_REQUEST_FREE(REQUEST, IERROR)
46     INTEGER REQUEST, IERROR
47
48     MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
49     INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
50     IERROR
51
52     MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
53     LOGICAL FLAG
```

```

    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
1
2
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
3
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
4
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
5
6
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
7
    LOGICAL FLAG
8
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
9
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
10
11
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
12
    ARRAY_OF_STATUSES, IERROR)
13
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
14
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
15
16
MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
17
    ARRAY_OF_STATUSES, IERROR)
18
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
19
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
20
21
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
22
    LOGICAL FLAG
23
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
24
25
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
26
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
27
28
MPI_CANCEL(REQUEST, IERROR)
29
    INTEGER REQUEST, IERROR
30
31
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
32
    LOGICAL FLAG
33
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
34
35
MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
36
    <type> BUF(*)
37
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
38
39
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
40
    <type> BUF(*)
41
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
42
43
MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
44
    <type> BUF(*)
45
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
46
47
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
48
    <type> BUF(*)

```

```

1      INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
2      MPI_START(REQUEST, IERROR)
3      INTEGER REQUEST, IERROR
4
5      MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
6      INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
7
8      MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUFF,
9                  RECVCOUNT, RECVMYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
10     <type> SENDBUF(*), RECVBUFF(*)
11     INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVMYPE,
12     SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
13
14     MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
15                          COMM, STATUS, IERROR)
16     <type> BUF(*)
17     INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
18     STATUS(MPI_STATUS_SIZE), IERROR
19
20     MPI_TYPE_CONTIGUOUS(COUNT, OLDDTYPE, NEWTYPE, IERROR)
21     INTEGER COUNT, OLDDTYPE, NEWTYPE, IERROR
22
23     MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDDTYPE, NEWTYPE, IERROR)
24     INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDDTYPE, NEWTYPE, IERROR
25
26     MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDDTYPE, NEWTYPE, IERROR)
27     INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDDTYPE, NEWTYPE, IERROR
28
29     MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
30                     OLDDTYPE, NEWTYPE, IERROR)
31     INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
32     OLDDTYPE, NEWTYPE, IERROR
33
34     MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
35                      OLDDTYPE, NEWTYPE, IERROR)
36     INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
37     OLDDTYPE, NEWTYPE, IERROR
38
39     MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
40                    ARRAY_OF_TYPES, NEWTYPE, IERROR)
41     INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
42     ARRAY_OF_TYPES(*), NEWTYPE, IERROR
43
44     MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
45     <type> LOCATION(*)
46     INTEGER ADDRESS, IERROR
47
48     MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
49     INTEGER DATATYPE, EXTENT, IERROR
50
51     MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
52     INTEGER DATATYPE, SIZE, IERROR

```

```

MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

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

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

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

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

```

A.10 Fortran Bindings for Collective Communication

```

MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

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

MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
    RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, IERROR

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

```



```

1  MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, REVCOUNT,
2              RECVTYPE, ROOT, COMM, IERROR)
3      <type> SENDBUF(*), RECVBUF(*)
4      INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, REVCOUNT, RECVTYPE, ROOT,
5      COMM, IERROR
6
6  MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE,
7              COMM, IERROR)
8      <type> SENDBUF(*), RECVBUF(*)
9      INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM, IERROR
10
11 MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNTS, DISPLS,
12              RECVTYPE, COMM, IERROR)
13      <type> SENDBUF(*), RECVBUF(*)
14      INTEGER SENDCOUNT, SENDTYPE, REVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
15      IERROR
16
16 MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE,
17              COMM, IERROR)
18      <type> SENDBUF(*), RECVBUF(*)
19      INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM, IERROR
20
21 MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, REVCOUNTS,
22              RDISPLS, RECVTYPE, COMM, IERROR)
23      <type> SENDBUF(*), RECVBUF(*)
24      INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, REVCOUNTS(*), RDISPLS(*),
25      RECVTYPE, COMM, IERROR
26
26 MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
27      <type> SENDBUF(*), RECVBUF(*)
28      INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
29
30 MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
31      EXTERNAL FUNCTION
32      LOGICAL COMMUTE
33      INTEGER OP, IERROR
34
34 MPI_OP_FREE( OP, IERROR)
35      INTEGER OP, IERROR
36
37 MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
38      <type> SENDBUF(*), RECVBUF(*)
39      INTEGER COUNT, DATATYPE, OP, COMM, IERROR
40
41 MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, REVCOUNTS, DATATYPE, OP, COMM,
42                  IERROR)
43      <type> SENDBUF(*), RECVBUF(*)
44      INTEGER REVCOUNTS(*), DATATYPE, OP, COMM, IERROR
45
45 MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
46      <type> SENDBUF(*), RECVBUF(*)
47      INTEGER COUNT, DATATYPE, OP, COMM, IERROR
48

```

A.11 Fortran Bindings for Groups, Contexts, etc.

```

MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR

MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR

MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR

MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR

MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR

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

```

```

1  MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
2      INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
3
4  MPI_COMM_FREE(COMM, IERROR)
5      INTEGER COMM, IERROR
6
7  MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
8      INTEGER COMM, IERROR
9      LOGICAL FLAG
10
11 MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
12     INTEGER COMM, SIZE, IERROR
13
14 MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
15     INTEGER COMM, GROUP, IERROR
16
17 MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
18     NEWINTERCOMM, IERROR)
19     INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
20     NEWINTERCOMM, IERROR
21
22 MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
23     INTEGER INTERCOMM, INTRACOMM, IERROR
24     LOGICAL HIGH
25
26 MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
27     EXTERNAL COPY_FN, DELETE_FN
28     INTEGER KEYVAL, EXTRA_STATE, IERROR
29
30 MPI_KEYVAL_FREE(KEYVAL, IERROR)
31     INTEGER KEYVAL, IERROR
32
33 MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
34     INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
35
36 MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
37     INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
38     LOGICAL FLAG
39
40 MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
41     INTEGER COMM, KEYVAL, IERROR

```

A.12 Fortran Bindings for Process Topologies

```

40 MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
41     INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
42     LOGICAL PERIODS(*), REORDER
43
44 MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
45     INTEGER NNODES, NDIMS, DIMS(*), IERROR
46
47 MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
48     IERROR)

```

```

    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER
1
2
MPI_TOPO_TEST(COMM, STATUS, IERROR)
3
    INTEGER COMM, STATUS, IERROR
4
5
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
6
    INTEGER COMM, NNODES, NEDGES, IERROR
7
8
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
9
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
10
11
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
12
    INTEGER COMM, NDIMS, IERROR
13
14
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
15
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
16
    LOGICAL PERIODS(*)
17
18
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
19
    INTEGER COMM, COORDS(*), RANK, IERROR
20
21
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
22
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
23
24
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
25
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
26
27
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
28
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
29
30
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
31
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
32
33
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
34
    INTEGER COMM, NEWCOMM, IERROR
35
    LOGICAL REMAIN_DIMS(*)
36
37
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
38
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
39
    LOGICAL PERIODS(*)
40
41
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
42
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
43
44
45
46
47
48

```

A.13 Fortran Bindings for Environmental Inquiry

```

MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
42
    CHARACTER*(*) NAME
43
    INTEGER RESULTLEN, IERROR
44
45
MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)
46
    EXTERNAL FUNCTION
47
48

```

```

1      INTEGER ERRHANDLER, IERROR
2      MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
3      INTEGER COMM, ERRHANDLER, IERROR
4
5      MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
6      INTEGER COMM, ERRHANDLER, IERROR
7
8      MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
9      INTEGER ERRHANDLER, IERROR
10
11     MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
12     INTEGER ERRORCODE, RESULTLEN, IERROR
13     CHARACTER*(*) STRING
14
15     MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
16     INTEGER ERRORCODE, ERRORCLASS, IERROR
17
18     DOUBLE PRECISION MPI_WTIME()
19
20     DOUBLE PRECISION MPI_WTICK()
21
22     MPI_INIT(IERROR)
23     INTEGER IERROR
24
25     MPI_FINALIZE(IERROR)
26     INTEGER IERROR
27
28     MPI_INITIALIZED(FLAG, IERROR)
29     LOGICAL FLAG
30     INTEGER IERROR
31
32     MPI_ABORT(COMM, ERRORCODE, IERROR)
33     INTEGER COMM, ERRORCODE, IERROR

```

A.14 Fortran Bindings for Profiling

```

33     MPI_PCONTROL(level)
34     INTEGER LEVEL, ...

```

MPI Function Index

MPLABORT, 201
MPLADDRESS, 69
MPLALLGATHER, 110
MPLALLGATHERV, 111
MPLALLREDUCE, 125
MPLALLTOALL, 112
MPLALLTOALLV, 113
MPLATTR_DELETE, 172
MPLATTR_GET, 172
MPLATTR_PUT, 171
MPLBARRIER, 95
MPLBCAST, 95
MPLBSEND, 28
MPLBSEND_INIT, 56
MPLBUFFER_ATTACH, 34
MPLBUFFER_DETACH, 34
MPLCANCEL, 54
MPLCART_COORDS, 185
MPLCART_CREATE, 180
MPLCART_GET, 184
MPLCART_MAP, 189
MPLCART_RANK, 185
MPLCART_SHIFT, 187
MPLCART_SUB, 188
MPLCARTDIM_GET, 184
MPLCOMM_COMPARE, 145
MPLCOMM_CREATE, 146
MPLCOMM_DUP, 146
MPLCOMM_FREE, 148
MPLCOMM_GROUP, 140
MPLCOMM_RANK, 145
MPLCOMM_REMOTE_GROUP, 159
MPLCOMM_REMOTE_SIZE, 158
MPLCOMM_SIZE, 144
MPLCOMM_SPLIT, 147
MPLCOMM_TEST_INTER, 158
MPLDIMS_CREATE, 180
MPIERRHANDLER_CREATE, 195
MPIERRHANDLER_FREE, 197
MPIERRHANDLER_GET, 196
MPIERRHANDLER_SET, 196
MPI_ERROR_CLASS, 198
MPI_ERROR_STRING, 197
MPI_FINALIZE, 200
MPI_GATHER, 96
MPI_GATHERV, 98
MPI_GET_COUNT, 22
MPI_GET_ELEMENTS, 75
MPI_GET_PROCESSOR_NAME, 194
MPI_GRAPH_CREATE, 181
MPI_GRAPH_GET, 184
MPI_GRAPH_MAP, 190
MPI_GRAPH_NEIGHBORS, 186
MPI_GRAPH_NEIGHBORS_COUNT, 186
MPI_GRAPHDIMS_GET, 183
MPI_GROUP_COMPARE, 139
MPI_GROUP_DIFFERENCE, 141
MPI_GROUP_EXCL, 142
MPI_GROUP_FREE, 143
MPI_GROUP_INCL, 141
MPI_GROUP_INTERSECTION, 140
MPI_GROUP_RANGE_EXCL, 143
MPI_GROUP_RANGE_INCL, 142
MPI_GROUP_RANK, 138
MPI_GROUP_SIZE, 138
MPI_GROUP_TRANSLATE_RANKS, 139
MPI_GROUP_UNION, 140
MPI_IBSEND, 39
MPI_INIT, 200
MPI_INITIALIZED, 201
MPI_INTERCOMM_CREATE, 160
MPI_INTERCOMM_MERGE, 160
MPI_IPROBE, 51
MPI_Irecv, 40
MPI_IRSEND, 40
MPI_IREND, 38
MPI_ISSEND, 39
MPI_KEYVAL_CREATE, 169
MPI_KEYVAL_FREE, 171
MPI_OP_CREATE, 121

1 MPIOP_FREE, 123
2 MPL_PACK, 85
3 MPL_PACK_SIZE, 87
4 MPLPCONTROL, 203
5 MPLPROBE, 52
6 MPLRECV, 20
7 MPLRECV_INIT, 57
8 MPLREDUCE, 114
9 MPLREDUCE_SCATTER, 126
10 MPLREQUEST_FREE, 43
11 MPLRSEND, 29
12 MPLRSEND_INIT, 57
13 MPLSCAN, 127
14 MPLSCATTER, 105
15 MPLSCATTERV, 106
16 MPLSEND, 17
17 MPLSEND_INIT, 55
18 MPLSENDRECV, 59
19 MPLSENDRECV_REPLACE, 60
20 MPLSSEND, 29
21 MPLSSEND_INIT, 56
22 MPLSTART, 58
23 MPLSTARTALL, 58
24 MPLTEST, 42
25 MPLTEST_CANCELLED, 54
26 MPLTESTALL, 48
27 MPLTESTANY, 46
28 MPLTESTSOME, 49
29 MPLTOPO_TEST, 183
30 MPLTYPE_COMMIT, 72
31 MPLTYPE_CONTIGUOUS, 63
32 MPLTYPE_EXTENT, 70
33 MPLTYPE_FREE, 72
34 MPLTYPE_HINDEXED, 67
35 MPLTYPE_HVECTOR, 65
36 MPLTYPE_INDEXED, 66
37 MPLTYPE_LB, 71
38 MPLTYPE_SIZE, 70
39 MPLTYPE_STRUCT, 68
40 MPLTYPE_UB, 72
41 MPLTYPE_VECTOR, 63
42 MPLUNPACK, 86
43 MPLWAIT, 41
44 MPLWAITALL, 47
45 MPLWAITANY, 45
46 MPLWAITSOME, 48
47 MPLWTICK, 199
48 MPLWTIME, 199