D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

October 19, 1993

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.

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

This is a draft of what will become the Final Report, Version 1.0, of the Message Passing Interface Forum. This document contains all the technical features proposed for the interface. This copy of the draft was processed by LaTeX on October 19, 1993.

MPIF invites comments on the technical content of MPI, as well as on the editorial presentation in the document. Comments received before January 15, 1994 will be considered in producing the final draft of Version 1.0 of the Message Passing Interface Specification.

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 establishing a practical, portable, efficient, and flexible standard for message passing.

Contents

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:

- 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
- Rolf Hempel, Process Topologies
- Ewing Lusk, Language Binding
- William Gropp, Environmental Management and Inquiry
- James Cownie, Profiling
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Marc Sears, Groups, Contexts, and Communicators
- Steven Huss-Lederman, Initial Implementation Subset
- Steve Otto, Editor

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

Ed Anderson	Joe Baron	Eric Barszcz	Scott Berryman
Rob Bjornson	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
John Kapenga	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	

Oak Ridge National Laboratory made the draft available by anonymous FTP mail servers and was instrumental in distributing the document.

MPI operated on a very tight budget (in reality, it had no budget when the first meeting was announced). DARPA 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 their own variant. More recently, several systems have demonstrated that a message passing system can be efficiently and be 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 [18], Express [17], nCUBE's Vertex [15], P4 [5], and PARMACS [4, 6]. Other important contributions have come from Zipcode [19, 20], Chimp [11, 12], PVM [3, 9], and PICL [14].

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 [21]. 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 [10]. MPI1 embodies the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. This proposal was intended to initiate discussion of standardization issues within the distributed memory concurrent computing community, and has served as a basis for the subsequent MPI standardization process. Since MPI1 was primarily intended to promote discussion and "get the ball rolling," it focuses mainly on point-to-point communications. MPI1 does not include any collective commu-

---
[1]Version of October 8, 1993

nication routines. MPI1 brought to the forefront a number of important standardization issues, and has served as a catalyst for subsequent progress, however, its major deficiency is that the management of resources is 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 establishing a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follow.

- 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 implementation that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for interface.
- Assume a reliable communication interface: User need not cope with communication failures. Such failures are dealt by the underlying communication subsystem.
- Define an interface that is not too different from current practice, such as PVM, Express, P4, etc and provide extension 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.

1.4 What Is Included In The Standard?

The standard includes:

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

In addition, we have provided a model implementation.

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
- Auxiliary functions such as timers
- Explicit support for threads
- No 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 version of MPI will address some of these issues.

1.6 Organization of this Document

The following is a list of the 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 *recv* 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 usefully be used for other purposes.

- Chapter 9, Initial Implementation Subset, suggests to MPI implementors a "core" subset of MPI that is useful, internally consistent, and should appear first in the evolution of an MPI implementation. The subset is defined so that consistent implementations can appear rapidly, and portable parallel programming with MPI can start in a timely manner.

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

- The MPI Function Index is a simple index showing the location of the language-independent definition of each MPI function.

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 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 a argument marked IN,
- the call may update a argument marked OUT,
- the call both uses and updates a argument marked INOUT.

There is one special case — if an argument is a handle to an opaque object, and the object is updated by the procedure call, then the argument is marked OUT. This is done 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 arguments are, syntactically, an INOUT argument, although, semantically, they are 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.

2.2 Semantic Terms

When discussing point-to-point and collective communications the following terms are used:

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

[1]Version as of October 10, 1993

locally-blocking if return from the procedure indicates the user is allowed to re-use resources specified in the call;

globally-blocking if return from the procedure indicates the communication operation has completed.

When discussing the behavior of any MPI function, not necessarily a communication function, the following terms are used.

local An operation is **local** if its completion depends only on the local executing process. Such an operation does not require communication with another user process.

non-local An operation is **non-local** if its completion may depend on communication occurring with another user process.

collective An operation is **collective** if all processes in a process group need to perform the operation.

2.3 Data Types

2.3.1 Opaque objects

MPI procedures use *opaque objects*. These hide the internal representation used, thus allowing: similar calls in C and Fortran; the avoidance of conflicts with the typing rules in these languages; future extension of functionality. The mechanism for opaque objects used here loosely follows the POSIX Fortran binding standard.

An opaque object can be allocated on user storage or on system (i.e. MPI managed) storage. We call the former a *user object* and the latter a *system object*. User objects are allocated and deallocated explicitly by the user as a regular variable. The type definitions for such objects are provided by MPI. The objects can be accessed and modified only via MPI functions that take such objects as arguments.

System objects are accessed only via *handles*. In C, a different type of handle is defined by MPI for each type of opaque system object. In Fortran, all handles have type `INTEGER`. The allocation and deallocation of such an object, and any access or update to it, are done only via MPI calls that accept as arguments handles to such objects.

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

Opaque system 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. A call to deallocate an opaque system object does not necessarily deallocate the object immediately, but may only mark it as candidate for deallocation. The object will be deallocated as as soon as there there is no pending operation that is using this object; the handle then becomes null. For example, if a communication object is freed after the communication started but before it completed, then the pending operation is not affected; one can use the handle in calls to MPI_WAIT or MPI_STATUS. In fact, one must complete the communication with such a call. However, once the operation completes, it is erroneous to re-use the freed handle for a new communication.

Implementation note: Some objects, such as datatype objects (see section 3.13), can be used by several, distinct, concurrent communication operations. Thus, to find out whether such object can be freed, one needs a reference count of pending operations.

MPI does not keep count of references to an opaque MPI object that are created by assigning to one handle the value of another, or are lost when a handle variable is deallocated (*e.g.*, at procedure exit). In the first case, a call to MPI_GROUP_FREE may leave a dangling reference; in the second case an opaque MPI object may be left with no handle to it, and no possibility of deallocating it. It is the user responsibility to avoid these situations. MPI keeps track only of changes in the number of active references to the MPI object that are a result of explicit MPI calls.

MPI may provide predefined opaque objects and predefined, static handles to these objects. Such objects may not be destroyed.

2.3.2 Array of handles

An MPI call may need an argument that is 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 beggining of the array; len indecates how many of them there are, and need not be the entire size of the array.

2.3.3 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_SET_ERRORMODE routine has a state type argument with values MPI_ERRORSFATAL, MPI_ERRORSRETURN, etc.

2.3.4 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, and sometimes set, using environmental inquiry or environmental management functions (Chapter 7).

Implementation note: Implementors should detect, whenever possible, illegal uses of "special values". Thus, the use of the MPI_ANY_TAG value to tag a message sent should be flagged as an error.

2.3.5 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.

2.4 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 which might be adopted later.

There are several important language binding issues not addressed by this standard. It is beyond the scope of this standard to discuss the interoperability of message passing between languages. Although it would be convenient to be able to guarantee that messages sent between languages be properly interpreted, the matching of types across languages prevents the inclusion of such a guarantee in this standard. It is fully expected that many implementations will have such features, and that such features are a sign of the quality of the implementation.

2.4.1 Data Types and Naming Conventions

Handles are represented as integers in Fortran and as pointers to structures in C. A special value MPI_NULL is used to represent null handles in Fortran; in C, the constant NULL is used as usual.

Typed communication routines support **character**, **double precision**, **real**, and **integer** data in Fortran. **int**, **double**, **real**, **short**, **char**, and **long** are supported in C, along with **unsigned int**, **unsigned long**, **unsigned short**, and **unsigned char**. In addition, a special **byte** type is supported in both languages. This type allows communication to be performed with no type conversion at all. In C, the definitions of named constants, function prototypes, and aggregate type information must be supplied in an include file `mpi.h`. In Fortran, the corresponding include file for named constants (`PARAMETER`s) will be called `mpif.h`.

2.4.2 Fortran 77 Binding Issues

All MPI names have an `MPI_` prefix, and all characters are capitals. 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; error codes are implementation dependent. 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.

```
double precision a
integer b
call MPI_send(a,...)
    .
    .
call MPI_send(b,...)
```

Figure 2.1: This is an example of calling a routine with mismatched formal and actual arguments.

- The same MPI subroutine may be called with different argument types. An example is shown in Figure 2.4.2. This violates the letter of the Fortran standard, but such a violation is common practice. The alternative would be to have a separate version of MPI_SEND, for example, for each data type.
- Although not required, it is strongly suggested that named MPI constants be provided in an include file, called `mpif.h`. On systems that do not support include files, the implementation should spedify the values of named constants.

2.4.3 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.

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 will return values, so that they can be implemented as macros. Logical flags are integers with values 1 and 0 meaning "true" and "false" respectively. Buffer pointers are of type `void*`.

2.5 Processes

An MPI program is executed by several autonomous processes that each execute 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 processor 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 global states.

MPI does not specify either the interaction of processes with signals, in a UNIX environment, or 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.

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, the current proposal does not specify a naming scheme for processes. We propose to always identify processes according to their relative rank in a group, so that, effectively, processes are identified by consecutive integers.

2.6 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 global program failures. Similarly, MPI itself provides no mechanisms for handling node failures. (A mechanism that allows the user to restrict the scope of an unrecoverable error is presented in section 7.2.)

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. The recommended implementation will provide generous limits on some of the resources so as to alleviate the portability problem this represents. A program is **safe** if it is not subject to resource errors, **unsafe**, otherwise.

All MPI calls return an error value that indicates successful completion of the operation, or the error condition that occurred, otherwise. However, MPI does not require the user to check for error values. It is recommended that an implementation provide an environment for MPI execution that can be set so that any error encountered in an MPI call will cause a program to fail with a suitable error message returned to the user. The environment for an MPI execution can also be set so that (non-fatal) errors do not cause a program to fail, and the user can provide his or her own error recovery code. A mechanism that achieves this in a POSIX environment is described in section 7.2.

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.

The behavior of erroneous programs is not defined by this document. The desired behavior is that a relevant error code be returned, if no acceptable behavior can be defined; however, efficiency considerations may dictate that the system does not detect certain types of errors. Also, implementations may go beyond this document in supporting in a meaningful manners codes that are defined here to be erroneous.

MPI, like any other similar message-passing library, allows users to write non-deterministic programs, where race conditions exist and different timing of communication may lead to different results. There is no requirement that repeated executions of the same such program will yield the same result, even when repeated with the same physical configuration — neither user nor MPI can be expected to have full control of the execution environment. However, repeatability is a desirable quality-of-implementation goal. In particular, it is expected that MPI implementations will avoid introducing additional non-determinism. For example, while MPI can take advantage of associativity and commutativity in reordering the execution of a global reduce operation, it is desired (but not required) that the same operations order be used in repeated executions.

2.7 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.

2.7.1 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`:

```
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.

As 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 or recommendations for the output from

```
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).

2.7.2 Interaction with signals in POSIX

The intent of MPI is to be thread and signal safe. This 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`, and `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`.

Chapter 3

Point to Point Communication

3.1 Introduction

Sending and receiving of messages by processes is the basic MPI communication mechanism. All other communication functions can be defined and implemented on top of this basic communication layer, using few auxiliary data management routines, and using the group and communicator manaipulation functions.

The basic point to point communication operations are **send** and **receive**. A **send** operation creates and sends a message. The operation specifies a **send buffer** in the sender memory from which the message data is taken. 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.

A **receive** operation consumes a message. The message to be received is selected according to the value on its envelope, and the message data is put into the **receive buffer**.

The next sections describe the basic (blocking) send and receive operations. We discuss send, receive, basic 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 heterogeneous and noncontiguous data, and conclude with a description of an implementation of MPI point to point communication using a small number of primitives.

3.2 Basic send operation

The syntax of the simplest send operation is given below.

[1]Version of Oct 14, 1993

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
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
```

3.2.1 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. The possible values of this parameter 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_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	

The possible values for this parameter for C and the corresponding C types are listed below.

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short
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	

The datatype MPI_BYTE does not correspond to a Fortran or C datatype. A value of type MPI_BYTE consists of 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.

MPI supports the basic datatypes of Fortran 77 and ANSI C. Additional MPI datatypes should be provided if the host language has additional data types: For example, MPI_LONG_LONG_INT, for 64 bit integers in C; MPI_REAL2, MPI_REAL4 and MPI_REAL8 for 2,4 or 8 byte reals, in Fortran; MPI_INTEGER1 MPI_INTEGER2 and MPI_INTEGER4 for 1, 2 or 4 byte integers in Fortran; etc.

3.2.2 Message envelope

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

source

destination

tag

context

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

The integer-valued message tag is specified by the tag parameter. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is implementation dependent and can be queried using the TagRange environmental inquiry function, as described in Chapter 7.

The context of the message sent is specified by the comm parameter. The message carries the context associated with this communicator (see Chapter 5).

The message destination is specified by the dest parameter as a rank within the process group associated with that same communicator (remote process group, for intercommunicators). The range of valid values is `0, ... , n-1`, where `n` is the number of processes in this group. Thus, point-to-point communications do not use absolute addresses, but only relative ranks within a group. This provides important modularity.

The message envelope would normally be encoded by a fixed-length message header. However, the actual mechanism used to associate an envelope with a message is implementation dependent: some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages; processes may be identified by relative ranks, or absolute ids; etc.

3.3 Basic receive operation

The syntax of the simplest 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 (state)
IN	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object

```
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 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 then or equal to the length of the receive buffer. I.e., all incoming data must fit, without truncation, into the receive buffer. The MPI_PROBE function described in section 3.9 can be used to receive messages of unknown length.

The selection of a message by a receive operation is done uniquely according to the value of the message envelope. The receive operation specifies an **envelope pattern**; a message can be received by that receive operation only if its envelope matches that pattern. A pattern specifies values for the source, tag and context fields of the message envelope. 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 context. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching context, 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** parameter of the receive operation. The message **context** is the context attached with the communicator specified by the parameter **comm**. The message 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** parameter is { `0, ... , n-1` }$\cup${ `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: a process can send a message to itself.

3.3.1 Return status

The source or the tag of a received message may not be known if wildcard values where used in the receive operation. Also, the actual length of the message received may not be known. Thus, this information needs to be returned by the receive operation. The information is returned by the **status** parameter of the MPI_RECV function. This is a parameter of a special MPI-defined type. Status variables need to be explicitly allocated by the user – they are not system objects.

The status variable can be "decoded" to retrieve the **count**, **source** and **tag** fields, using the query functions listed below. The information returned by these query functions is the information last stored in the **status** variable by a receive function. It is erroneous to call these query functions if the **status** variable was never set by a receive.

MPI_GET_SOURCE(status, source)

IN	status	return status of receive operation (Status)
OUT	source	source rank (integer)

```
int MPI_Get_source(MPI_Status status, int *source)
```

```
MPI_GET_SOURCE(STATUS, SOURCE, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), SOURCE, IERROR
```

Returns the rank of the message source in the group associated with the communicator that was used to receive (remote group, for intercommunicators).

MPI_GET_TAG(status, tag)

IN	status	return status of receive operation (Status)
OUT	tag	message tag (integer)

```
int MPI_Get_tag(MPI_Status status, int *tag)
```

```
MPI_GET_TAG(STATUS, TAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), TAG, IERROR
```

Returns the tag of received message.

MPI_GET_COUNT(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype of each receive buffer element
OUT	count	number of received elements (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 elements received. (Here, again, we count *elements*, not *bytes*.) The datatype argument should match the argument provided by the receive call that set the status variable.

Note that it is not mandatory to query the return status after a receive. The receiver will use these calls, or some of them, only when the information they return is needed.

Implementation note:
One expects decode functions to be in-lined in many implementations.

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

3.4 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*, within each context: if two messages are successively sent from the same source, to the same destination, with the same context, then the messages are received in the order they were sent. I.e., if the receiver posts two successive receives that both match either message, then the first receive will receive the first message, and the second receive will receive the second message. This requirement facilitates matching of sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and wildcard MPI_ANY_SOURCE is not used in receives.

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 though 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.

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.

Fairness MPI makes no guarantee of *fairness* in the handling of communication. Suppose that a send was 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 The current practice for many commercial message-passing libraries is that (short) messages are buffered by the system, thus allowing blocking send operations to complete ahead of the matching receives. It is expected that many MPI implementations will follow this practice, and provide the same level of buffering that is available on the libraries they replace. However, message buffering is not a universal practice. Even on systems where buffering occur, the amount of buffer space available and the way it is allocated is bound to be implementation dependent.

Therefore, message buffering is not mandated by MPI and is seen as a quality of implementation issue. A valid MPI implementation of MPI_SEND is to block the sender until a matching receive has been initiated. In general, the programmer can make no assumptions on the availability of buffer space, and how this space is allocated. Thus, portable (safe) MPI code should work under the assumption that an arbitrary subset of the send operations are going to return before a matching receive is posted, and the rest will block until a matching receive is posted.

MPI implementation will provide information on the amount of available buffer space and on the buffering policy via the environmental enquiries described in Chapter 7.

Examples (involving two processes with ranks 0 and 1)

The following program is safe, and should always succeed.

```
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)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

The following program is erroneous, and should always fail.

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

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

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second processor is executed; the receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will deadlock.

The following program is unsafe, and may succeed or fail, depending on implementation.

```
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_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

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

3.5 Data Type Matching

One can think of message transmission as consisting of 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.

A type specified for an entry by a send operation matches the type specified for that entry by a receive operation if both operations use identical names: MPI_INTEGER matches MPI_INTEGER, MPI_REAL matches MPI_REAL, and so on.

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: an entry with type name MPI_INTEGER matches a Fortran variable of type `INTEGER`, an entry with type name MPI_REAL matches a Fortran variable of type `REAL`, and so on. There is one exception to this last rule: An entry with type name MPI_BYTE 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 value of the message entry will be the binary value of the corresponding byte in memory.

We thus have two cases:

- 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 should 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.

The following examples illustrate these two cases.

First program:

```
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(a(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

This code is correct if both sender and receiver programs have allocated consecutive storage for ten real numbers, starting from `a(1)`.

Second program:

```
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(a(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

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

Third program:

```
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(a(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is correct, irrespective of the type of a(1) and following variables in store.

Some implementations may disallow the transfer of Fortran character strings using the MPI type MPI_BYTE.

3.6 Data conversion

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

We use the following terminology:

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

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

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical or character values, and to convert a floating point value to the nearest value that can 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 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 no conversion ever occurs when an MPI program executes in a homogeneous system, where all processes run in the same environment. Also note the different behavior of MPI_BYTE and of MPI_CHAR. A buffer descriptor entry with datatype of MPI_CHAR can only match a C variable of type CHAR; and representation conversion may occur when values of type MPI_CHAR are transferred., e.g., from an EBCDIC encoding to an ASCII encoding.

Consider the previous three examples.

The first program is correct, assuming both sender and receiver declared ten consecutive real variables in storage starting at a(1). 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: e.g. the sender may use a four byte representation and the receiver an eight byte representation for reals. If the send or receive buffer do not contain ten consecutive real variables, then the program is erroneous, and its behavior is undefined.

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 the variables in the send buffer are of different types as the variables in the receive buffer, or 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.

The current draft does not provide a mechanism for inter-language communication: messages sent by Fortran calls should be received by Fortran calls and messages sent by C calls should be received by C calls (this follows from the requirements of type matching and the fact that Fortran and C datatypes are distinct). If inter-language communication is needed then one needs to invoke C communication routines from a Fortran program or Fortran communication routines from a C program.

Implementation note:
The current definition does not require messages to carry data type information. A message can be composed and sent using only the information provided by the send call, and can be received and stored using only the information provided by the receive call. If messages are sent between different machines then one can either use a "universal" data encoding for messages, use knowledge of the receiver environment in order to convert data at the sender, or use knowledge of the sender environment in order to convert data at the receiver. In either case the datatype parameter in the local call can be used to derive the types of the values transferred.

Additional type information can be used to provide better error detection.

3.7 Communication Modes

The basic send operation described in section 3.2 used the **standard** communication mode. In such a communication mode, a send operation can be started whether or not a matching receive was posted. The completion of the send operation indicates that the message and its envelope have been safely stored away and that the sender is free to access and modify the sender buffer. Thus, the operation is **locally-blocking**: it does not return until the send operation has completed **locally**, on the sender side.

The completion of a send operation gives no indication that the message was received on the receiver side. A blocking send may be implemented so that it returns only after a matching receive has been executed on the receiver side. This avoids the need to buffer message data out of sender or receiver memory. In this case the send operation completes only after the matching receive has started executing. On the other hand, it is also possible for MPI to buffer messages, so as to allow the sender to proceed ahead of the receiver. In such a case the send operation may complete successfully before the message was received.

There are two additional communication modes:

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. In 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; it is merely that it can be used only in part of the situations where a regular sent can be used.

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. (I.e., the receive has been posted, and the incoming message has been matched to this posted receive.) 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 ends before both processes "attend" to the communication; the completion of a synchronous send is a **global** event, and the operation is **globally-blocking**.

Two additional send functions are provided for the two additional communication modes. The communication mode is indicated by a one letter prefix: R for ready and S for synchronous.

Send in ready mode

MPI_RSEND (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
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

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

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

Send in synchronous mode

MPI_SSEND (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
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

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

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

There is only one receive mode, 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. It is **asynchronous**: the completion of a receive is a local operation, and a receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

Communication imposes an order on the events occurring at the communicating nodes. It is always the case that the completion of a receive occurs after the start of the matching send. If the synchronous send mode is used, then it is also the case that the completion of a send occurs after the start of the matching receive. Of course, on each process, a communication completes after it is started. No other order is imposed by MPI. E.g., if the standard send mode is used, the send operation may complete before the matching receive was started.

In a multi-threaded implementation of MPI, the system may deschedule 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 the user responsibility not to access or modify a communication buffer until the communication completes; otherwise the outcome of the computation is undefined.

Implementation note: A ready send can be implemented as a standard send; in such a 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.

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.

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

3.8 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 call initiates the send operation, but does not complete it. The send call will return before the message was copied out of the send buffer. A separate 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 initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A separate 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 sends can use the same three modes as blocking sends: `standard`, `ready` and `synchronous`. These carry the same meaning. The initiation and completion of a standard send do not depend on the status of a matching receive. A ready send can be initiated only if a matching receive has already been initiated, otherwise the call is erroneous; its completion does not depend on the status of a matching receive. A synchronous send can be initiated before a matching receive has been initiated, but will complete only after a matching receive has been initiated, and has started receiving the message generated by the send operation.

In all cases the operation that initiates the communication (send or receive) is a local, nonblocking call. Such call may fail because because the system has exhausted available resources (e.g., exceeded the limit on number of pending communications per node). Good quality implementations of MPI will set these limits high enough so that a nonblocking communication initiation call will fail only in “pathological” cases.

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

3.8.1 Communication Objects

Nonblocking communications use opaque communication 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. An opaque communication object identifies various properties of a communication operation, such as the (send or receive) buffer that is associated with it, its context, the tag and destination parameters to be used for a send, or the tag and source parameters to be used for a receive. In addition, this object stores information about the status of the pending communication operation that is performed with this object.

3.8.2 Communication initiation

We use the same naming conventions as for blocking communication: a prefix of `R` (`S`) is used for `READY` (`SYNCHRONOUS`) mode. In addition a prefix of `I` (for `IMMEDIATE`) indicates that the call is nonblocking.

Initiate a standard mode nonblocking communication.

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
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication object (handle)

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

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

Initiate a ready mode nonblocking communication.

MPI_IRSEND(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
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication object (handle)

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

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

Initiate a synchronous mode nonblocking communication.

MPI_ISSEND(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
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication object (handle)

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

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

Initiate a nonblocking receive.

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

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
IN	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication object (handle)

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Comm_request *request)
```

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

These calls allocate a communication object and associates it with the request. The request can be used later to query about the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may buf copying data out of the send buffer. The sender should not access any part of the send buffer after a nonblocking send operation returns, until the send completes.

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

3.8.3 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, or any other location that can be referenced by the send operation (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, and that the status object is set; the receiver is now free to access the receive buffer, or any other location that can be referenced by the receive operation. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated).

MPI_WAIT(request, status)

IN	request	handle to communication object
OUT	status	status object

```
int MPI_Wait(MPI_Comm_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 send 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 becomes null.

The call returns in status information on the completed operation. The status object for a receive operation can be queried using the functions described in section 3.3.1. The status parameter is not used or updated for a send operation, and an arbitrary value may be passed for this parameter in the call.

MPI_TEST(request, flag, status)

IN	request	handle to communication object
OUT	flag	logical
OUT	status	status object

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

```
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, FLAG, 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 becomes null. The call returns flag=false, otherwise. In such a case, the value of the status object is undefined.

MPI_TEST is a local, nonblocking operation.

The return status object for a receive operation carries information that can be accessed using the functions described in section 3.3.1. The status argument is not used or updated for a send operation and an arbitrary value can be passed for this parameter in the call.

In multi-threaded environment, the use of a blocking receive operation (MPI_WAIT) may allow the operating system to de-schedule the blocked thread and schedule another thread for execution, if such is available. The use of a nonblocking receive operation (MPI_TEST) allows the user to schedule alternative activities within a single thread of execution.

Example:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0)
  THEN
    CALL MPI_ISEND(request, a(1), 10, MPI_REAL, 1, tag, comm, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
  ELSE
    CALL MPI_IRECV(request, a(1), 15, MPI_REAL, 0, tag, comm, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
END IF
```

The functions MPI_WAIT and MPI_TEST can be used to complete both sends and receives; they will be also used to complete any other nonblocking communication call provided by MPI.

3.8.4 Semantics of Nonblocking Communications

The semantics of nonblocking communication is defined by suitably extending the definitions in section 3.4 for order and progress.

Nonblocking communication operations are ordered according to the execution order of the calls that initiate the communication; this defines the meaning of non-overtaking communication.

Example:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0)
   THEN
      CALL MPI_ISEND(h1, a, 1, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_ISEND(h2, b, 1, MPI_REAL, 1, 0, comm, ierr)
   ELSE    ! rank.EQ.1
      CALL MPI_IRECV( h1, a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, ierr)
      CALL MPI_IRECV( h2, b, 1, MPI_REAL, 0, 0, comm, ierr)
END IF
CALL MPI_WAIT(h1,status)
CALL MPI_WAIT(h2,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 receives. This precludes an implementation where different queues are used to post the two receives.

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:

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

This code should not deadlock in a correct MPI implementation.

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

Nonblocking communications cannot be replaced by blocking communications, even in the synchronous communication mode. Consider the following example:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0)
  THEN
    CALL MPI_ISEND(request1, a(1), count, MPI_REAL, 1, tag1, comm, ierr)
    CALL MPI_ISEND(request2, a(2), count, MPI_REAL, 1, tag2, comm, ierr)
    CALL MPI_WAIT(request1, status, ierr)
    CALL MPI_WAIT(request2, status, ierr)
  ELSE
    CALL MPI_IRECV(request2, a(2), count, MPI_REAL, 0, tag2, comm, ierr)
    CALL MPI_IRECV(request1, a(1), count, MPI_REAL, 0, tag1, comm, ierr)
    CALL MPI_WAIT(request2, status, ierr)
    CALL MPI_WAIT(request1, status, ierr)
END IF
```

The code is guaranteed to execute correctly, even in an implementation that does not buffer messages (the only resource requirement is the ability to have two pending communications). If one replaces the nonblocking sends (receives) by blocking sends (receives), and deletes the calls to MPI_WAIT, then the code will deadlock. The code will also deadlock if one deletes the calls that initiate communication and replaces each call to MPI_WAIT by the appropriate blocking send or receive.

3.8.5 Multiple Completions

It is convenient to be able to wait for the completion of any or all the operations in a set, 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.

MPI_WAITANY (count, array_of_requests, index, status)

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

```
int MPI_Waitany(int count, MPI_Comm_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 communication requests in the array has completed. Returns the index of that request in the array, and returns the status of that operation in the object associated with the 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. In particular, the associated communication object is is deallocated, and the handle to it in array_of_requests is set to null. If more then one operation is enabled and can terminate, one is arbitrarily chosen. There is no requirement that the choice satisfy any fairness criterion.

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

IN	count	list length (integer)
INOUT	array_of_requests	array of handles to communication objects
OUT	index	index of handle for operation that completed, or MPI_UNDEFINED if none completed (integer)
OUT	flag	true if one of the operations is complete
OUT	status	status object

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

```
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
```

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

Causes either one or none of the operations associated with the communication requests to return. In the former case, it returns in `index` the index of its request in the array, returns in `status` the status of that operation, deallocates the associated communication object, and sets the request to null. In the latter case, it returns a value of MPI_UNDEFINED in index and status is undefined.

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`; in the later case, it is set to MPI_UNDEFINED.

MPI_WAITALL(count, array_of_requests, array_of_statuses)

IN	count	lists length (integer)
IN	array_of_requests	array of handles to communication objects
OUT	array_of_statuses	array of status objects

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

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

Blocks until all communication operations associated with requests in the list complete, and return the status of all these operations. 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. All communication objects are deallocated and all requests are set to null.

The 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_TESTALL(count, array_of_requests, flag, array_of_statuses)

IN	count	lists length (integer)
INOUT	array_of_requests	array of handles to communication objects
OUT	flag	(logical)
OUT	array_of_statuses	array of status objects

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

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

Causes either all or none of the operations associated with the communication requests to complete. It returns flag = true if all communications associated with requests in the array have completed. In this case, each status entry is set to the status of the corresponding communication. All communication objects are deallocated, and all requests are set to null. Otherwise, flag = false is returned, no request is modified and the values of the status entries are undefined. This is a nonblocking operation that returns immediately.

Example:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank < 2)
  THEN         ! rank=0 or rank=1
    CALL MPI_ISEND(request, a, n, MPI_REAL, 2, tag, comm, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
  ELSE         ! rank=2
    CALL MPI_IRECV(request_list(0), a, n, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_IRECV(request_list(1), b, n, MPI_REAL, 1, tag, comm, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAITANY(2, request_list, index, status, ierr)
    IF(index.EQ.0)
      THEN
        **** handle message from process 0 ****
        CALL MPI_WAIT(request_list(1), status, ierr)
        **** handle message from process 1 ****
      ELSE
        **** handle message from process 1 ****
        CALL MPI_WAIT(request_list(0), status, ierr)
        **** handle message from process 0 ****
    END IF
END IF
```

The calls introduced in this subsection can be used to wait or test for the completion of an arbitrary mix of nonblocking communication calls, including a mix sends and receives and of any additional nonblocking communication calls provided by MPI.

The status argument in MPI_WAITANY and MPI_WAITALL and the array_of_statuses argument in MPI_WAITALL and MPI_TESTALL are not used or updated if the completing communication is a send. If all operations in the list provided by the argument array_of_requests are send operations, then an arbitrary value can be passed to the status (resp. array_of_statuses) argument.

An array_of_requests argument that is passed to any of the last four functions may contain null entries. These entries are ignored and the call executes as if it was supplied an array of requests containing only the non-null entries. As a request in array_of_requests is set to null when the corresponding operation completes, this allows to reuse the same array for completing in successive calls the operations associated with this array. Thus, the last example can be rewritten as follows:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank < 2)
  THEN         ! rank=0 or rank=1
```

```
    CALL MPI_ISEND(request, a, n, MPI_REAL, 2, tag, comm, ierr)
    CALL MPI_WAIT(request, status, ierr)
  ELSE        ! rank=2
    CALL MPI_IRECV(request_list(0), a, n, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_IRECV(request_list(1), b, n, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_WAITANY(2, request_list, index, status, ierr)
        **** handle first message  ****
    CALL MPI_WAITANY(2, request_list, index, status, ierr)
        **** handle second message  ****
```

If MPI_WAITANY or MPI_TESTANY are called with an empty array_of_requests argument (i.e., an array of length zero, or an array with all entries null) then they return the value MPI_UNDEFINED in index; MPI_WAITALL with an empty array_of_requests argument has no effect; MPI_TESTALL with an empty array_of_requests argument returns flag=true.

3.9 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 cancelled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully.

MPI_IPROBE(source, tag, comm, flag, status)

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

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

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

MPI_IPROBE returns flag = true if there is a message that can be received and that matches the pattern specified by the parameters source, tag, and comm. It returns flag = false, otherwise. If MPI_IPROBE returns flag = true, then the status object can be subsequently queried using the functions described in section 3.3.1 to find the source, tag and length of the message. The values returned by these queries are the same values that would have been returned by a call to MPI_RECV executed at the same point in the program. The return status object is undefined if flag = false.

A subsequent receive executed with the same context, and the source and tag returned by querying the status returned by MPI_IPROBE will receive the message that was matched by the probe, if no other intervening receive occurred after the probe. If the receiving process is multi-threaded, it is the user responsibility to ensure that the last condition holds.

MPI_PROBE(source, tag, comm, status)

IN	source	source rank, or MPI_ANY_SOURCE (integer)
IN	tag	tag value, or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object

```
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 which returns only after a matching message has been found.

MPI_CANCEL(request)

IN	request	communication object (handle)

```
int MPI_Cancel(MPI_Comm_request request)
```

```
MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

A call to MPI_CANCEL marks for cancellation a pending nonblocking communication operation (send or receive). The cancel call is nonblocking, and local. It returns immediately, possibly before the communication is actually cancelled.

It is still necessary to complete a communication that has been marked for cancellation, using a call to MPI_WAIT or MPI_TEST (or any of the derived operations). If the operation has been cancelled, then information to that effect will be returned in the status argument of the operation that completes the communication. If a communication is marked for cancellation, then a MPI_WAIT call for that communication is guaranteed to return, irrespective of the activities of other processes (i.e., MPI_WAIT behaves as a local function); similarly if MPI_TEST is repeatedly called in a busy wait loop for a cancelled communication, then MPI_TEST will eventually be successful.

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

MPI_TEST_CANCELLED(status, flag)

IN	status	status object
IN	flag	(logical)

```
int MPI_Test_cancelled(MPI_Status status, int *flag)
```

```
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), FLAG, IERROR
```

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

3.10 Persistent communication objects

Often a communication with the same parameter 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 parameters to the communication object once and, then, repeatedly using the communication object to initiate and complete messages. The communication object 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 processor and communication controller, but not the overhead for communication between one communication controller and another.

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

The function MPI_CREATE_SEND creates a communication object for a standard mode send operation, and binds to it all the parameters of a send operation.

MPI_CREATE_SEND(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
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication object (handle)

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

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

The call allocates a new communication object and associates the communication request with it.

The function MPI_CREATE_RSEND creates a communication object for a ready mode send operation.

MPI_CREATE_RSEND(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
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication object (handle)

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

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

The function MPI_CREATE_SSEND creates a communication object for a synchronous mode send operation.

MPI_CREATE_SSEND(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
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication object (handle)

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

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

The function MPI_CREATE_RECV creates a communication object for a receive operation.

MPI_CREATE_RECV(buf, count, datatype, source, tag, comm, request)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements received (integer)
IN	datatype	type of each element
IN	dest	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	request	communication object (handle)

```
int MPI_Create_recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Comm_request *request)
```

```
MPI_CREATE_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    INTEGER BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

A communication (send or receive) that uses a predefined communication object is initiated by the function MPI_START.

MPI_START(request)

INOUT	request	communication object (handle)

```
int MPI_Start(MPI_Comm_request request)
```

```
MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

The communication object associated with request should be an object that was created by one of the previous four functions, so that all the communication parameters are already defined. A send can be started provided that the previous send using the same object has completed, or as soon as the object is created, if it has not yet been used in a communication. In addition, if the communication mode is `ready` then a matching receive should be posted. The send buffer should not be updated after the send is started, until the operation completes.

A receive can be started provided that the preceding receive using the same object has completed, or as soon as the object is created, if it has not yet been used in a communication. The receive buffer should not be accessed after the receive is started, until the operation completes.

The call is nonblocking, with similar semantics as the nonblocking communication operations described in section 3.8.

MPI_STARTALL(count, array_of_requestss)

IN	count	list length (integer)
INOUT	array_of_requests	array of communication objects

```
int MPI_Startall(int count, MPI_Comm_request *array_of_requests)
```

```
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
```

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

Start all communications associated with requests in array_of_requests. A call to MPI_STARTALL(count, array_of_requests) has the same effect as calls to MPI_START(array_of_requests[i]), executed for `i=0 ,..., count-1`, in some arbitrary order.

A communication started with a call to MPI_START or MPI_STARTALL is completed by a call to MPI_WAIT, MPI_TEST, or one of the derived functions described in section 3.8.5. These communication completion functions do not deallocate the communication object, and this can be reused anew by a MPI_START call. The object needs to be explicitly deallocated by a call to the function MPI_COMMOBJ_FREE, below. We thus have two types of communication objects: **persistent** objects, which are allocated by a call to MPI_CREATE_xxx, and are explicitly deallocated by MPI_COMMOBJ_FREE, and **ephemeral** objects that persist for one communication only: they are created by a nonblocking communication initiation function, and are freed by the communication completion call.

MPI_COMMOBJ_FREE(request)

INOUT	request	communication object (handle)

```
int MPI_Commreq_free(MPI_Comm_request *request)
```

```
MPI_COMMOBJ_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

deallocated when there are no pending communications involving this object, at which point the request becomes null.

The call is nonblocking. It is permissible to call MPI_COMMOBJ_FREE(request) after a communication that uses request has been initiated, but before it has completed. The object will be deallocated after the communication completes. It is erroneous to initiate a communication with a request that has been freed.

A correct invocation of the functions described in this section will occur in a sequence of the form

Create (**Start Complete**)* **Free** ,

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

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

3.11 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 destination, 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 shift, then one needs to order correctly the sends and receives (e.g., even processes send, next receive, odd processes receive first, next send) so as to prevent cyclic dependencies that 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, and vice versa.

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
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
IN	source	rank of source (integer)
IN	recvtag	receive tag (integer)
IN	comm	communicator (handle)
OUT	status	status object

```
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, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

Execute a blocking send and receive operation. Both send and receive use the same tag value and the same communicator. However, the send buffer and receive buffer are disjoint, and may have different length and different 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
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

```
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 obtain 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. Additional intermediate buffering is needed for the "replace" variant.

A send-receive cannot be implemented by a blocking send followed by a blocking receive or a blocking receive, followed by a blocking send. Consider the following code, where two processes exchange messages:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0)
  THEN
    CALL MPI_SENDRECV(send_buff, send_count, MPI_REAL, 0, send_tag,
       recv_buff, recv_count, MPI_REAL, 1, recv_tag, comm, status, ierr)
  ELSE
    CALL MPI_SENDRECV(send_buff, send_count, MPI_REAL, 1, send_tag,
       recv_buff, recv_count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

If the send receives are replaced either by blocking send, followed by blocking receive, or blocking receive, followed by blocking send, then the code may deadlock. On the other hand, send-receive can be implemented using nonblocking sends and receives.

3.12 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, e.g., in the case of a non-circular shift done with calls to send-receive.

The special value MPI_PROCNULL can be used instead of a rank wherever a source or a destination parameter is required in a call. A communication with process MPI_PROCNULL has no effect: a send to MPI_PROCNULL succeeds and returns as soon as possible. A receive from MPI_PROCNULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with source = MPI_PROCNULL is executed then the status object returns source = MPI_PROCNULL, tag = MPI_ANY_TAG and count=0.

3.13 Derived datatypes

Up to now, all point to point communication 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 provide functions that 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 beeing transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allows 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 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 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

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

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

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

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 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, in replacement 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 address later in section 3.13.5 the case where the second argument count has value > 1.

The basic datatypes presented in section 3.2.1 are particular cases of a general datatype, and are predefined. Thus, MPI_INT is a predefined handle to a datatype with type map $\{(\mathtt{int}, 0)\}$, with one entry of type int and displacement zero. And similarly for all other basic datatypes.

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

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

then

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

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

Example: Assume that $Type = \{(\mathtt{double}, 0), (\mathtt{char}, 8)\}$ (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 multiple 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.

3.13.1 Datatype constructors

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

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

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

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

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

Example: let oldtype have type map $\{(\texttt{double}, 0), (\texttt{char}, 8)\}$, with extent 16, and let count $= 3$. The type map of the datatype returned by newtype is

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

i.e., alternating double and char elements, with displacements $0, 8, 16, 24, 32, 40$.

In general, assume that the type map of oldtype is

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

with extent *extent*. Then newtype has a type map with $\mathsf{count} \cdot \mathsf{n}$ entries defined by:

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

$$..., (type_0, disp_0 + extent \cdot (\mathsf{count} - 1)), ..., (type_{n-1}, disp_{n-1} + extent \cdot (\mathsf{count} - 1))\}.$$

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.

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

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

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

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

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

$$\{(\mathtt{double}, 0), (\mathtt{char}, 8), (\mathtt{double}, 16), (\mathtt{char}, 24), (\mathtt{double}, 32), (\mathtt{char}, 40),$$
$$(\mathtt{double}, 64), (\mathtt{char}, 72), (\mathtt{double}, 80), (\mathtt{char}, 88), (\mathtt{double}, 96), (\mathtt{char}, 104)\} :$$

two blocks with three copies each of the old type, starting 4*16 apart.

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

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

In general, assume that oldtype has type map

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

with extent *extent*. The newly created datatype has a type map with $\mathtt{count} \cdot \mathtt{blocklength} \cdot n$ entries:

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$
$$(type_0, disp_0 + extent), ..., (type_{n-1}, disp_{n-1} + extent), ...,$$
$$(type_0, disp_0 + (\mathtt{blocklength} - 1) \cdot extent), ..., (type_{n-1}, disp_{n-1} + (\mathtt{blocklength} - 1) \cdot extent),$$
$$(type_0, disp_0 + \mathtt{stride} \cdot extent), ..., (type_{n-1}, disp_{n-1} + \mathtt{stride} \cdot extent), ...,$$
$$(type_0, disp_0 + (\mathtt{stride} + \mathtt{blocklength} - 1) \cdot extent), ...,$$
$$(type_{n-1}, disp_{n-1} + (\mathtt{stride} + \mathtt{blocklength} - 1) \cdot extent), ...,$$
$$(type_0, disp_0 + \mathtt{stride} \cdot (\mathtt{count} - 1) \cdot extent), ...,$$
$$(type_{n-1}, disp_{n-1} + \mathtt{stride} \cdot (\mathtt{count} - 1) \cdot extent), ...,$$
$$(type_0, disp_0 + (\mathtt{stride} \cdot (\mathtt{count} - 1) + \mathtt{blocklength} - 1) \cdot extent), ...,$$
$$(type_{n-1}, disp_{n-1} + (\mathtt{stride} \cdot (\mathtt{count} - 1) + \mathtt{blocklength} - 1) \cdot extent)\}$$

A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, oldtype, 1, 1, newtype), or to a call to MPI_TYPE_VECTOR(1, oldtype, n, count, newtype), `n` arbitrary.

Hvector The function MPI_TYPE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather then in elements. The use for both types of vector constructors is illustrated in section 3.13.6. (`H` is used for "heterogeneous").

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

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

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

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

Assume that oldtype has type map

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

with extent *extent*. The newly created datatype has a type map with $\texttt{count}\cdot\texttt{blocklength}\cdot n$ entries:

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$
$$(type_0, disp_0 + extent), ..., (type_{n-1}, disp_{n-1} + extent), ...,$$
$$(type_0, disp_0 + (\texttt{blocklength} - 1)\cdot extent), ..., (type_{n-1}, disp_{n-1} + (\texttt{blocklength} - 1)\cdot extent),$$
$$(type_0, disp_0 + \texttt{stride}), ..., (type_{n-1}, disp_{n-1} + \texttt{stride}), ...,$$
$$(type_0, disp_0 + \texttt{stride} + (\texttt{blocklength} - 1) \cdot extent), ...,$$
$$(type_{n-1}, disp_{n-1} + \texttt{stride} + (\texttt{blocklength} - 1) \cdot extent), ...,$$
$$(type_0, disp_0 + \texttt{stride} \cdot (\texttt{count} - 1)), ...,$$
$$(type_{n-1}, disp_{n-1} + \texttt{stride} \cdot (\texttt{count} - 1)), ...,$$
$$(type_0, disp_0 + \texttt{stride} \cdot (\texttt{count} - 1) + (\texttt{blocklength} - 1) \cdot extent), ...,$$
$$(type_{n-1}, disp_{n-1} + \texttt{stride} \cdot (\texttt{count} - 1) + (\texttt{blocklength} - 1) \cdot extent)\}$$

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

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

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

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

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

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

$$\{(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104)$$
$$(\texttt{double}, 0), (\texttt{char}, 8)\} :$$

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

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

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

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

A call to MPI_TYPE_VECTOR(count, oldtype, stride, blocklength, newtype) is equivalent to a call to MPI_TYPE_INDEX(count, oldtype, D, B, newtype) where

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

and

$$\mathsf{B[j]} = \mathsf{blocklength}\ , j = 0, ..., \mathsf{count} - 1.$$

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.

MPI_TYPE_HINDEXED(count, oldtype, array_of_displacements, array_of_blocklengths, newtype)

IN	count	number of blocks – also number of entries in array_of_displacements and array_of_blocklengths (integer)
IN	oldtype	old datatype (handle)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	array_of_blocklengths	number of elements in each block (array of nonnegative integers)
OUT	newtype	new datatype (handle)

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

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

Assume that oldtype has type map

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

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

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

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

$$(type_0, disp_0 + \mathsf{D[count - 1]}), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[count - 1]}), ...,$$

$$(type_0, disp_0 + \mathsf{D[count - 1]} + (\mathsf{B[count - 1]} - 1) \cdot extent), ...,$$

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

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

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

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

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

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

Example: Let type have type map

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

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

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

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 parameter, where T[i] is a handle to

$$type_i = \{(type_0^i, disp_0^i), ..., (type_{n_i}^i, disp_{n_i}^i)\},$$

with extent $extent_i$. Let B be the array_of_blocklength parameter and D be the array_of_indices parameter. Then the newly created datatype has a type map with $\sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]} \cdot n_i$ entries

$$\{(type_0^0, disp_0^0 + \mathsf{D[0]}), ..., (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D[0]}), ...,$$

$$(type_0^0, disp_0^0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot extent_0), ..., (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot extent_0), ...,$$

$$(type_0^{\mathsf{count}-1}, disp_0^{\mathsf{count}-1} + \mathsf{D[count}-1]), ..., (type_{n_{\mathsf{count}-1}}^{\mathsf{count}-1}, disp_{n_{\mathsf{count}-1}}^{\mathsf{count}-1} + \mathsf{D[count}-1]), ...,$$

$$(type_0^{\mathsf{count}-1}, disp_0^{\mathsf{count}-1} + \mathsf{D[count}-1] + (\mathsf{B[count}-1] - 1) \cdot extent_{\mathsf{count}-1}), ...,$$

$$(type_{n_{\mathsf{count}-1}}^{\mathsf{count}-1}, disp_{n_{\mathsf{count}-1}}^{\mathsf{count}-1} + \mathsf{D[count}-1] + (\mathsf{B[count}-1] - 1) \cdot extent_{\mathsf{count}-1})\}$$

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

3.13.2 Address and extent functions

The displacements in a general datatype are relative to some buf 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 parameter 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_ADDR address)
```

```
MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR
```

Returns the (byte) address of location.

Example:

```
   REAL A(100,100)
   INTEGER I1, I2, DIFF
   CALL MPI_ADDRESS(A(1,1), I1)
   CALL MPI_ADDRESS(A(10,10), I2)
   DIFF = I2 - I1
! The value of DIFF is 1009*sizeofreal; the values of I1 and I2 are
! implementation dependent.
```

Some implementations may not support calls to MPI_ADDRESS in Fortran, where the location argument is of type CHARACTER.

The following auxiliary functions provide useful information on derived datatypes.

MPI_TYPE_EXTENT(datatype, extent)

IN	datatype	datatype (handle)
OUT	extent	datatype extent (integer)

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

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

Returns the extent of a datatype – where extent is as defined in Eq. 3.1.

MPI_TYPE_SIZE(datatype, size)

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

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

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

Returns the number of bytes occupied by entries in the datatype – i.e. the sum of the sizes of the datatype elements.

MPI_TYPE_COUNT(datatype, count)

IN	datatype	datatype (handle)
OUT	count	datatype count (integer)

```
int MPI_Type_count(MPI_Datatype datatype, int *count)
```

```
MPI_TYPE_COUNT(DATATYPE, COUNT, IERROR)
    INTEGER DATATYPE, COUNT, IERROR
```

Returns the number of entries in the datatype.

3.13.3 Lower-bound and upper-bound markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, and overwrite the definition given by Equation 3.1. This allows 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 a usage are provided in section 3.13.6. 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 ($extent(\mathsf{MPI\_LB}) = extent(\mathsf{MPI\_UB}) = 0$). They do not affect the size or count of a datatype, and do not affect the 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: 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, T, D, B, type1) creates a new datatype that has an extent of 9 (from -3 to 5), 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)`} . (Entries of type `lb` or `ub` can be deleted if they are not at the endpoints of the datatype.)

In general, if

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

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

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

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

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

Then

$$extent(Typemap) = ub(Typemap) - lb(Typemap) + \epsilon$$

If $type_i$ requires alignment to a byte address that is 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**.

3.13.4 Commit and free

A datatype object has to be **committed** before it can be used in a communication. 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. A commited datatype can still be used as a parameter in datatype constructors.

Note: the commit operation commits the datatype, i.e., 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.

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
```

There is no need to commit basic datatypes; they are "pre-committed".

MPI_TYPE_FREE(datatype)

INOUT	datatype	handle to datatype object

```
int MPI_Type_free(MPI_Datatype datatype)
```

```
MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

Marks the datatype object associated with datatype for deallocation. The object will be deallocated after any pending communication that uses this object completes, at which point datatype becomes null. It is erroneous to use a datatype as argument to an MPI function after it was freed.

Example:

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
              ! new type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! now type1 can be used for communication
type2 = type1
              ! type2 can be used for communication
              ! (it is a handle to same object as type1)
CALL MPI_TYPE_VECTOR(3, MPI_REAL, 4, 5, type1, ierr)
              ! new uncommitted type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! now type1 can be used anew for communication
```

Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system should behave as if a separate copy of the datatype was created whenever this datatype was incorporated in a new derived datatype.

Implementation note:
The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather then copying them. In such case, one

needs to keep track of active datatype definition references in order to know when a datatype object can be freed.

3.13.5 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. The same holds true for the other communication functions that have a datatype argument. Thus,

MPI_SEND(buf, count, datatype, dest, tag, comm) is equivalent to

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

The same applies to all other communication functions that have a count and datatype argument.

Let datatype have type map

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

with extent *extent* (empty entries deleted). Then the send buffer used by the send operation consists of $n \times \texttt{count}$ entries, where entry $i \times n + j$ is at location

$$addr_{i,j} = \texttt{buf} + extent \times i + disp_j$$

and has type $type_j$, for $i = 0, ..., \texttt{count} - 1$ and $j = 0, ..., n - 1$. 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 3.5. The message sent contains $n \times \texttt{count}$ entries, where entry $i \times n + j$ 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

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

with extent *extent* (empty entries deleted). Then the receive buffer used for this receive operation consists of the $n \times \texttt{count}$ entries, where entry $i \times n + j$ is at location $\texttt{buf} + extent \times i + disp_j$ and has type $type_j$. If the incoming message consists of k elements, then we must have $k \leq n \times \texttt{count}$; the $i \times n + j$-th element of the message should have a type that matches $type_j$.

Note that type matching is defined according to the type signature of the corresponding datatypes, i.e., the sequence of basic type components, and does not depend on the datatype definition.

Example:

```
...
CALL MPI_TYPE_CONTIGUOUS( MPI_REAL, 2, type2, ...)
CALL MPI_TYPE_CONTIGUOUS( MPI_REAL, 4, type4, ...)
CALL MPI_TYPE_CONTIGUOUS( type2, 2, 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 each of the receives.

A datatype used in a receive operation may not have overlaping entries in its type map: the same location in the receive buffer cannot be written twice during a receive.

Type matching requirements imply that (nonempty) entries in the type map of a datatype cannot overlap, unless they coincide and have the same basic type, or one of them have basic type byte: otherwise, when the datatype is used in a communication, at least on of the two overlaping entries will not be type-consistent with the communication buffer.

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

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

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 $0 \leq k \leq \texttt{count} \times n$.

The number of 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
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 used function MPI_GET_COUNT has a different behavior: it returns the number of "top elements" received. In the previous example, MPI_GET_COUNT may return any integer value k, where $0 \leq k \leq \texttt{count}$. If MPI_GET_COUNT returns k, then the number of basic elements received (and the value returned by MPI_GET_ELEMENTS) is $n \times k$. If the number of basic elements received is not a multiple of n, i.e. if the receive operation has not received an integral number of datatype "copies", then MPI_GET_COUNT returns the value MPI_UNDEFINED.

Example:

```
...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_COMMIT(Type2, ierr)
...
CALL MPI_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, i, ierr)       ! returns i=1
      CALL MPI_GET_ELEMENTS(stat, i, ierr)    ! returns i=2
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, i, ierr)       ! returns i=MPI_UNDEFINED
      CALL MPI_GET_ELEMENTS(stat, 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.

Implementation note: The current definition implies that a receive cannot change the value of storage between 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 structure is copied from one process to another. This would prevents 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.

3.13.6 Examples

The following examples illustrate the use of derived datatypes.

First example: 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
      MPI_STATUS status

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

```
      CALL MPI_COMM_RANK(MPI_ALL, myrank)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

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

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

C     create datatype for the entire section
      CALL MPI_TYPE_HVECTOR( 9, twoslice,  100*100*sizeofreal, 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_ALL, status, ierr)
```

Second example: Copy the (strictly) lower triangular part of a matrix.

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

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

      CALL MPI_COMM_RANK(MPI_ALL, 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_INDEX( 100, MPI_REAL, disp, block, ltype, ierr)

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

Third example: Transpose a matrix

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

```
C      transpose matrix a onto b

       CALL MPI_COMM_RANK(MPI_ALL, myrank)

       CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

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

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

Another approach to the transpose problem:

```
       REAL a(100,100), b(100,100)
       INTEGER  disp(2), blocklen(2), type(2), row, row1, sizeofreal
       INTEGER  myrank, ierr
       MPI_STATUS status

       CALL MPI_COMM_RANK(MPI_ALL, myrank)

C      transpose matrix a onto b

       CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

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

C      create datatype for one row, with the extent of one real number
       disp(1) = 0
       disp(2) = sizeofreal
       type(1)  = row
       type(2)  = MPI_UB
       blocklen(1)  = 1
       blocklen(2)  = 1
       CALL MPI_TYPE_STRUCT( 2, type, disp, blocklen, row1)

       CALL MPI_TYPE_COMMIT( row1, ierr)

C      send 100 rows and receive in column major order
       CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
                  MPI_REAL, myrank, 0, MPI_ALL, status, ierr)
```

Fourth example: manipulate an array of structures.

```
struct Partstruct
   {
   int    class;  /* particle class */
   double d[6];   /* particle coordinates */
   char   b[7];   /* some additional information */
   };

struct Partstruct    particle[1000];

int                  i, dest, rank;
MPI_Comm     comm;


/* build datatype describing structure */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_int, MPI_double, MPI_char};
int          blocklen[3] = {1, 6, 7};
int          disp[3];


/* compute displacements of structure components */

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

MPI_Type_struct( 3, type, disp, blocklen, 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};
int          disp1[4];

/* compute displacements of structure components */

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

/* build datatype describing structure */
```

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


                /* 4.1:
         send the entire array */

MPI_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 index zero (needs to be recomputed
                              if classes change) */
MPI_Datatype Ztype;

int zdisp[1000], zblock[1000], j, k;
int zzblock[2] = {1,1};
int zztype[2], zzindex[2];


/* compute indices of class zero particles */
j = 0;
for(i=0; i < 1000; i++)
  if (particle[i].class==0)
     {
     zindex[j] = i;
     zblock[j] = 1;
     j++;
     }

/* create datatype for class zero particles  */
MPI_Type_indexed( j, Particletype,  zindex, zblock, Zparticles);

/* prepend particle count */
MPI_Address((void *)&j, zzindex);
zzindex[1] = particle;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zztype, zzindex, zzblock, Ztype);

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

```
      /* A probably more efficient way of defining Zparticles */

/* consecutive particles with index zero are handled as one block */
j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
    zindex[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, Particletype, zindex, zblock, Zparticles);


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

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

int          sizeofentry;

MPI_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, MPI_REAL, sizeofentry, 2, Allpairs);
MPI_Type_commit( Allpairs);
MPI_Send( particle.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 */
int disp2[3];
int type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int blocklen2[3] = {1, 2, 1};

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

MPI_Type_struct( 3, type2, disp2, blocklen2, Onepair);
MPI_Commit( Onepair);
```

```
MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);
```

Fifth example: same array manipulations as in the previous example, but using absolute addresses in datatypes.

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

struct Partstruct particle[1000];

           /* build datatype describing first array entry */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          block[3] = {1, 6, 7};
int          disp[3];

MPI_Address( (void *)particle, disp);
MPI_Address( (void *)particle[0].d, disp);
MPI_Address( (void *)particle[0].b, disp);
MPI_Type_struct( 3, stype, sindex, sblock, Particletype);
/* Particletype describes first array entry -- using absolute
   addresses */

                  /* 5.1:
            send the entire array */

MPI_Commit( Particletype);
MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);


                 /* 5.2:
         send the entries of class zero,
         preceded by the number of such entries */

MPI_Datatype Zparticles, Ztype;

int zindex[1000], zblock[1000], i, j, k;
int zzblock[2] = {1,1};
int zztype[2], zzindex[2];


j=0;
```

```
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
    zindex[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_Type_indexed( j, Particletype,  zindex, zblock, Zparticles);
/* Zparticles describe particles with class zero, using
   their absolute addresses*/

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

MPI_Type_commit( Ztype);
MPI_Send( MPI_bottom, 1, Ztype, dest, tag, comm);
```

Sixth example: 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};
int            disp[2];
MPI_Datatype   mpi_utype[2];
int            i,j;

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

MPI_Address( (void *)u, &i);
MPI_Address( (void *)(u+1), &j);
disp[0] = 0; disp[1] = j-i
```

```
type[1] = MPI_UB;

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

type[0] = MPI_FLOAT;
MPI_Type_struct(2, type, disp, blocklen, 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);
```

3.13.7 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 segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, use of **addresses**, i.e. 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 parameter 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.

We say that a datatype is **absolute** if all displacements within this datatype are valid (absolute) addresses; it is **relative** otherwise.

A correct program obeys the following constraints in the use of datatypes:

- If the oldtype argument used in MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_HVECTOR, MPI_TYPE_INDEX, or MPI_TYPE_HINDEX is absolute, then all addresses within it must be to variables contained within the same array or structure; the result datatype is also absolute, and all computed addresses must also fall within the same array or structure.

- If all entries of the array_of_types argument of MPI_TYPE_STRUCT are absolute addresses (computed by MPI_ADDRESS), then the result datatype is also absolute. Each new address computed from an old address must fall within the same array or record as the old address. (Addresses in the old type may not be all within the same array or record.)
- If a communication call is invoked with parameters buf and datatype, then either buf = MPI_BOTTOM and datatype is a handle to an absolute datatype, or buf is set to the address of a program variable, and all displacements in datatype relative to this base yield addresses that are within the same sequential storage as this variable.

In summary, the type constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_HVECTOR, MPI_TYPE_INDEX and MPI_TYPE_HINDEX can be recursively applied to build datatypes that will combine variables that belong to the same sequential storage; variables that do not belong to the same sequential storage can be combined together using one application of MPI_TYPE_STRUCT.

Implementation note:
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.

3.14 Universal communication functions

This section is not part of the MPI definition. Its goal is to indicate how the definition or implementation of the point to point communication functions can be reduced to a small number of primitives. We assume the availability of the primitive functions listed at the end of the section, and the ability to allocate memory for new variables.

We assume that the function MPI_TYPE_COMMIT is not changing the representation of a datatype, so that this function is redundant.

Communication with a count argument that is > 1 can be reduced to communication with count =1, by replicating the datatype argument count times. Communication with a a buf argument that is not MPI_BOTTOM can be reduced to communication with buf = MPI_BOTTOM by creating a suitable displaced new datatype. Thus:

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

```
Type[1] = datatype
MPI_ADDRESS( buf, Disp[1]);
Blocklen[1] = count;
MPI_TYPE_STRUCT( 1, Type, Disp, Blocklen, Newtype)
MPI_SEND( MPI_BOTTOM, 1, Newtype, dest, tag, comm)
```

The same construction applies to all other communication functions with arguments count and datatype. We shall henceforth restrict ourselves to communication that involves only one element with an absolute address (count =1, and buf = MPI_BOTTOM).

It is convenient to to make explicit the **persistence** attribute of communication objects. A persistent communication object need be explicitly deallocated by a MPI_COMMOBJ_FREE operation. On the other hand, an ephemeral communication object is good for a single communication. It is deallocated by the system when the first communication it is used for completes.

The function MPI_COMM_INIT is a new universal function for the creation of communication objects:

MPI_COMM_INIT(datatype, source-dest, tag, comm, op-mode, persistence, request)

IN	datatype	datatype of element sent or received
IN	source-dest	rank of destination or source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
IN	op-mode	one of MPI_STANDARD, MPI_READY, MPI_SYNCHRONOUS or MPI_RECV
IN	persistence	one of MPI_PERSISTENT or MPI_EPHEMERAL
OUT	request	new communication object (handle)

3.14.1 Persistent communication objects

MPI_CREATE_SEND(MPI_BOTTOM, 1, datatype, dest, tag, comm, request) is

```
MPI_COMM_INIT(datatype, dest, tag, comm, MPI_STANDARD,
    MPI_PERSISTENT, request)
```

The functions MPI_CREATE_RSEND, MPI_CREATE_RSEND and MPI_CREATE_RECV are dealt with in a similar manner.

The functions MPI_START and MPI_COMMOBJ_FREE are primitive.

3.14.2 Nonblocking communication initiation

MPI_ISEND(MPI_BOTTOM, 1, datatype, dest, tag, comm, request) is

```
MPI_COMM_INIT(datatype, dest, tag, comm, MPI_STANDARD,
     MPI_EPHEMERAL, request)
MPI_START(request)
```

The functions MPI_IRSEND, MPI_ISSEND and MPI_IRECV are handled in a similar manner.

3.14.3 Communication completion

The two primitive completion operations are MPI_WAITANY and MPI_TESTALL.

The function MPI_WAIT can be implemented by a call to MPI_WAITANY with an array-of-requests argument of length one.

The function MPI_WAITALL can be implemented as a loop where an MPI_WAIT call is executed for each successive request in the array-of-requests.

The function MPI_TEST can be implemented by a call to MPI_TESTALL with an array-ofrequests of length one.

The function MPI_TESTANY can be implemented as a loop where an MPI_TEST call is executed for each successive request in the array-of-request.

3.14.4 Blocking communication

MPI_SEND(MPI_BOTTOM, 1, datatype, dest, tag, comm) is

```
MPI_COMM_INIT(datatype, dest, tag, comm, MPI_STANDARD,
      MPI_EPHEMERAL, request)
MPI_START(request)
MPI_WAIT(request, dontcarestatus)
```

The functions MPI_RSEND and MPI_SSEND are handled in a similar manner.

MPI_RECV(MPI_BOTTOM, 1, datatype, source, tag, comm) is

```
MPI_COMM_INIT(datatype, dest, tag, comm, MPI_RECV, MPI_EPHEMERAL, request)
MPI_START(request)
MPI_WAIT(request, status)
```

3.14.5 Probe and cancel

The functions MPI_PROBE and MPI_CANCEL are primitive.

3.14.6 Return status

The functions MPI_GET_SOURCE, MPI_GET_TAG, MPI_GET_LEN, MPI_PROBE_LEN, and MPI_TEST_CANCELLED are primitive.

These functions are simple macros that access records in a structure.

3.14.7 send-receive and exchange

MPI_SENDRECV(send_buf, send_count, send_type, dest, recv_buf, recv_count, recv_type, source, tag, comm, status) is

```
MPI_ISEND(send_buf, send_count, send_type, dest, tag, comm, request[0])
MPI_IRECV(recv_buf, recv_count, recv_type, source, tag, comm, request[1])
MPI_WAITALL(2, request, status_array)
status = status_array[1]
```

The nonblocking sends and receives can be replaced by their primitive implementation.

The function MPI_SENDRECV_REPLACE can be handled in a similar manner; a temporary buffer need be allocated to replicate the send and receive buffer. A size for this buffer can be computed using the function MPI_TYPE_SIZE.

3.14.8 Derived datatypes

We have outlined in section 3.13.1 how each datatype constructor can be expressed in terms of the next one. Thus all datatype constructors can be expressed in terms of the constructor MPI_TYPE_STRUCT; the functions MPI_TYPE_FREE, MPI_TYPE_EXTENT, MPI_TYPE_SIZE, MPI_TYPE_COUNT and MPI_ADDRESS are primitive.

Chapter 4

Collective Communication

4.1 Introduction

Collective communication is defined to be communication that involves a group of processes. The functions provided by the MPI collective communication include:

- Broadcast from one member to all members of a group.
- Barrier synchronization across all group members
- Gather data from all group members to one member.
- Scatter data from one member to all members of a group.
- Global operations such as sum, max, min, etc., were the result is known by all group members and a variation where the result is known by only one member. The ability to have user defined global operations.
- Scan across all members of a group (also called parallel prefix).
- Broadcast from all members to all members of a group.
- Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all).

While vendors may optimize some of these collective routines for their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions.

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 the chapter on point-to-point functions. 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. In this chapter such processes are called the *root*. Some arguments in the collective functions are specified as "significant only at root". These

[1]Version of Oct 18, 1993

arguments are ignored for all participants except the root, and can be set to any value. 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.

Collective routines 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 the locations in the communication buffer, or any other location that can be referenced by the collective operation. It does not indicate that other processes in the group have started the operation (unless otherwise indicated in the description of the operation). The successful completion of a collective communication call may depend on the execution of a matching call at all processes in the group. Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. Collective communication calls may use the same communicators as point to point communication; MPI guarnatees 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 the correct use of the collective routines can be found at the end of this chapter.

4.2 Communication Functions

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. In this chapter a communicator can be thought of as a group identifier linked with a context. (Inter-communicators, that is between groups communicators, are not allowed in the collective functions.)

4.3 Barrier synchronization

MPI_BARRIER(comm)

IN	comm	communicator (handle)

```
int MPI_BARRIER(MPI_Comm comm )
```

```
MPI_BARRIER(COMM)
    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 Data move functions

Figure 4.1 illustrates the the different collective move functions supported by MPI.

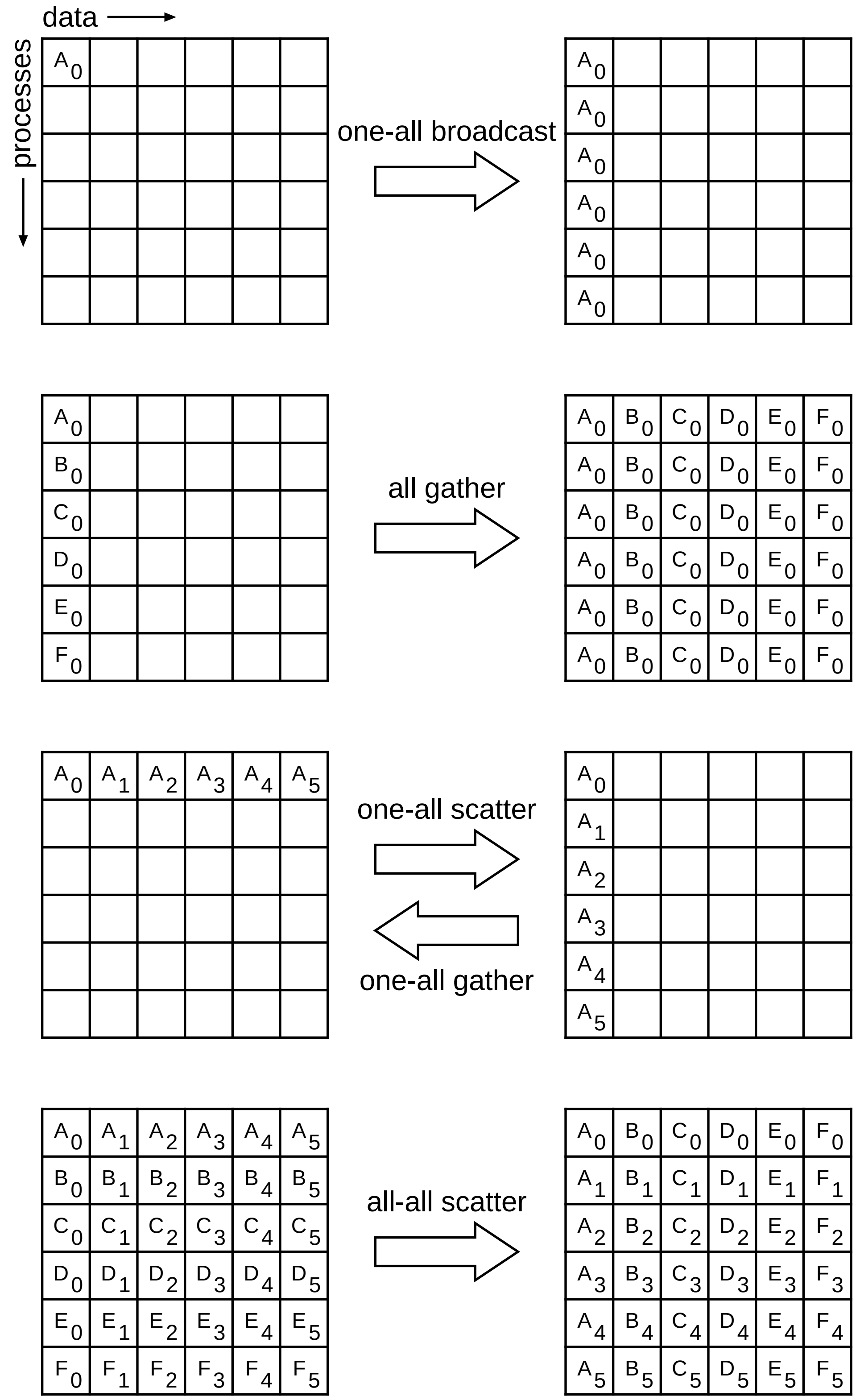

Figure 4.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the one-all broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

4.4.1 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
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)

```
int  MPI_BCAST(void* buf, int count, MPI_Datatype datatype, int root,
             MPI_Comm comm )
```

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

MPI_BCAST broadcasts a message from the process with rank root to all other processes of the group. It is called by all members of group using the same arguments for comm, root and matching arguments for count, datatype. On return the contents of the buffer of the process with rank root is contained in the buffer of the calling process.

4.4.2 Gather

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

```
int MPI_GATHER(void* sendbuf, int sendcnt, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype, int root,
             MPI_Comm comm)
```

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

Each process (including the root process) sends the contents of its send buffer to the root process. The root process receives the messages at stores them in rank order. The outcome is as if each of the `n` processes in the group had executed a call to,

$$\texttt{MPI\_SEND}(\texttt{sendbuf}, \texttt{sendcount}, \texttt{sendtype}, \texttt{root}, ...),$$

and the root had executed `n` calls to,

$$\texttt{MPI\_RECV}(\texttt{recvbuf}[\texttt{i}], \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...),$$

where,

$$\texttt{recvbuf}[\texttt{i}] = \texttt{recvbuf} + \texttt{i} \times \texttt{extent}(\texttt{recvtype}).$$

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 $\times$n, rectype, ...).

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 *exactly* match the type signature of recvcount, recvtype, and the type signature of sendcount, sendtype must be the same on all processes. Exact matching means the basic types in the type signature must match (as in point-to-point communications) and the number of basic types must be the same on both sides (in contrast with point-to-point).

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.

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, displs, recvcounts, recvtype, root, 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
OUT	recvbuf	address of receive buffer (choice, significant only at root)
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)
IN	recvcounts	integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

```
int MPI_GATHERV(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int *displs, int *recvcounts, MPI_Datatype,
```

```
              int root, MPI_Comm comm)
```

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, DISPLS, RECVCOUNTS,
              RECVTYPE, ROOT, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DISPLS, RECVCOUNTS, RECVTYPE, ROOT, COMM,
    IERROR
```

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count of data from each process (since recvcounts is now an array), and 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 has sent a message to the root,

$$\mathtt{MPI\_SEND(sendbuf, sendcount, sendtype, root, ...)},$$

and the root executed `n` receives,

$$\mathtt{MPI\_RECV(recvbuf} + \mathtt{disp[i]} \times \mathtt{extent(recvtype), recvcount[i], recvtype, i, ...)}.$$

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 typemap implied by sendcount, sendtype need not be the same for each process, but it still must exactly match the typemap implied by recvcounts, recvtype. That is, typemap of sendcount, sendtype on process `i` must exactly match typemap of recvcounts[i], recvtype on process root.

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.

For both functions, the specification of count(s), type(s), (displacements), should not cause any location on the root to be written more than once. Such a call is erroneous.

We illustrate the matching conditions with the following examples.

4.4.3 Examples of Usage of MPI_GATHER, MPI_GATHERV

Example 1

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

```
MPI_Comm comm;
MPI_Group mygroup;
int gsize,sendarray[100];
int root, *rbuf;

...

/* The variable comm is set elsewhere in the program
 */
MPI_Comm_group( comm, &mygroup );
MPI_Group_size( mygroup, &gsize );
```

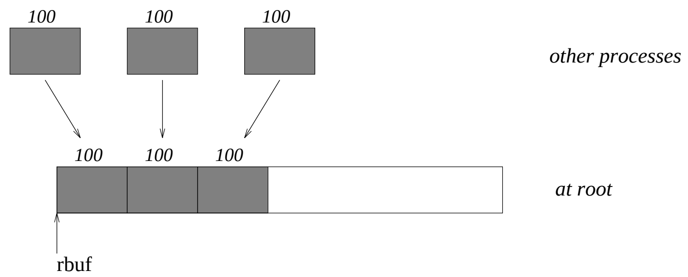

Figure 4.2: The root process gathers 100 ints from each process in the group.

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

Example 2

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

```
MPI_Comm comm;
MPI_Group mygroup;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Datatype rtype;

...

/* The variable comm is set elsewhere in the program
 */
MPI_Comm_group( comm, &mygroup );
MPI_Group_size( mygroup, &gsize );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( rtype );
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

Example 3

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.

```
MPI_Comm comm;
MPI_Group mygroup;
int gsize,sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;
```

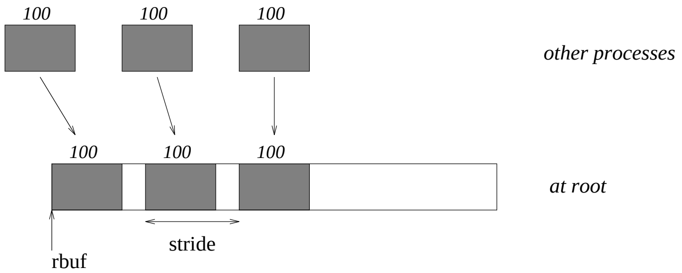

Figure 4.3: The root process gathers 100 `int`s from each process in the group, each set is placed `stride` ints apart.

```
...

MPI_Comm_group( comm, &mygroup );
MPI_Group_size( mygroup, &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;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, displs, rcounts, MPI_INT,
                                                         root, comm);
```

Note that the behavior of MPI_GATHERV is undefined if $stride < 100$.

Example 4

Same as Example 3 on the receiving side, but send the 100 ints from the 0th column of a 100×150 int array, in C. See figure 4.4.

```
MPI_Comm comm;
MPI_Group mygroup;
int gsize,sendarray[100][150];
int root, *rbuf, stride;
MPI_Datatype stype;
int *displs,i,*rcounts;

...

MPI_Comm_group( comm, &mygroup );
MPI_Group_size( mygroup, &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) {
```

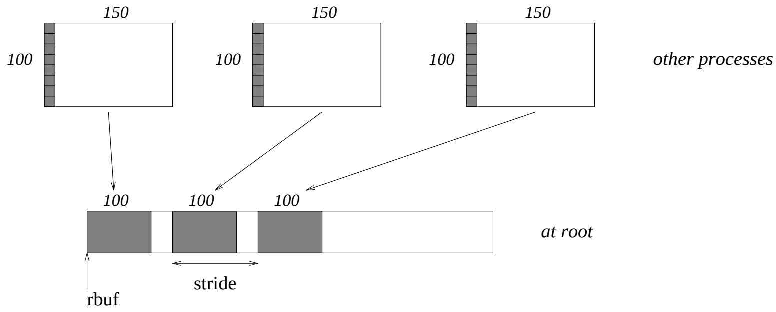

Figure 4.4: The root process gathers column `0` of a 100×150 C array, and each set is placed `stride` ints apart.

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

Example 5

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;
    MPI_Group mygroup;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, stride, myrank;
    MPI_Datatype stype;
    int *displs,i,*rcounts;

    ...

    MPI_Comm_group( comm, &mygroup );
    MPI_Group_size( mygroup, &gsize );
    MPI_Comm_rank( comm, &myrank );
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    for (i=0; i<gsize; ++i) {
        displs[i] = i*stride;
        rcounts[i] = 100-i;     /* note change from previous example */
    }
    /* Create datatype for the column we are sending
```

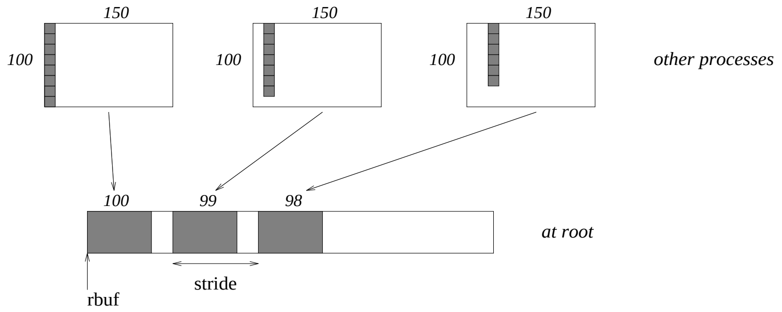

Figure 4.5: The root process gathers 100-i ints from column `i` of a 100×150 C array, and each set is placed `stride` ints apart.

```
 */
MPI_Type_vector( 100-myrank, MPI_INT, 150, 1, &stype);
MPI_Type_commit( stype );
/* sptr is the address of start of "myrank" column
 */
sptr = &sendarray[0][0] + myrank*sizeof(int);
MPI_Gatherv( sptr, 1, stype, rbuf, displs, rcounts, MPI_INT, root, comm);
```

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

Example 6

Same as Example 5, 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 — this was also done in Example 3, 2nd part, section 3.13.6.

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

...

MPI_Comm_group( comm, &mygroup );
MPI_Group_size( mygroup, &gsize );
MPI_Comm_rank( comm, &myrank );
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100-i;
}
/* 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, type, disp, blocklen, &stype );
    MPI_Type_commit( stype );
    sptr = &sendarray[0][0] + myrank*sizeof(int);
    MPI_Gatherv( sptr, 100-myrank, stype, rbuf, displs, rcounts, MPI_INT,
                                                            root, comm);
```

Example 7

Same as Example 5 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See figure 4.6.

```
    MPI_Comm comm;
    MPI_Group mygroup;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, *stride, myrank, bufsize;
    MPI_Datatype stype;
    int *displs,i,*rcounts,offset;

    ...

    MPI_Comm_group( comm, &mygroup );
    MPI_Group_size( mygroup, &gsize );
    MPI_Comm_rank( comm, &myrank );

    stride = (int *)malloc(gsize*sizeof(int));
    ...
    /* stride[i] for i = 0 to gsize-1 is set somehow
     */

    /* we have to compute how large rbuf needs to be
     */
    bufsize = 0;
    for (i=0; i<gsize; ++i) {
        bufsize += stride[i];
    }
    rbuf = (int *)malloc(bufsize*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    offset = 0;
    for (i=0; i<gsize; ++i) {
        displs[i] = offset;
        offset += stride[i];
        rcounts[i] = 100-i;
    }
```

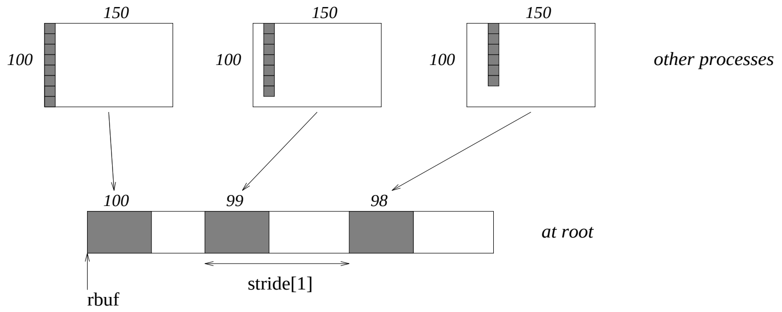

Figure 4.6: The root process gathers 100-i ints from column `i` of a 100×150 C array, and each set is placed `stride[i]` ints apart (a varying stride).

```
/* Create datatype for the column we are sending
 */
MPI_Type_vector( 100-myrank, MPI_INT, 150, 1, &stype);
MPI_Type_commit( stype );
sptr = &sendarray[0][0] + myrank*sizeof(int);
MPI_Gatherv( sptr, 1, stype, rbuf, displs, rcounts, MPI_INT,
                                                    root, comm);
```

Example 8

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;
MPI_Group mygroup;
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_group( comm, &mygroup );
MPI_Group_size( mygroup, &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, type, disp, blocklen, &stype );
    MPI_Type_commit( stype );
    sptr = &sendarray[0][0] + myrank*sizeof(int);
    MPI_Gatherv( sptr, num, stype, rbuf, displs, rcounts, MPI_INT,
                                                           root, comm);
```

4.4.4 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)
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements
IN	root	rank of sending process (integer)
IN	group	communicator (handle)

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

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

MPI_SCATTER is the inverse operation to MPI_GATHER.

The root process sends a message to each process (including itself); each process stores the incoming message in its receive buffer. The outcome is as if the root exectuted `n` send operations,

$$\texttt{MPI\_SEND}(\texttt{sendbuf} + \texttt{i} \times \texttt{extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i}, ...),$$

and each process executed a receive,

$$\texttt{MPI\_RECV}(\texttt{recvbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

An alternative description is that the root send a message with MPI_SEND(sendbuf, sendcount$\times$n, sendtype, ...); this message is split into `n` equal segments, and the $i-th$ segment is sent to the i-th process in the group; and each process receives this message as above.

The type signature of sendcount, sendtype must *exactly* match the type signature of recvcount, recvtype, and the type signature of sendcount, sendtype must be the same on all processes.

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.

MPI_SCATTERV(sendbuf, displs, sendcounts, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	displs	integer array (of length group size). Entry `i` specifies the displacement (relative to sendbuf from which to take the outgoing data to process `i`
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each processor
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements
IN	root	rank of sending process (integer)
IN	group	communicator (handle)

```
int MPI_SCATTERV(void* sendbuf, int *displs, int *sendcounts,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype, int root, MPI_Comm comm)
```

```
MPI_SCATTERV(SENDBUF, DISPLS, SENDCOUNTS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER DISPLS, SENDCOUNTS, 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), and 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 excuted n send operations,

$$\mathtt{MPI\_SEND}(\mathtt{sendbuf} + \mathtt{displs}[\mathtt{i}] \times \mathtt{extent}(\mathtt{sendtype}), \mathtt{sendcount}[\mathtt{i}], \mathtt{sendtype}, \mathtt{i}, ...),$$

and each process executed a receive,

$$\mathtt{MPI\_RECV}(\mathtt{recvbuf}, \mathtt{recvcount}, \mathtt{recvtype}, \mathtt{i}, ...).$$

The typemap implied by sendcount, sendtype need not be the same for each process, but it still must exactly match the typemap implied by recvcounts, recvtype. That is, typemap of sendcount[i], sendtype on process root must exactly match typemap of recvcount, recvtype on process i.

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.

For both functions, the specification of count(s), type(s), (displacements), should not cause any location on the root to be read more than once.

4.4.5 Examples of Usage of MPI_SCATTER, MPI_SCATTERV

Example 9

The reverse of Example 1. Scatter sets of 100 ints from the root to each process in the group. See figure 4.7.

```
    MPI_Comm comm;
    MPI_Group mygroup;
    int gsize,*sendbuf;
    int root, rbuf[100];

    ...

    /* The variable comm is set elsewhere in the program
     */
    MPI_Comm_group( comm, &mygroup );
    MPI_Group_size( mygroup, &gsize );
    sendbuf = (int *)malloc(gsize*100*sizeof(int));
    MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 10

The reverse of Example 3. 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.

```
    MPI_Comm comm;
    MPI_Group mygroup;
```

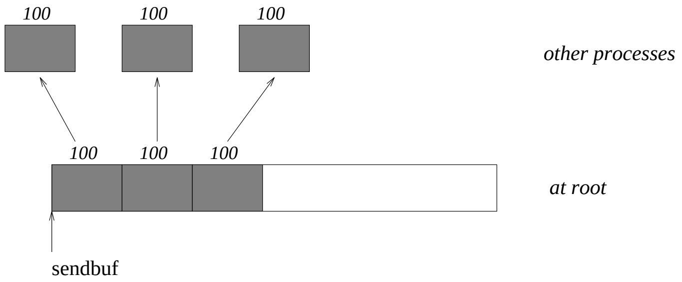

Figure 4.7: The root process scatters sets of 100 `int`s to each process in the group.

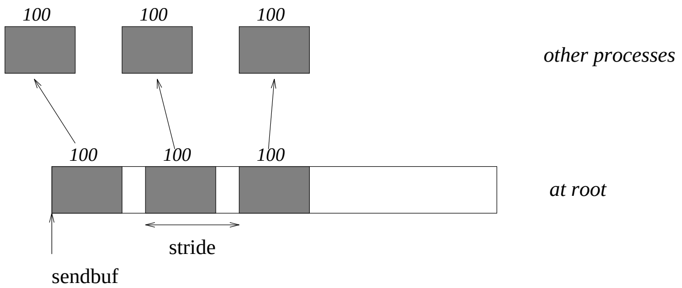

Figure 4.8: The root process scatters sets of 100 `int`s, moving by `stride` ints from send to send in the scatter.

```
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;

...

MPI_Comm_group( comm, &mygroup );
MPI_Group_size( mygroup, &gsize );
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatter( sendbuf, displs, scounts, MPI_INT, rbuf, 100, MPI_INT,
                                                          root, comm);
```

Example 11

The reverse of Example 7. 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.

```
MPI_Comm comm;
MPI_Group mygroup;
int gsize,rbuf[100][150],*rptr;
int root, *sendbuf, myrank, bufsize, *stride;
```

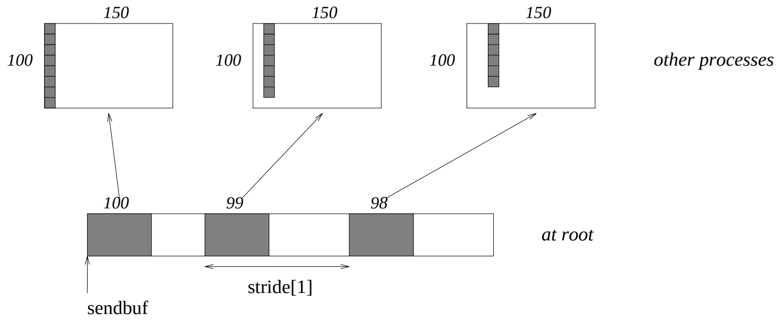

Figure 4.9: The root scatters gathers blocks of 100-i ints into column `i` of a 100×150 C array. At the sending side, the blocks are `stride[i]` ints apart.

```
    MPI_Datatype rtype;
    int i, *displs, *scounts, offset;

    ...

    MPI_Comm_group( comm, &mygroup );
    MPI_Group_size( mygroup, &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, MPI_INT, 150, 1, &rtype);
    MPI_Type_commit( rtype );
    rptr = &rbuf[0][0] + myrank*sizeof(int);
    MPI_Scatterv( sendbuf, displs, scounts, MPI_INT, rptr, 1, rtype,
                                                            root, comm);
```

4.4.6 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
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
IN	comm	communicator (handle)

```
int MPI_ALLGATHER(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype, MPI_Comm comm)
```

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM)
    <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 typemap of sendcount, sendtype must exactly match the typemap of recvcount, recvtype. Thus, the outcome of a call to MPI_ALLGATHER(...) is as if all processes executed n calls to,

$$\mathtt{MPI\_GATHER}(\mathtt{sendbuf}, \mathtt{sendcount}, \mathtt{sendtype}, \mathtt{recvbuf}, \mathtt{recvcount}, \mathtt{recvtype}, \mathtt{root}, \mathtt{comm}),$$

for $\mathtt{root} = 0, \cdots, \mathtt{n} - 1$.

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, displs, recvcounts, 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
OUT	recvbuf	address of receive buffer (choice)
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`
IN	recvcounts	integer array (of length group size) containing the number of elements that are received from each process
IN	recvtype	data type of receive buffer elements
IN	comm	communicator (handle)

```
int MPI_ALLGATHERV(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int *displs, int *recvcounts, MPI_Datatype,
              MPI_Comm comm)
```

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, DISPLS, RECVCOUNTS,
              RECVTYPE, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DISPLS, RECVCOUNTS, RECVTYPE, COMM, IERROR
```

MPI_ALLGATHERV can be thought of as MPI_GATHERV, 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. These blocks need not all be the same size. The typemap of sendcount, sendtype on process `j` must exactly match the typemap of recvcounts[j], recvtype, (on all processes). The outcome is as if all processes executed calls to,

$$\mathtt{MPI\_GATHERV}(\mathtt{sendbuf}, \mathtt{sendcount}, \mathtt{sendtype}, \mathtt{recvbuf}, \mathtt{displs}, \mathtt{recvcounts}, \mathtt{recvtype},$$

$$\mathtt{root}, \mathtt{comm}),$$

for $\mathtt{root} = 0, \cdots, \mathtt{n} - 1$.

For both MPI_ALLGATHER and MPI_ALLGATHERV, all arguments on all processes are significant. The argument comm must have identical values on all processes.

4.4.7 All-to-All Scatter/Gather

MPI_ALLTOALL(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
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
IN	comm	communicator (handle)

```
int MPI_ALLTOALL(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype, MPI_Comm comm)
```

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

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The `j`-th block sent from process `i` is received by process `j` and is placed in the `i`-th block of recvbuf. The typemap of sendcount, sendtype must exactly match the typemap of recvcount, recvtype.

The outcome is as if each process executed a send to each process (itself included) with a call to,

$$\mathtt{MPI\_SEND}(\mathtt{sendbuf} + \mathtt{i} \times \mathtt{extent}(\mathtt{sendtype}), \mathtt{sendcount}, \mathtt{sendtype}, \mathtt{i}, \ldots),$$

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

$$\mathtt{MPI\_RECV}(\mathtt{recvbuf} + \mathtt{i} \times \mathtt{extent}(\mathtt{recvtype}), \mathtt{recvcount}, \mathtt{i}, \ldots).$$

MPI_ALLTOALLV(sendbuf, sdispls, sendcounts, sendtype, recvbuf, rdispls, recvcounts, recvtype, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	sdispls	integer array (of length group size). Entry `j` specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process `j`
IN	sendcounts	integer array equal to the group size specifying the number of elements to send to each processor
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer (choice)
IN	rdispls	integer array (of length group size). Entry `i` specifies the displacement (relative to recvbuf at which to place the incoming data from process `i`
IN	recvcounts	integer array equal to the group size specifying the maximum number of elements that can be received from each processor
IN	recvtype	data type of receive buffer elements
IN	comm	communicator (handle)

```
int MPI_ALLTOALLV(void* sendbuf, int *sdispls, int *sendcounts,
             MPI_Datatype sendtype, void* recvbuf, int *rdispls,
             int *recvcounts, MPI_Datatype, MPI_Comm comm)
```

```
MPI_ALLTOALLV(SENDBUF, SDISPLS, SENDCOUNTS, SENDTYPE, RECVBUF, RDISPLS,
             RECVCOUNTS, RECVTYPE, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SDISPLS, SENDCOUNTS, SENDTYPE, RDISPLS, RECVCOUNTS, 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, but the typemap of sendcounts[j], sendtype on process `i` must exactly match the typemap of recvcounts[i], recvtype on process `j`.

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

$$\mathtt{MPI\_SEND}(\mathtt{sendbuf} + \mathtt{displs}[\mathtt{i}], \mathtt{sendcount}[\mathtt{i}], \mathtt{sendtype}, \mathtt{i}, ...),$$

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

$$\mathtt{MPI\_RECV}(\mathtt{recvbuf} + \mathtt{displs}[\mathtt{i}], \mathtt{recvcount}[\mathtt{i}], \mathtt{recvtype}, \mathtt{i}, ...).$$

For both MPI_ALLTOALL and MPI_ALLTOALLV, all arguments on all processes are significant. The argument comm must have identical values on all processes.

Implementation note: Although the discussion of collective communication in terms of point to point operation implies that each message is transfered directly from sender to receiver, implementations may use a tree communication pattern, where messages are forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient.

4.5 Global Compute Operations

The functions in this section perform one of the following operations across all the members of a group:

global max on integer and floating point data types

global min on integer and floating point data types

global sum on integer and floating point data types

global product on integer and floating point data types

global AND on logical and integer data types

global OR on logical and integer data types

global XOR on logical and integer data types

rank of process with maximum value

rank of process with minimum value

user defined (associative) operation

user defined (associative and commutative) operation

4.5.1 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 (basic types only)
IN	op	operation (MPI-defined constant)
IN	root	rank of root process (integer)
IN	comm	communicator (handle)

```
int MPI_REDUCE(void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, void op, int root, MPI_Comm comm)
```

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

Combines the values provided in the send buffer of each process in the group, using the operation op, and returns the combined value in the receive buffer of the process with rank root. Each process can provide one value, or a sequence of values, 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 floating point numbers, then recvbuf(1) = global max(sendbuf(1)) and recvbuf(2) = global max(sendbuf(2)). All send buffers should define sequences of equal length of entries all of the same data type, where the type is a **basic** MPI datatype and one of those allowed for operands of op (see below). For all operations the number of elements in the send buffer are the same as for the receive buffers. For all operations, the type of elements in the send buffer are the same as for the receive buffers.

The operation defined by op is associative and commutative, and the implementation can take advantage of associativity and commutativity in order to change order of evaluation. The routine is called by all group members using the same arguments for count, datatype, op, root and comm.

We list below the supported options for op.

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	maximum value and rank of process with it
MPI_MINLOC	minimum value and rank of process with it

The MPI_MINLOC (MPI_MAXLOC) operations return both minimum (maximum) values and the ranks of processes containing those values. The potentially mixed-type nature of the output buffer is a concern, so MPI treats the buffers uniformly and coerces the ranks to the same type as the values. When MPI_MINLOC or MPI_MAXLOC are invoked, the input buffer should contain m elements of a data type to which the operation MPI_MIN or MPI_MAX can be applied, followed by space for another m elements of the same type. Internally, the function will coerce the rank of the calling process (an integer type) to the type of the data values, and carry these values along during the reduction. The operation returns at the root the m minimum (or maximum) values, followed by the m ranks of the processes containing these values. To recover the ranks of the processes as integers, the second set of values can now be copied out of the buffer and coerced to integer values.

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

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

Note that ties are resolved in favor of the process with lower rank and hence this operation is associative and commutative.

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

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

Operation / Type Compatibility

All of the options for op in MPI_REDUCE do not apply for each of the possible MPI basic datatypes. We enumerate the allowed combinations here. First, define groups of MPI basic datatypes in the following way.

C integer: `MPI_CHAR, MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG`

Fortran integer: `MPI_INTEGER`

Floating point: `MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION`

Logical: `MPI_LOGICAL`

Complex: `MPI_COMPLEX`

Byte: `MPI_BYTE`

Now, the valid datatypes for each option is specified below.

MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC	C integer, Fortran integer, Floating point
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical
MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Byte

Example of Reduce

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: a[]
     */
    double a[30];
    double in[60],out[60];
    int i,outranks[30];

    for (i=0; i<30; ++i) {
in[i] = a[i];
    }
    MPI_Reduce( in, out, 30, MPI_DOUBLE, MPI_MAXLOC, root, comm );
    /* At this point, the answer resides on process root
     */
    if (myrank == root) {
/* read ranks out and coerce back to ints
 */
        for (i=0; i<30; ++i) {
    outranks[i] = out[30+i];
        }
    }
```

4.5.2 User-Reduce

MPI_USER_REDUCE(sendbuf, recvbuf, count, datatype, function, root, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice, significant only at root)
IN	count	number of elements in input buffer (integer)
IN	datatype	data type of elements of input buffer
IN	function	user defined function
IN	root	rank of root process (integer)
IN	comm	communicator (handle)

```
int MPI_USER_REDUCE(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void function, int root, MPI_Comm comm)
```

```
MPI_USER_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, ROOT, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

Similar to the reduce operation function above except that a user supplied function is used. function is a function with three arguments. A C prototype for such function is `void f( invec, inoutvec, *len)`. Both invec and inoutvec are arrays with *len entries. The function computes element-wise a commutative and associative operation on each pair of entries and returns the result in inoutvec. A pseudo-code for function is given below, where op is the commutative and associative operation defined by function.

```
for(i=0; i < *len; i++) {
        inoutvec[i] op= invec[i]
}
```

The type of the elements of invec and of inoutvec match the type of the elements of the send buffers and the receive buffer. No MPI functions may be called inside the user defined function.

The addition of the third argument, *len, in function allows the system to avoid calling function for each element in the input buffer. Rather, the system can choose to apply function to chunks of input.

Example of User-Reduce

Compute the product of an array of complex numbers, in C.

```
typedef struct _complex {
    double real,imag;
} Complex;

/* the user-defined function
 */

void myProd( Complex invec[], Complex inoutvec[], int *len )
{
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inoutvec[i].real*invec[i].real -
                   inoutvec[i].imag*invec[i].imag;
        c.imag = inoutvec[i].real*invec[i].imag +
                   inoutvec[i].imag*invec[i].real;
        inoutvec[i] = c;
    }
}

/* and, to call it...
 */
...

    /* each process has an array of 100 Complexes
     */
```

```
    Complex a[100], answer[100];
    MPI_Datatype ctype;

    /* explain to MPI what a "complex" is (remember, this is C!)
     * Note we are assuming compiler implements type Complex as
     * two contiguous doubles
     */
    MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
    MPI_Type_commit( ctype );

    MPI_User_reduce( a, answer, 100, ctype, myProd, root, comm );

    /* At this point, the answer, which consists of 100 Complexes,
     * resides on process root
     */
```

MPI_USER_REDUCEA(sendbuf, recvbuf, count, datatype, function, root, comm)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice, significant only at root)
IN	count	number of elements in input buffer (integer)
IN	datatype	data type of elements of input buffer
IN	function	user defined function
IN	root	rank of root process (integer)
IN	comm	communicator (handle)

```
int MPI_USER_REDUCEA(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, void function, int root, MPI_Comm comm)
```

```
MPI_USER_REDUCEA(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, ROOT, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

Identical to MPI_USER_REDUCE, except that the operation defined by function is not required to be commutative, but only associative. Thus, the order of computation can be modified only using associativity. Use of this function means that the implementation *cannot* try and change the order of computation of the reduce by assuming commutativity.

Implementation note:

MPI_USER_REDUCEA and MPI_USER_REDUCE can have identical implementations, if one does not wish to take advantage of commutativity.

Implementation note: The addition of the third argument, *len in function allow the system to avoid calling function for each element in the input buffer; rather, the system can choose to apply function to chunks of inputs, where the size of the chunk is chosen by the system so as to optimize communication and computation pipelining. E.g., *len could be set to be the typical packet size in the communication subsystem.

MPI includes variants of each of the reduce operations where the result is known to all processes in the group on return.

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
IN	op	operation
IN	comm	communicator (handle)

```
int MPI_ALLREDUCE(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void op, MPI_Comm comm)
```

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

Same as the MPI_REDUCE operation function except that the result appears in the receive buffer of all the group members.

MPI_USER_ALLREDUCE(sendbuf, recvbuf, count, datatype, function, 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
IN	function	user defined function
IN	comm	communicator (handle)

```
int MPI_USER_ALLREDUCE(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void function, MPI_Comm comm)
```

```
MPI_USER_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, COMM, IERROR
```

Same as the MPI_USER_REDUCE operation function except that the result appears in the receive buffer of all the group members.

MPI_USER_ALLREDUCEA(sendbuf, recvbuf, count, datatype, function, 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
IN	function	user defined function
IN	comm	communicator (handle)

```
eint MPI_USER_ALLREDUCEA(void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, void function, MPI_Comm comm)
```

```
MPI_USER_ALLREDUCEA(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, COMM, IERROR
```

Same as MPI_USER_REDUCEA, except that the result appears in the receive buffer of all the group members.

Implementation note: The all-reduce operations can be implemented as a reduce, followed by a broadcast. However, a direct implementation can lead to better performance.

4.5.3 Reduce-Scatter

MPI also includes variants of each of the reduce operations where the result is scattered to all processes in the group on return.

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

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	datatype	data type of elements of input buffer
IN	op	operation
IN	comm	communicator (handle)

```
int MPI_REDUCE_SCATTER(void* sendbuf, void* recvbuf, int *recvcounts,
            MPI_Datatype datatype, void op, MPI_Comm comm)
```

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

MPI_REDUCE_SCATTER first does a componentwise reduction on vectors provided by the processes. 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 recvcounts[i] elements. The i-th segment is sent to process with rank i.

Implementation note: The MPI_REDUCE_SCATTER routine is functionally equivalent to: A MPI_REDUCE operation function with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster.

4.5.4 Scan

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	starting address of send buffer (choice)
IN	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
IN	op	operation
IN	comm	communicator (handle)

```
int MPI_SCAN(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void op, MPI_Comm comm )
```

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

MPI_SCAN is used to perform a parallel prefix with respect to an associative and commutative reduction operation 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,...,i. The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE.

MPI_USER_SCAN(sendbuf, recvbuf, count, datatype, function, comm)

IN	sendbuf	address of input buffer
OUT	recvbuf	address of output buffer
IN	count	number of elements in input and output buffer (integer)
IN	datatype	data type of buffer
IN	function	user provided function
IN	comm	communicator (handle)

```
int MPI_USER_SCAN(void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, void function, MPI_Comm comm)
```

```
MPI_USER_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, COMM, IERROR
```

Same as the MPI_SCAN operation function except that a user supplied function is used. function is an associative and commutative function with an input vector, an inout vector, and a length argument. The types of the two vectors and of the returned values all agree. See MPI_USER_REDUCE for more details.

MPI_USER_SCANA(sendbuf, recvbuf, count, datatype, function, comm)

IN	sendbuf	address of input buffer (choice)
OUT	recvbuf	address of output buffer (choice)
IN	count	number of elements in input and output buffer (integer)
IN	datatype	data type of buffer
IN	function	user defined function
IN	comm	communicator (handle)

```
int MPI_USER_SCANA(void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, void function, MPI_Comm comm)
```

```
MPI_USER_SCANA(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, COMM, IERROR
```

Same as MPI_USER_SCAN, except that the user-defined operation need not be commutative.

Implementation note:
MPI_USER_SCAN could be implemented as MPI_USER_SCANA, though for some architectures it may be possible to arrive at a faster MPI_USER_SCAN by taking advantage of commutativity.

4.6 Correctness

4.6.1 Synchronization Side-Effects

A correct program should invoke collective communications so that deadlock will not occur, whether collective communication is synchronizing or not. The following examples illustrate dangerous use of collective routines.

The following example illustrates an erroneous program.

```
/* Example A */
switch(rank)
   {
   case 0: { MPI_Bcast(var1, count, type, 0, comm);
             MPI_Send(var2, count, type, 1, tag, comm);
             break;
           }
   case 1: { MPI_Recv(var2, count, type, 0, tag, comm);
             MPI_Bcast(var1, count, type, 0, comm);
             break;
           }
   }
```

Process zero executes a broadcast, followed by a blocking send operation; process one first executes a matching blocking receive, followed by the matching broadcast call. 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 following example is correct, but non-deterministic:

```
/* Example B */
switch(rank)
   {
    case 0: { MPI_Bcast(var1, count, type, 0, comm);
              MPI_Send(var2, count, type, 1, tag, comm);
              break;
            }
    case 1: { MPI_Recv(var2, count, type, MPI_ANY_SOURCE, tag, comm);
              MPI_Bcast(var1, count, type, 0, comm);
              MPI_Recv(var2, count, type, MPI_ANY_SOURCE, tag, comm);
              break;
            }
    case 2: { MPI_Send(var2, count, type, 1, tag, comm);
              MPI_Bcast(var1, 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, with different matchings of sends and receives are illustrated below.

```
           First Execution

   0             1               2
                      /-----  send
                recv <-/
broadcast     broadcast       broadcast
  send ---\
           \--> recv

           Second Execution

  0              1               2
broadcast
  send ---\
           \-->  recv
               broadcast       broadcast
                         /---  send
                 recv <---/
```

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. To assume that collective communication functions do or do not have certain synchronizing side-effects is non-portable.

4.6.2 Multiple Calls to Collective

It is the user's responsibility to make sure that there are no two concurrently executing collective calls that use the same communicator on the same process. Since all collective communication calls are blocking this restriction only affects multithreaded implementations. On the other hand, it is legitimate for one process to start a new collective communication call even though a previous call that uses the same communicator has not yet terminated on another process. As illustrated in the following example:

```
/* Example C */
 MPI_Bcast(var1, count, type, 0, comm);
 MPI_Bcast(var2, count, type, 1, comm);
```

In a nonsynchronizing implementation of broadcast, process zero may start executing the second broadcast before process one terminated the first broadcast. Both process zero and one may terminate their two broadcast calls before other processes have started their calls. It is the implementor's responsibility to ensure this will not cause any error.

Implementation note:
Assume that broadcast is implemented using point-to-point MPI communication. The following two rules are satisfied:

1. All receives specify their source explicitly (no wildcards).
2. Each process sends all messages that pertain to one collective call before sending any message that pertain to a subsequent collective call.

Then messages belonging to successive broadcasts cannot be confused, as the order of point-to-point messages is preserved. This is true, in general, for any collective library.

A collective communication may execute in a context while point-to-point communications that use the same context are pending, or occur concurrently. This is illustrated in example B above, the first process may receive a message sent with the context of communicator comm while it is executing a broadcast with the same communicator. It is the implementor's responsibility to ensure this will not cause any confusion.

Implementation note: Assume that collective communications are implemented using point-to-point MPI communication. Then, in order to avoid confusion, whenever a communicator is created, a "hidden communicator" need be created for collective communication. A direct implementation of MPI collective communication can achieve a similar effect more cheaply, e.g., by using a hidden tag or context bit to indicate whether the communicator is used for point-to-point or collective communication.

Chapter 5

Groups, Contexts, Communicators, and Cacheing

5.1 Introduction

We begin this chapter with a perspective of why MPI has included features that are not found in existing vendor systems, and why these are important. One of the main motivations for the features described here is to allow for the systematic development of parallel libraries.

5.1.1 Why we need libraries

Parallel libraries are needed to encapsulate the distracting complications associated with distributed realizations of key algorithms. They 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 dependent on size of system, or type of floating point), this too needs to be hidden from the user.

5.1.2 Why existing systems do not support libraries

It is difficult to write libraries with message passing systems that vendors produce, because they have no way for the library to insulate itself effectively from the other point-to-point message passing that goes on in the system. For instance, message tags are a means for a library to hide its messages from the user. However, many libraries may choose the same tags, or the vendor's library may itself indiscriminately use tags when implementing collective operations (*e.g.*, global sum); in that case, collective and point-to-point messages may get mixed up.

Collective operations, such as reduce, broadcast, and so on, are not defined in all vendor systems, and when defined are most often only available over the full set of processors (or processes) allocated to the user. Libraries need to work on different collections of processes, and to have available collective operations that only impact the processes that care about the collective operation.

A third issue is that libraries don't want to describe point-to-point communication in terms of low-level, hardware dependent names. In fact, they might like to use virtual names for processes that somehow reflect the relationship of processes to one another in the distributed data structure natural to an algorithm. Libraries (or program modules) that execute concurrently on different processes may want to use the same process names for their internal communication.

5.1.3 What features are needed to support libraries

So, we have established three key areas that will support the creation of libraries:

- Safe communication space, that guarantees that a library can communicate as it needs to,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processors,
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures.

5.1.4 What these features are called in MPI

The features that implement these important concepts are as follows, in MPI:

- **Contexts** provide the ability to have separate safe "universes" of message-passing in MPI. A context is something like an additional tag that differentiates messages, but the system manages it, and the user never manipulates contexts. We say that communicators (below) are a specific context of communication, when we are speaking loosely. The use of separate communication contexts by distinct libraries (or distinct library invocations) will insulate communication internal to the library execution from external communication. This allows to invoke the library even if there are pending communications, and avoids the need to synchronize entry or exit into library code.
- **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.
- **Communicators** (see [13]) encapsulate all of these ideas in an object that provides the right scope for all communication operations in MPI.

The current practice in many communication libraries is that there is a a unique, predefined communication universe that includes all processes available when the library 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) involves all processes. This practice can be followed in MPI by using the predefined communicator MPI_COMM_WORLD. Users that are satisfied with such practice can plug in MPI_COMM_WORLD wherever a communicator argument is required, and ignore the rest of this chapter.

The discussion has dealt so far with **intra-communication**: communication within a group. MPI also supports **inter-communication**: communication across groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with the other using local ranks within the second module for addressing. 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 bind two groups together with a context shared by both groups. MPI provides mechanisms for creating and manipulating inter-communicators. They are used in point-to-point communication in the same manner as regular **intra-communicators**. Users that do not need inter-communication in their applications can safely ignore this extension.

Communicators also provide a "cacheing" 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 further adorn communicators, 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.

5.2 Basic Concepts

5.2.1 Groups

A **group** is an ordered set of process identifiers (henceforth processes); process identifiers are implementation-dependent objects. Each process in a group is associated with an integer **rank**, starting 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).

Implementation note: A group may be represented by a virtual-to-real process-address-translation table. Each communicator object would have a pointer to such a table.

5.2.2 Contexts

A **context** is the MPI mechanism for partitioning communication space. A defining property of a context is that a message sent in a context cannot be received in another context. Contexts are not explicit MPI objects; they appear only along with groups in communicators. Distinct communicators in the same process have distinct contexts. A context is essentially a tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication.

Implementation note: A possible implementation is for a context to be a label attached to messages and matched on receive. Each intra-communicator stores the value of that label. Communicator generating functions use a collective communication to agree on a new group-wide unique context. In inter-communication, it is more convenient to use two labels, one used by group A to send and group B to receive, and the second used by group B to send and group A to receive. Since contexts are not explicit objects, other implementations are possible.

5.2.3 Communicators

Communicators bring together the concepts of group and context. Furthermore, to support implementation-specific optimizations, and virtual topologies, they "cache" additional information. The source and destination of a message is identified by the rank of that process within the group. For collective communication, the 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 local process addressing.

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

5.2.4 Predefined Communicators

An initial 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 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 simultaneouly 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 do not appear as pre-defined constants, 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. An inquiry function MPI_GET_HOST is provided in Chapter 7 so that MPI programs can determine if there is a host process, and find its rank in MPI_COMM_WORLD if it should exist. 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 MPI_UNDEFINED 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 ranks1 and ranks2 arrays (integer)
IN	ranks1	array of zero or more valid ranks in group1
IN	group2	group2 (handle)
OUT	ranks2	array of corresponding ranks in group2, MPI_UNDEFINED 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
```

5.3.2 Group Constructors

These functions allow construction of 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).

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.

MPI_GROUP_INCL(group, n, ranks, newgroup)

IN	group	group (handle)
IN	n	number of elements in array ranks (and size of newgroup)
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

```
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 empty. This function can, for instance, be used to reorder the elements of a group.

MPI_GROUP_EXCL(group, n, ranks, newgroup)

IN	group	group (handle)
IN	n	number of elements in array ranks
IN	ranks	array of integer ranks in group not to appear in newgroup
OUT	newgroup	new group derived from above, preserving the order defined by group (handle)

```
int MPI_Group_excl(MPI_group group, int n, int *ranks, MPI_Group *newgroup)
```

```
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS, NEWGROUP, IERROR
```

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.

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)

IN	group	group (handle)
IN	n	number of triplets in array ranges (integer)
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group or processes to be included in newgroup
OUT	newgroup	new group derived from above, in the order defined by ranges (handle)

```
int MPI_Group_range_incl(MPI_group group, int n, int **ranges,
             MPI_Group *newgroup)
```

```
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(*,*), NEWGROUP, IERROR
```

If ranges consist of the triplets

$$(first_1, last_1, stride_1), ..., (first_n, last_n, stride_n)$$

then newgroup consists of the sequence of processes in group with ranks

$$first_1, first_1 + stride_1, ..., first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, ...$$

$$first_n, first_n + stride_n, ..., first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$$

Each computed rank must be a valid rank in group and all computed ranks must be distinct, 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 ranks 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 newgroup.
OUT	newgroup	new group derived from above, preserving the order in group (handle)

```
int MPI_Group_range_excl(MPI_group group, int n, int **ranges,
             MPI_Group *newgroup)
```

```
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(*,*), NEWGROUP, IERROR
```

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous.

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

Note: the range operations do not explicitly enumerate ranks, and therefore are more scalable if implemented efficiently. Hence, we recommend MPI programmers to use them whenenever possible, as high-quality implementations will take advantage of this fact.

5.3.3 Group Destructors

MPI_GROUP_FREE(group)

INOUT	group	group (handle)

```
int MPI_Group_free(MPI_group *group)
```

```
MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR
```

This operation marks a group object for deallocation. The handle group is set to null by the call. The group object is actually deallocated only if there are no other active references to it. If a group has been bound to one or more communicators, then the user has to free all "copies" of the group.

Implementation note:
One can keep a reference count that is incremented for each call to MPI_COMM_MAKE and MPI_COMM_DUP, and 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.

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. (High quality implementations will be able to amortize these overheads over several calls.)

5.4.1 Communicator Accessors

The following are all local operations.

MPI_COMM_SIZE(comm, size)

IN	comm	communicator (handle)
OUT	size	number of processes in the group of comm (integer)

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

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
```

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
```

5.4.2 Communicator Constructors

The following are collective functions that are invoked by all processes in the group associated with comm.

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 (defined above).

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 all its cached information. Returns in newcomm a new communicator with the same group, same cached information, but a new context.

Implementation note: 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.

MPI_COMM_MAKE(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_make(MPI_Comm comm, MPI_group group, MPI_Comm *newcomm)
```

```
MPI_COMM_MAKE(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 a null handle 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.

Implementation note: Since all processes calling MPI_COMM_DUP or MPI_COMM_MAKE 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. If new communicators are created without synchronizing the processes involved then the commucation system should be able to cope with messages arriving in a context that has not yet been allocated at the receiving process.

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assigment (integer)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

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

```
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the parameter 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 a null communicator handle. This is a collective call, but each process can provide different values for color and key.

A call to MPI_COMM_MAKE(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.

5.4.3 Communicator Destructors

MPI_COMM_FREE(comm)

INOUT	comm	communicator to be destroyed (handle)

```
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 null. The object is actually deallocated only if there are no other active references to it. The associated group may also be deallocated, if there are no active references to it. It is the user responsibility to free each copy of a communicator object.

Implementation note: A reference count mechanism can 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 local, though the debugging version of an MPI library might choose to synchronize.

5.5 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. It also describes a name service that simplifies writing programs containing inter-communication.

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."

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."

An inter-communication operation 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.

Local and remote groups of an inter-communicator are not required to be disjoint. An intra-communicator can be seen as a particular case of an inter-communicator, where local and remote groups happen to be identical. The semantics of accessor function are defined so as to be consistent with this interpretation. 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 any of the constructor routines for intra-communicators.

Implementation note: 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 $\mathbf{C}_{\mathcal{P}}$, and a process **Q** in group $\mathcal{Q}$, which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$. (Note that $\mathcal{P}$ and $\mathcal{Q}$ do not have to be distinct.) Then

- $\mathbf{C}_{\mathcal{P}}$**.group** describes the group $\mathcal{Q}$ and $\mathbf{C}_{\mathcal{Q}}$**.group** describes the group $\mathcal{P}$.
- $\mathbf{C}_{\mathcal{P}}$**.send_context** = $\mathrm{C}_{\mathcal{Q}}$**.receive_context** and the context is unique in $\mathcal{Q}$;
  $\mathbf{C}_{\mathcal{P}}$**.receive_context** = $\mathbf{C}_{\mathcal{Q}}$**.send_context** and this context is unique in $\mathcal{P}$.
- $\mathbf{C}_{\mathcal{P}}$**.source** is rank of **P** in $\mathcal{P}$ and $\mathbf{C}_{\mathcal{Q}}$**.source** is rank of **Q** in $\mathcal{Q}$.

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 parameter using the inter-communicator. Then **Q** matches **receive_context** to the message context and source parameter to the message source.

The same algorithm is appropriate for intra-communicators as well.

5.5.1 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, FLAG, IERROR
```

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 intercommunicator.

When an inter-communicator is used as an input argument to the communicator accessors described under intra-communication, the following table describes behavior.

MPI_COMM_* Function Behavior (in Inter-Communication Mode)	
MPI_Comm_size	returns the size of the remote group.
MPI_Comm_group	returns the remote group.
MPI_Comm_rank	returns the rank in the local group

5.5.2 Intercommunicator Constructors and Destructors

Intercommunicators can be created using the function MPI_COMM_DUP. The functions MPI_COMM_MAKE and MPI_COMM_SPLIT do not apply to intercommunicators. In addition, the functions MPI_INTERCOMM_START and MPI_INTERCOMM_FINISH can be used to bind two groups into an intercommunicator; the function MPI_INTERCOMM_MERGE creates an intracommunicator by merging the two groups of an intercommunicator.

Construction of an inter-communicator from two intracommunicators requires separate collective operations in the local group and in the remote group, both for the starting and the finishing operations, as well as a point-to-point communication between a process in the local group and a process in the remote group. The explicit synchronization can cause deadlock in modular programs with cyclic communication graphs, even if:

- within each local group calls are executed in the same order,
- the local and remote operations are decoupled and the construction is performed "loosely synchronously" (by calling the two routines MPI_INTERCOMM_START() and MPI_INTERCOMM_FINISH()).

MPI_INTERCOMM_START and MPI_INTERCOMM_FINISH are both collective operations in their local group, but the call within each group need not be synchronized with the call in the other group.

The inter-communicator objects are destroyed in the same way as intra-communicator objects, that is, by calling MPI_COMM_FREE.

The functions MPI_INTERCOMM_START and MPI_INTERCOMM_FINISH can be used to create an intercommunicator from two existing intracommunicators, 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; i.e. a "parent" communicator exists where both leaders belong, and each leader knows the rank of the other leader in this parent communicator (the two leaders could be the same process). Furthermore, members of each group know the rank of their leader.

In static MPI implementation the MPI_COMM_WORLD communicator can be this parent communicator. Since this parent communicator may be shared for multiple intercommunicator creating functions, and the same process may be leaders for several such calls, an additional user-provided tag is used to differentiate calls.

In dynamic MPI implementations, where, e.g., a process may spawn new children 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.

MPI_INTERCOMM_START(local_comm, local_leader, peer_comm, remote_leader, tag, inter_request)

IN	local_comm	local intra-communicator (handle)
IN	local_leader	rank of local group leader in local_comm (integer)
IN	peer_comm	"parent" intra-communicator; significant only at the MPI _arglocal_leader (handle)
IN	remote_leader	rank of remote group leader in peer_comm; significant only at the local_leader (integer)
IN	tag	"safe" tag (integer)
OUT	inter_request	handle for MPI_INTERCOMM_finish()

```
int MPI_Intercomm_start(MPI_Comm local_comm, int local_leader,
```

```
            MPI_Comm peer_comm, int remote_leader, int tag,
            MPI_Intercomm_request *inter_request)
```

```
MPI_INTERCOMM_START(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
            NEW_COMM, INTER_REQUEST, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    INTER_REQUEST, IERROR
```

This call starts off an inter-communicator creation operation, returning a handle for the completion of the operation in handle; it is collective in local_comm. The call does not wait for the remote group to execute MPI_INTERCOMM_start(). inter_request is conceptually similar to the communication requesthandle used by non-blocking point-to-point routines. The handle is constructed by a _start routine and destroyed by the matching _finish routine; these handles are not valid for any other use. It is erroneous to call this routine a second time with the same bridge_comm, remote_leader and tag arguments, without an intervening call to MPI_INTERCOMM_finish() to complete the pending call.

Processes should provide identical comm, local_leader and tag arguments within each group.

MPI_INTERCOMM_FINISH(inter_request, newcomm)

IN	inter_request	handle returned by MPI_INTERCOMM_start()
OUT	newcomm	new inter-communicator (handle)

```
int MPI_Intercomm_finish(MPI_Intercomm_request *inter_request,
            MPI_Comm *newcomm)
```

```
MPI_INTERCOMM_FINISH(INTER_REQUEST, IERROR)
    INTEGER INTER_REQUEST, NEWCOMM, IERROR
```

MPI_INTERCOMM_FINISH completes an asynchronous inter-communicator creation operation, This routine is collective in the goup associated with communicator local_comm of the corresponding call to MPI_INTERCOMM_START. It waits for the remote group to call MPI_INTERCOMM_START but does not wait for the remote group to call MPI_INTERCOMM_FINISH.

Implementation note: MPI_INTERCOMM_START initiates asynchronous sends and receives for an exchange of information between leaders. MPI_INTERCOMM_FINISH completes the exchange of information, next execute a broadcast in each local group, with the local leader at the root.

MPI_INTERCOMM_MERGE(intercomm, key, newintracomm

IN	intercomm	Intercommunicator (handle)
IN	key	Integer key
OUT	newintracomm	new intra-communicator (handle)

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int key, MPI_Comm *newintracomm)
```

```
MPI_INTERCOMM_MERGE(INTERCOMM, KEY, NEWINTERCOMM, IERROR)
```

```
    INTEGER INTERCOMM, KEY, NEWINTRACOMM, IERROR
```

This function returns an intra-communicator for the union of the two groups that are associated with intercomm. All processes should provide the same key value within each of these two groups (there should be exactly two keys values used, one unique value in each group). The process in the group that provided a lower value are ordered before those that provided the higher value. Processes in the intersection may choose either key value. Ties are broken arbitrarily. This call is blocking and collective within the union of the groups.

5.5.3 Name Service

MPI provides a name service to simplify construction of inter-communicators. This service allows a local process group to create an inter-communicator when the only available information about the remote group is a user-defined character string. A loosely synchronous version is provided by routines MPI_INTERCOMM_NAME_START and MPI_INTERCOMM_FINISH.

MPI_INTERCOMM_NAME_START(comm, name, inter_request)

IN	comm	local intra-communicator (handle)
IN	name	character string
OUT	inter_request	handle for MPI_INTERCOMM_FINISH()

```
int MPI_Intercomm_name_start(MPI_Comm comm, char *name,
              MPI_Intercomm_request *inter_request)
```

```
MPI_INTERCOMM_NAME_START(COMM, NAME, NEW_COMM, INTER_REQUEST, IERROR)
    INTEGER COMM, NAME, NEW_COMM, INTER_REQUEST, IERROR
```

This is the name-served equivalent of MPI_INTERCOMM_START. The operation is completed by a call to MPI_INTERCOMM_FINISH.

5.6 Cacheing

MPI provides a "cacheing" facility that allows an application to attach arbitrary pieces of information, called *attributes*, to communicators. More precisely, the cacheing facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI 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.

These capabilities, in some form, are required by built-in MPI routines such as collective communication and virtual topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and virtual topology 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.

5.6.1 Functionality

Attributes are local to the process and specific to the communicator to which they are attached. They 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 for the attribute to be copied). Attributes are scalar values, equal in size or larger than a C-language pointer.

The cacheing interface defined here represents that attributes are 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 group or communicator is destroyed or copied.
- store and retrieve the value of an attribute;

MPI provides the following services related to cacheing. They are all local.

MPI_ATTR_GET_KEY(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_Attr_get_key(copy_fn, delete_fn, int keyval, void* extra_state)
```

```
MPI_ATTR_GET_KEY(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    INTEGER COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR
```

Generates a new attribute key. Keys are locally unique in 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. A C prototype for such a function is

copy_fn(oldcomm, newcomm, keyval, extra_state)

This function does whatever it needs to do with the new communicator newcomm based on what attribute information was in the old communicator olcomm; it returns 1 if successful, 0 if not. 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). MPI_COMM_DUP is the only function that calls the copy_fn. copy_fn may be specified as null, in which case no copy callback occurs for keyval.

Analogous to copy_fn is a callback deletion function, defined as follows:

delete_fn(comm, keyval, extra_state)

This function does whatever is needed to remove the attribute from the communicator specified by comm. It may be specified as null, in which case no delete callback occurs for keyval.

MPI_ATTR_FREE_KEY(keyval)

IN	keyval	Frees the integer key value (integer)

```
int MPI_Attr_free_key(int keyval)
```

```
MPI_ATTR_FREE_KEY(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR
```

Frees an extant attribute key. 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 a process) to the key have been freed.

MPI_ATTR_PUT_VALUE(comm, keyval, attribute_val)

IN	comm	communicator to which attribute will be attached (handle)
IN	keyval	key value, as returned by MPI_ATTR_get_key (integer)
IN	attribute_val	attribute value

```
int MPI_Attr_put_value(MPI_Comm comm, int keyval, void* attribute_val)
```

```
MPI_ATTR_PUT_VALUE(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_VALUE. If the value is already present, a subsequent call to MPI_ATTR_PUT_VALUE replaces the old value, exactly as if a call to MPI_ATTR_DELETE_VALUE(comm, keyval) had first been called (see below).

MPI_ATTR_GET_VALUE(comm, keyval, attribute_val, found)

IN	comm	communicator to which attribute is attached (handle)
IN	keyval	key value (integer)
OUT	attribute_val	attribute value, unless found = false
OUT	found	true or false, depending on whether the attribute value has been put into this attribute set

```
int MPI_Attr_get_value(MPI_Comm comm, int keyval, void* attribute_val,
              int *found)
```

```
MPI_ATTR_GET_VALUE(COMM, KEYVAL, ATTRIBUTE_VAL, FOUND, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, FOUND, IERROR
```

Retrieves attribute value by key.

MPI_ATTR_DELETE_VALUE(comm, keyval)

IN	comm	communicator to which attribute is attached (handle)
IN	keyval	The integer key value to be erased from comm.

```
int MPI_Attr_delete_value(MPI_Comm comm, int keyval)
```

```
MPI_ATTR_DELETE_VALUE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
```

Delete attribute from cache by key. This function invokes the attribute delete function delete_fn specified when the keyval was created.

Whenever a communicator is replicated using the function MPI_COMM_DUP all callback copy functions for attributes that are currently set are invoked. Whenever a communicator is deleted using the function MPI_COMM_FREE all callback delete functions for attributes that are currently set are invoked.

Example

This example shows how to write a collective communication operation that uses cacheing to be more efficient after the first call. The coding style assumes that MPI function results return only error statuses.

```
static int gop_key_assigned = 0;  /* 0 only on first entry */
static int gop_key;               /* key for this module's stuff */

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_assigned)     /* get a key on first call ever */
  {
    gop_key_assigned = 1;
    if ( ! MPI_Attr_get_key( gop_stuff_copier,
                             gop_stuff_destructor,
                             &gop_key, (void *)0));
    /* get the key while assigning its copy and delete callback
       behavior. */
```

```
      MPI_Abort ("Insufficient keys available");
    }

    MPI_Attr_get_value (comm, gop_key, &gop_stuff, &foundflag);
    if (foundflag == MPI_FOUND)
    { /* 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));
      if (gop_stuff == NULL) { /* abort on out-of-memory error */ }

      gop_stuff -> ref_count = 1;

      /* Second, fill in *gop_stuff with whatever we want.
         This part isn't shown here */

      /* Third, store gop_stuff as the attribute value */
      MPI_Attr_put_value ( comm, gop_key, gop_stuff);
    }
    /* Then, in any case, use contents of *gop_stuff
       to do the global op ... */
  }

  /* The following routine is called by MPI when a group is freed */

  gop_stuff_destructor (comm, keyval, gop_stuff, extra)
  MPI_Comm comm;
  int keyval;
  gop_stuff_type *gop_stuff;
  void *extra;
  {
    if (keyval != gop_key) { /* abort -- programming error */ }

    /* The group's being freed removes one reference to gop_stuff */
    gop_stuff -> ref_count -= 1;

    /* If no references remain, then free the storage */
    if (gop_stuff -> ref_count == 0) {
      free((void *)gop_stuff);
```

```
    }
  }

  /* The following routine is called by MPI when a group is copied */
  gop_stuff_copier (comm, keyval, gop_stuff, extra)
  MPI_Comm comm;
  int keyval;
  gop_stuff_type *gop_stuff;
  void *extra;
  {
    if (keyval != gop_key) { /* abort -- programming error */ }

    /* The new group adds one reference to this gop_stuff */
    gop_stuff -> ref_count += 1;
  }
```

5.7 Formalizing the Loosely Synchronous Model (Usage, Safety)

5.7.1 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 (quiescent). 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.

5.7.2 Models of Execution

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.

Non-reentrant parallel procedures

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process. That is, concurrent invocations of the same parallel procedure may occur only within disjoint groups of processes. 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 context can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. Or, it can be done a compile/link time, if the implementation has additional mechanisms to reserve context values. Communicators to be used by the different procedures can be built in a preamble, if the executing groups are statically defined; if the executing groups change dynamically, then a new communicator has to be built whenever the executing group changes, but this new communicator can be built using the same type of preallocated context. 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 context per library.

Parallel procedures that are non-reentrant within each executing group

This covers the case where, at any point in time, for each process group, there can be at most one active invocation of a parallel procedure by a process member. However, it might be possible that the same procedure is concurrently invoked in two partially (or completely) overlapping groups. For example, the same collective communication function may be concurrently invoked on two partially overlapping groups.

In such a case, a context is associated with each parallel procedure and each executing group, so that overlapping execution groups have distinct communication contexts. (One does not need a different context from each group; one merely needs a "coloring" of the groups, so that one can generate the communicators for each parallel procedure when the execution groups are defined. Here again, one only need one context for each library, if no two procedures from the same library can be concurrently active in the same group.

Note that, for collective communication libraries, we do allow several concurrent invocations within the same group: a broadcast in a group may be started in a process before the previous broadcast in that group ended at another process. In such a case, one cannot rely on context mechanisms to disambiguate successive invocations of the same parallel procedure within the same group: the procedure needs to be implemented so as to avoid confusion (back masking). For example, for broadcast, one may need to carry additional information in messages, such as the broadcast root, to help in such disambiguation; one also relies on the pair-wise preservation of message order by MPI. With such an approach, we may be gaining performance, but we lose modularity. It is insufficient to implement the parallel procedure so that it works correctly in isolation, when invoked only once; it needs to be implemented so that any number of successive invocations will execute correctly. Of course, the same approach can be used for other parallel libraries.

Well-nested parallel procedures

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in a subset of a group executing the same parallel procedure. Thus, processes that execute the same parallel procedure have the same execution stack.

In such a case, a new context needs to be dynamically allocated for each new invocation of a parallel procedure. However, a stack mechanism can be used for allocating new contexts. Thus, a possible mechanism is first to allocate a large number of contexts (up to the upper bound on the depth of nested parallel procedure calls), and then use a local stack management of these context's on each process to create a new communicator (using MPI_COMM_MAKE) for each new invocation.

The General case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new context need to be created for each invocation. It is the user responsibility to make sure that, if two distinct parallel procedures are invoked concurrently on overlapping sets of processes, then context allocation or communicator creation is properly coordinated.

5.8 Motivating Examples

5.8.1 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:

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

   MPI_Comm_rank(MPI_COMM_WORLD, &me);   /* local */
   MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */

   if((me % 2) == 0)
      MPI_Send(..., ((me + 1) % size), SOME_TAG, MPI_COMM_WORLD);
   else
      MPI_Recv(..., ((me - 1 + size) % size), SOME_TAG, MPI_COMM_WORLD);
   ...
   MPI_Finalize();
}
```

Example #1b schematically illustrates message exchanges between "even" and "odd" processes in the "all" communicator.

5.8.2 Current Practice #2

```
main(int argc, char **argv)
{
  int me, count;
  void *data;
  ...

  MPI_Init(&argc, argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  if(me == 0)
  {
      /* get input, create buffer ``data'' */
      ...
  }

  MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);

  ...

  MPI_Finalize();
}
```

This example illustrates the use of a collective communication.

5.8.3 (Approximate) Current Practice #3

```
main(int argc, char **argv)
{
  int me, count, count2;
  void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
  MPI_Group MPI_GROUP_WORLD, grp0, 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_incl(MPI_GROUP_WORLD, 1, ranks, &grp0);    /* local */
  MPI_Group_difference(MPI_GROUP_WORLD, grp0, &grprem);/* local */
  MPI_Comm_make(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_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.

5.8.4 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)
{
  int me;
  MPI_Group MPI_GROUP_WORLD, subgroup;
  int ranks[] = {2, 4, 6, 8};
  ...
  MPI_Init(&argc, argv);
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

  MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
  MPI_Group_rank(subgroup, &me);     /* local */

  MPI_Comm_make(MPI_COMM_WORLD, subgroup, &the_comm);

  if(me != MPI_UNDEFINED_RANK)
  {
     /* asynchronous receive: */
     MPI_Irecv(buffer, count, MPI_DOUBLE, MPI_ANY_SOURCE,
               TAG_ARBITRARY, the_comm, request);
  }
```

```
    for(i = 0; i < SOME_COUNT, i++)
      MPI_Reduce(..., the_comm);
    MPI_Finalize();
  }
```

5.8.5 Library Example #1

The main program:

```
  main(int argc, char **argv)
  {
    int done = 0;
    user_lib_t *libh_a, *libh_b;
    void *dataset1, *dataset2;
    ...
    MPI_Init(&argc, argv);
    ...
    init_user_lib(MPI_COMM_WORLD, &libh_a);
    init_user_lib(MPI_COMM_WORLD, &libh_b);
    ...
    user_start_op(libh_a, dataset1);
    user_start_op(libh_a, dataset2);
    ...
    while(!done)
    {
       /* work */
       ...
       MPI_Reduce(..., MPI_COMM_WORLD);
       ...
       /* see if done */
       ...
    }
    user_end_op(libh_a);
    user_end_op(libh_b);
    MPI_Finalize();
  }
```

The user library initialization code:

```
  void init_user_lib(void *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 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);
  }
```

5.8.6 Library Example #2

The main program:

```
  main(int argc, char **argv)
  {
    int ma, mb;
    ...

    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_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_make(MPI_COMM_WORLD, group_a, &comm_a);
    MPI_Comm_make(MPI_COMM_WORLD, group_b, &comm_b);

    MPI_Comm_rank(comm_a, &ma);
```

```
     MPI_Comm_rank(comm_b, &mb);

     if(ma != MPI_UNDEFINED_RANK)
        lib_call(comm_a);

     if(mb != MPI_UNDEFINED_RANK)
     {
       lib_call(comm_b);
       lib_call(comm_b);
     }

     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_Sync(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_Sync 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 "combine" have strong enough source selectivity so that they are inherently okay. So are multiple calls to a typical tree-broadcast algorithm with the same root. However, multiple calls to a typical tree-broadcast algorithm — with different roots — could back-mask, producing undesirable results. Therefore, such algorithms would have to utilize the monotonically increasing tags (with 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.

5.8.7 Inter-Communication Examples

Example 1: Three-Group "Pipeline"

```
+---------+        +---------+        +---------+
|         |        |         |        |         |
| Group 0 | <----> | Group 1 | <----> | Group 2 |
|         |        |         |        |         |
+---------+        +---------+        +---------+
```

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 B only) */
  MPI_Intercomm_request inter_request1, inter_request2;
  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_start( myComm, 0, MPI_COMM_WORLD, 1,
                         1, &inter_request1 );
    MPI_Intercomm_finish( inter_request1, myFirstComm );
  }
  else if (membershipKey == 1)
  {              /* Group 1 communicates with groups 0 and 2. */
```

```
      MPI_Intercomm_start( myComm, 0, MPI_COMM_WORLD, 0,
                           1, &inter_request1 );
      MPI_Intercomm_start( myComm, 0, MPI_COMM_WORLD, 2,
                           12, &inter_request2 );
      MPI_Intercomm_finish( inter_request1, myFirstComm );
      MPI_Intercomm_finish( inter_request2, mySecondComm );
    }
    else if (membershipKey == 2)
    {                     /* Group 2 communicates with group 1. */
      MPI_Intercomm_start( myComm, 0, MPI_COMM_WORLD, 1,
                           12, &inter_request1 );
      MPI_Intercomm_finish( inter_request1, myFirstComm );
    }

    /* Do work ... */

    MPI_Finalize();
  }
```

Example 2: Three-Group "Ring"

```
+-----------------------------------------------------------+
|                                                           |
|    +---------+         +---------+         +---------+    |
|    |         |         |         |         |         |    |
+--> | Group 0 | <-----> | Group 1 | <-----> | Group 2 | <--+
     |         |         |         |         |         |
     +---------+         +---------+         +---------+
```

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

```
  main(int argc, char **argv)
  {
    MPI_Comm  myComm;     /* intra-communicator of local sub-group */
    MPI_Comm myFirstComm; /* inter-communicators */
    MPI_Comm mySecondComm;
    MPI_Intercomm_request inter_request1, inter_request2;
    int membershipKey;
    int rank;

    MPI_Init(&argc, argv);
    MPI_Comm_rank(MPI_COMMM_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_start( myComm, 0, MPI_COMM_WORLD, 1,
                         1, &inter_request1 );
    MPI_Intercomm_start( myComm, 0, MPI_COMM_WORLD, 2,
                         2, &inter_request2 );
  }
  else if (membershipKey == 1)
  {         /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_start( myComm, 0, MPI_COMM_WORLD, 0,
                         1, &inter_request1 );
    MPI_Intercomm_start( myComm, 0, MPI_COMM_WORLD, 2,
                         12, &inter_request2 );
  }
  else if (membershipKey == 2)
  {        /* Group 2 communicates with groups 0 and 1. */
    MPI_Intercomm_start( myComm, 0, MPI_COMM_WORLD, 0,
                         2, &inter_request1 );
    MPI_Intercomm_start( myComm, 0, MPI_COMM_WORLD, 1,
                         12, &inter_request2 );
  }

  /* Everyone has the same "FINISH" code... */
  MPI_Intercomm_finish(inter_request1, &myFirstComm);
  MPI_Intercomm_finish(inter_request2, &mySecondComm);

  /* Do some work ... */

  MPI_Finalize();
}
```

Example 3: Three-Group "Pipeline" Using Name Service

```
+---------+        +---------+        +---------+
|         |        |         |        |         |
| Group 0 | <-----> | Group 1 | <-----> | Group 2 |
|         |        |         |        |         |
+---------+        +---------+        +---------+
```

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 B only) */
    MPI_Intercomm_request inter_request1, inter_request2;
    int color, rank, size;

    MPI_Init(&argc, argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    ...

    /* User code must generate color in the range [0, 1, 2] */
    if ( rank < ((size+1)/3) )
      color = 0;
    else if ( rank < (2*(size+1)/3) )
      color = 1;
    else
      color = 2;

    /* Build intra-communicator for local sub-group */
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &myComm);

    /* Build inter-communicators.  Group membership conditions must be */
    /* provided by the user. */
    if (color == 0)
    {       /* Group 0 communicates with group 1. */
      MPI_Intercomm_name_start(myComm, "Connect 01", &inter_request1);
      MPI_Intercomm_finish (inter_request1, &myFirstComm);
    }
    else if (color == 1)
    {       /* Group 1 communicates with groups 0 and 2. */
      MPI_Intercomm_name_start(myComm, "Connect 10", &inter_request1);
      MPI_Intercomm_name_start(myComm, "Connect 12", &inter_request2);
      MPI_Intercomm_finish (inter_request1, &myFirstComm);
      MPI_Intercomm_finish (inter_request2, &mySecondComm);
    }
    else if (color == 2)
    {       /* Group 2 communicates with group 1. */
      MPI_Intercomm_name_start(myComm, "Connect 21", &inter_request1);
      MPI_Intercomm_finish (inter_request1, &myFirstComm);
    }

    /* Do some work ... */

    MPI_Finalize();
  }
```

Example 4: Three-Group "Ring" Using Name Service

```
+-----------------------------------------------------------+
|                                                           |
|    +---------+          +---------+          +---------+    |
|    |         |          |         |          |         |    |
+--> | Group 0 | <-----> | Group 1 | <-----> | Group 2 | <--+
     |         |          |         |          |         |
     +---------+          +---------+          +---------+
```

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

```
main(int argc, char **argv)
{
  MPI_Comm  myComm;   /* intra-communicator of local sub-group */
  MPI_Comm  myFirstComm;               /* inter-communicators */
  MPI_Comm  mySecondComm;
  MPI_Intercomm_request inter_request1, inter_request2;
  int color, rank, size;

  MPI_Init(&argc, argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...

  /* User code must generate color in the range [0, 1, 2] */
  if ( rank == 0 )
    color = 0;
  else if ( rank == 1 )
    color = 1;
  else
    color = 2;

  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, color, rank, &myComm);

  /* Build inter-communicators.  Group membership conditions must be */
  /* provided by the user. */
  if (color == 0)
  {     /* Group 0 communicates with groups 1 and 2. */
    MPI_Intercomm_name_start(myComm, "Connect 01", &inter_request1);
    MPI_Intercomm_name_start(myComm, "Connect 02", &inter_request2);
  }
  else if (color == 1)
  {     /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_name_start(myComm, "Connect 10", &inter_request1);
    MPI_Intercomm_name_start(myComm, "Connect 12", &inter_request2);
  }
```

```
    else if (<I am in group 2>)
    {    /* Group 2 communicates with groups 0 and 1. */
      MPI_Intercomm_name_start(myComm, "Connect 20", &inter_request1);
      MPI_Intercomm_name_start(myComm, "Connect 21", &inter_request2);
    }

    /* Everyone has the same "FINISH" code... */
    MPI_Intercomm_finish (inter_request1, &myFirstComm);
    MPI_Intercomm_finish (inter_request2, &mySecondComm);

    /* Do some work ... */

    MPI_Finalize();
  }
```

Chapter 6

Process Topologies

6.1 Introduction

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 process ordering does not reflect the 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 as simple as a two– or three–dimensional grid. In the more general case 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 physical hardware processors. The application-specific, 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 in detail, however, is outside the scope of MPI. The description of the virtual process topology, on the other hand, only depends on the application, and is, therefore, machine–independent. The functions which are proposed in this paper only deal with this side of the mapping problem.

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 near–optimal results [16]. 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 the consequence. On some machines this will lead to unnecessary contention in the interconnect network. Therefore, though we don't discuss the physical mapping, the mere existence of the virtual topology information may be used as advice by the runtime system. Some details about predicted and measured performance improvements which result from good process–to–processor mapping on modern wormhole–routing architectures can be found in [8, 7].

6.2 Virtual Topologies

In the most general case the communication pattern of a set of processes is be represented by a graph. The nodes stand for the processes, and the edges connect processes which communicate with each other. MPI provides message–passing between any pair of processes in

[1]Version as of Oct. 16, 1993

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 only means that this connection is neglected in the virtual topology. This 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 try to assign the processes to neighboring processors. Edges in the communication graph are not weighted. Processes are either neighbors or not. Experience with similar techniques in PARMACS [4, 6] show that this information is usually sufficient for a good mapping. Additionally, a more precise specification would be more difficult for the user to set up, and it would make the interface functions more complicated.

In principle, a function for the definition of such a graph structure would be sufficient for all applications. However, in many applications the graph structure is very regular, and the detailed set–up of the graph would be inconvenient for the user and resource–consuming during 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 then that of general graphs. Thus, it is desirable to separate out this case explicitly.

Process coordinates in a Cartesian structure begin their numbering at 0. Row major numbering is 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 which are defined elsewhere. Topology information is associated with communicators. It can maintained using the caching mechanism described in Chapter 5. The topology functions complement and interoperate with the collective communication functions described in Chapter 4.

The current design of the process topology functions assumes that MPI is free to choose any process in a group for any location in the topology. There are no special processes which have to be aligned with topology substructures. This assumption fits best to a "local SPMD model" in which all processes in the same group execute the same program.

6.4 Overview of the proposed MPI functions

The functions MPI_MAKE_GRAPH and MPI_MAKE_CART are used to create general virtual topologies and Cartesian topologies, respectively. These topology creation functions are collective. As for other collective calls, the code should be written to work correctly, whether the call synchronizes or not.

The topology definition function MPI_MAKE_CART can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction it is specified whether the process structure is periodic or not. Similar functions are contained in EXPRESS [17] and PARMACS. 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_MAKE_DIMS can be used to compute an "even" distribution of processes along a given number of dimensions.

The function MPI_TOPO_STATUS can be used to inquire on the topology associated with a communicator. The topological information can be extracted from the communicator using the functions MPI_GET_GRAPHDIMS and MPI_GET_GRAPH, for general graphs, and MPI_GET_CARTDIM and MPI_GET_CART, for Cartesian toplogies. Several additional functions are provided to manipulate Cartesian topologies: the functions MPI_CART_RANK and MPI_CART_COORD translate Cartesian coordinates into a group rank, and vice-versa; the function MPI_CART_SUB can be used to extract a Cartesian subspace; and the function MPI_CART_SHIFT provides the information needed to execute a shift in a Cartesian dimension. The two functions MPI_GRAPH_NNEIGBORS and MPI_GRAPH_NEIGHBORS can be used to extract the neighbors of a node in a graph. The function MPI_CART_SUB is collective; all other functions are local.

Two additional functions, MPI_MAP_GRAPH and MPI_MAP_CART are presented in the last section. These functions are not part of the MPI definition. However, these two functions, together with the communicator manipulation functions presented in Chapter 5, are sufficient to implement all other topology functions. The last section outlines such implementation.

6.5 Topology Constructors

MPI_MAKE_CART(comm_old, ndims, dims, periods, reorder, comm_cart)

IN	comm_old	communicator for parent group (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 aperiodic (false) in each dimension
IN	reorder	ranking may be reordered (true) or not (false) (logical)
OUT	comm_cart	communicator for new Cartesian group (handle)

The function 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, 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 the processes that are not member of the grid are returned a null handle. The call is erroneous if it specifies a grid that is larger than the the group size.

In a portable program the number of available processors is usually not known at compile time. In the case of a Cartesian topology the function MPI_MAKE_DIMS helps in setting the numbers of processes per coordinate direction, depending on the number of processes in the group and optional constraints which can be specified by the user:

MPI_MAKE_DIMS(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

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. 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 the corresponding dimension; only those entries where `dims[i] = 0` will be modified by the call.

Note that an error will occur if `nnodes` is not a multiple of $\prod_{i, dims[i] \neq 0} dims[i]$.

Negative input values of `dims[i]` will also cause an error. All elements of `dims` must be set to 0 for completely automatic process distribution.

For `dims[i]` set by the call, `dims[i]` will be ordered in nonincreasing order. Array `dims` is suitable for use as input to routine MPI_MAKE_CART.

Examples:

Assume that `dims` = (0,0). Then
MPI_MAKE_DIM(6, 2, dims) returns (3,2);
MPI_MAKE_DIM(7, 2, dims) returns (7,1).

Assume that `dims` = (0,3,0). Then
MPI_MAKE_DIM(6, 3, dims) returns (2,3,1);
MPI_MAKE_DIM(7, 3, dims) is erroneous.

The function MPI_MAKE_DIMS is local.

For general graph structure, the constructor is

MPI_MAKE_GRAPH(comm_old, nnodes, index, edges, reorder, comm_graph)

IN	comm_old	communicator for parent group (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 for new graph group (handle)

The function 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, so as to choose a good embedding of the virtual topology onto the physical machine. If the size nnodes of the graph is smaller than the size of the group of comm, then the processes that are not member of the grid are returned a null handle. The call is erroneous if it specifies a graph that is larger than then the group size.

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 definitions of the arguments nnodes, index, and edges are illustrated with the following simple example:

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], $0 \leq j \leq \mathrm{index}[0] - 1$ and the list of neighbors of node i, $i > 0$, is stored in edges[j], $\mathrm{index}[i-1] \leq j \leq \mathrm{index}[i] - 1$.

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], $1 \leq j \leq \mathrm{index}[0]$ and the list of neighbors of node i, $i > 0$, is stored in edges[j], $\mathrm{index}[i-1] + 1 \leq j \leq \mathrm{index}[i]$.

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.

Implementation note: The following topology information is likely to be stored with a communicator:

1. Type of topology (Cartesian/graph)
2. For a Cartesian topology:
   - ndims (number of dimensions),
   - dims (numbers of processes per coordinate direction),
   - periods (periodicity information),
   - own_position (own position in grid, could also be computed from rank and dims)
3. For a graph topology:
   - index, edges (vectors defining graph structure)

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, it does not have to be stored explicitely. An additional zero entry at the start of array index will simplify access to the topology information.

6.5.1 Topology inquiry functions

If a topology has been defined with one of the above functions, the topology information can be looked up using inquiry functions. They all are local calls, and do not require any communication.

The function MPI_TOPO_STATUS returns the type of topology which is assigned to a communicator.

MPI_TOPO_STATUS(comm, status)

IN	comm	communicator (handle)
OUT	status	topology type of communicator comm (choice)

The output value status is one of the following:

MPI_GRAPH	graph topology
MPI_CART	Cartesian topology
MPI_UNDEFINED	no topology

Functions MPI_GET_GRAPHDIMS and MPI_GET_GRAPH retrieve the graph topology information that was associated with a communicator by MPI_MAKE_GRAPH.

MPI_GET_GRAPHDIMS(comm, nnodes, nedges)

IN	comm	communicator for group with graph structure (handle)
OUT	nnodes	number of nodes in graph (integer)
OUT	nedges	number of edges in graph (integer)

The information provided by MPI_GET_GRAPHDIMS can be used to dimension the vectors index and edges long enough in the following call to MPI_GET_GRAPH.

MPI_GET_GRAPH(comm, maxindex, maxedges, index, edges)

IN	comm	communicator for group with graph structure (handle)
IN	maxindex	length of vector index in the calling program (integer)
IN	maxedges	length of vector edges in the calling program (integer)
OUT	index, edges	vectors containing the graph structure (for details see the definition of MPI_MAKE_GRAPH)

The functions MPI_GET_CARTDIM and MPI_GET_CART return the cartesian topology information that was associated with a communicator by the function MPI_MAKE_CART.

MPI_GET_CARTDIM(comm, ndims)

IN	comm	communicator for group with Cartesian structure (handle)
OUT	ndims	number of dimensions of the Cartesian structure (integer)

MPI_GET_CART(comm, maxdims, dims, period, coord)

IN	comm	communicator for group with Cartesian structure (handle)
IN	maxdims	length of vectors `size, period`, and `coord` in the calling program (integer)
OUT	dims	number of processes for each Cartesian dimension (array of integer)
OUT	periods	periodicity (true/false) for each Cartesian dimension (array of logical)
OUT	coord	coordinates of calling process in Cartesian structure (array of integer)

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:

MPI_CART_RANK(comm, coord, rank)

IN	comm	communicator for group with Cartesian structure (handle)
IN	coord	integer array (of size `ndims`) specifying the Cartesian coordinates of a process
OUT	rank	rank of specified process (integer)

The inverse rank-to-coordinates translation is provided by the function:

MPI_CART_COORD(comm, rank, maxdims, coord)

IN	comm	communicator for group with Cartesian structure (handle)
IN	rank	rank of a process within group of comm (integer)
IN	maxdims	length of vector `coord` in the calling program (integer)
OUT	coord	integer array (of size `ndims`) containing the Cartesian coordinates of specified process (integer)

The following functions provide "adjacency" information for a general graph topology.

MPI_GRAPH_NNEIGHBORS(comm, rank, nneighbors)

IN	comm	communicator for a group with graph topology
IN	rank	rank of process in group (integer)
OUT	nneighbors	number of neighbors of specified process (integer)

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

IN	comm	communicator for a group with graph topology
IN	rank	rank of process in group (integer)
OUT	maxneighbors	size of array neighbors
OUT	neighbors	ranks of processes that are neighbors to specified process (array of integer)

Example: Suppose that comm is a communicator for a group with a shuffle-exchange topology: The group has 2^n members. Each process $a_1, \ldots, a_n$ ($a_i \in \{0,1\}$) has three neighbors: $exchange(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ ($\bar{a} = 1 - a$), $shuffle(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$, and $unshuffle(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for $n = 3$

node		exchange	shuffle	unshuffle
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

Each process stores 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)
```

6.5.2 Cartesian shift coordinates

If the process topology is a Cartesian structure, a collective shift operation is most likely to be used along a coordinate direction. In MPI shift operations are handled by the more general send–receive functions. As input they take 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 the send–receive function. The user specifies the coordinate direction for the shift and the step length. The function is local.

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

IN	comm	communicator for group with Cartesian structure (handle)
IN	direction	coordinate dimension of shift (integer)
IN	disp	shift amount (> 0: upwards shift, < 0: downwards shift) (integer)
OUT	rank_source	rank of source process (integer)
OUT	rank_dest	rank of destination process (integer)

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_PROCNULL may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.

Example: The communicator comm is associated with a two-dimensional periodic Cartesian topology. 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 by i.

```
....
C find process rank
      CALL MPI_COMM_RANK(comm, rank, ierr, ierr))
C find Cartesian coordinates
      CALL MPI_CART_COORD(comm, rank, coord, ierr)
C compute shift source and destination
      CALL MPI_CART_SHIFT(comm, 2, coord(2), source, dest, ierr)
C skew array
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
                                status, ierr)
```

6.5.3 Partitioning of Cartesian structures

If a Cartesian topology has been created with MPI_MAKE_CART, the function MPI_CART_SUB can be used to partition the communicator group into subgroups which form lower–dimensional Cartesian subsgrids, and built for each subgroup a communicator with the associated subgrid Cartesian topology.

MPI_CART_SUB(comm, remain_dims, comm_new)

IN	comm	communicator for group with Cartesian structure (handle)
IN	remain_dims	the `i`-th entry of remain_dims specifies whether the `i`-th dimnsion is kept in the subgrid (`true`) or is dropped (`false`) (logical vector)
OUT	comm_new	communicator containing the subgrid which includes the calling process (handle)

Example:

Assume that MPI_MAKE_ CART(..., comm) has defined a $2\times3\times4$ grid. Let `remain_dims = (true, false, true)`. Then a call to MPI_SUBPART(comm, remain_dims, comm_new) will create communicators for three new groups with eight processes, each with a 2×4 Cartesian topology. If `remain_dims = (false, false, true)` then the call to MPI_SUBPART(comm, remain_dims, comm_new) will create communicators for six new groups with four processes, each with a one-dimensional Cartesian topology.

6.5.4 Low–level topology functions

The two new functions introduced in this section are not part of the MPI definition. However, they can be used to implement all other topology functions.

MPI_MAP_CART(comm_old, ndims, dims, period, newrank)

IN	comm_old	communicator for parent group (handle)
IN	ndims	number of dimensions of Cartesian structure (integer)
IN	dims	integer array of size `ndims` specifying the number of processes in each coordinate direction
IN	periods	logical array of size `ndims` specifying the periodicity specification in each coordinate direction
OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)

This function 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 – i.e., not to perform any reordering.

The function MPI_MAKE_CART(comm_old, ndims, dims, periods, reorder, comm_cart), with `reorder = true` can be implemented by calling MPI_MAP_CART(comm_old, ndims, dims, period, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_cart), with `color = 0` if `newrank` $\neq$ `MPI_UNDEFINED`, `color = MPI_UNDEFINED` otherwise, and `key = newrank`.

The function MPI_SUBCART(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.

The corresponding new function for general graph structures is

MPI_MAP_GRAPH(comm_old, nnodes, index, edges, newrank)

IN	comm_old	communicator for parent group (handle)
IN	nnodes	number of graph nodes (integer)
IN	index, edges	integer arrays specifying the graph structure, see MPI_MAKE_GRAPH
OUT	newrank	reordered rank of the calling process; MPI_UNDEFINED if the calling process does not belong to graph (integer)

The function MPI_MAKE_GRAPH(comm_old, nnodes, index, edges, reorder, comm_cart), with `reorder = true` can be implemented by calling MPI_MAP_GRAPH(comm_old, nnodes, index, edges, newrank) then calling MPI_COMM_SPLIT(comm, color, key, comm_cart), with `color = 0` if `newrank` $\neq$ `MPI_UNDEFINED`, `color = MPI_UNDEFINED` otherwise, and `key = newrank`.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator.

6.6 A simple example

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 example 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 for the ranks of its neighbors in the four directions. 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 interprocess 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). Note that there is only local communication in this exchange operation.

```
      integer ndims, num_neigh
      logical reorder
      parameter (ndims=2, num_neigh=4, reorder=.true.)
      integer comm, comm_cart, dims(ndims), neigh_def(2), neigh_rank(num_neigh)
      integer own_position(ndims), i, j, rank
      logical period(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_MAKDIM(comm, ndims, dims)
      period(1) = .TRUE.
      period(2) = .TRUE.
C     Create a grid structure in WORLD group and inquire about own position
      call MPI_MAKE_CART (comm, ndims, dims, period, reorder, rank, comm_cart)
      call MPI_GAT_CART (comm_cart, ndims, dims, period, own_position)
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))
      neigh_def(1) = i+1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2))
      neigh_def(1) = i
      neigh_def(2) = j-1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3))
      neigh_def(1) = i
      neigh_def(2) = j+1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4))
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 and communication buffering). The procedures for entering and leaving the MPI execution environment are also dexcribed here.

7.1 Implementation information

7.1.1 Environmental Inquiries

The following routines allow to inquire on the execution environment of an MPI program.

MPI_GET_VALID_TAG_RANGE (low, high)

OUT	low	Smallest valid tag value
OUT	high	Largest valid tag value

```
int MPI_GET_VALID_TAG_RANGE(int *low, int *high)
```

```
MPI_GET_VALID_TAG_RANGE(LOW, HIGH, IERROR)
    INTEGER LOW, HIGH, IERROR
```

This routine gets the range of valid tags; that is, the user may use any tag value from `low` to `high` inclusive.

MPI_GET_PROCESSOR_NAME(len, name)

IN	len	Length (in characters) of the space in `name`
OUT	name	A unique specifier for the actual (as opposed to virtual) node.

```
int MPI_GET_PROCESSOR_NAME(int len, char *name)
```

```
MPI_GET_PROCESSOR_NAME(LEN, NAME, IERROR)
    INTEGER LEN, NAME, IERROR
```

[1]Version as of October 14, 1993

The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). A error code of MPI_UNDEFINED is allowed but strongly discouraged. The value of MPI_MAX_PROCESSOR_NAME is the maximum length required by MPI_GET_PROCESSOR_NAME (including, in the C binding, the terminating null).

MPI_GET_HOST(comm, rank)

IN	comm	communicator (handle)
OUT	rank	rank of HOST in group of comm; MPI_PROCNULL if there is no HOST

```
int MPI_GET_HOST(MPI_Comm comm, int *rank)
```

```
MPI_GET_HOST(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR
```

This routines gets the rank of the HOST process in the group associated with communicator comm, if there is such. MPI does not specify what it means for a process to be a HOST, nor does it requires that a HOST exists.

7.1.2 Buffering

While it is possible to design algorithms that do not require that any buffering be provided for sending and receiving "blocking" messages, many practitioners expect that the message-passing system will provide some buffering. That is, there is often an expectation that the program fragment, executed by all tasks,

```
MPI_send( buf, size, datatype, nbr, tag, comm )
MPI_recv( buf, size, datatype, nbr, tag, comm )
```

will execute for messages whose size is not too large. The purpose of this section is to give the user a way to determine what "too large" is and to help the implementation provide enough buffering for the user's need.

Note that nothing in this section *requires* an implementation to provide any buffering; that is, an implementation may define the above program fragment to be erroneous for any positive value of size.

General buffering parameters

MPI_GET_BUFFER_PARAMS(numsend, sizesend, numrecv, sizerecv, totalnumsednd, totalsizesen, totalnumrecv, totalsizerecv)

OUT	numsend	Bound on the number of pending messages with source at calling process and same destination
OUT	sizesend	Bound on the total size in bytes of pending messages with source at calling process and same destination
OUT	numrecv	Bound on the number of pending messages with destination at calling process and same source
OUT	sizerecv	Bound on the total size in bytes of pending messages with destination at calling process and same source
OUT	totalnumsend	Bound on the number of pending messages with source at the calling process and any destination
OUT	totalsizesend	Bound on the total size in bytes of pending messages with source at the calling process and any destination
OUT	totalnumrecv	Bound on the number of pending messages from any source with the calling process as destination
OUT	totalsizerecv	Bound on the total size in bytes of pending messages from any source with destination at the calling node

```
int MPI_GET_BUFFER_PARAMS(int *numsend, int *sizesend, int *numrecv,
             int *sizerecv, int *totalnumsednd, int *totalsizesen,
             int *totalnumrecv, int *totalsizerecv)
```

```
MPI_GET_BUFFER_PARAMS(NUMSEND, SIZESEND, NUMRECV, SIZERECV, TOTALNUMSEDND,
             TOTALSIZESEN, TOTALNUMRECV, TOTALSIZERECV, IERROR)
    INTEGER NUMSEND, SIZESEND, NUMRECV, SIZERECV, TOTALNUMSEDND,
    TOTALSIZESEN, TOTALNUMRECV, TOTALSIZERECV, IERROR
```

A program where at any point in time no process has more then numsend pending messages sent by it to the same destination, no more than sizesend bytes of data in pending messages sent by it to the same destination, no more than totalnumsend pending messages sent by it to any destination, no more than totalsizesend bytes of data in pending messages sent by it to any destination, no more than numrecv pending messages sent to it from the same source, no more than sizerecv bytes in pending messages sent to it from the same source, no more than totalnumrecv pending messages sent to it from any source, and no more than totalsizerecv bytes in pending messages sent to it from any source is guaranteed no to deadlock because of the lack of buffer space for pending messages.

Examples:

1. Suppose that pending messages are always buffered at the sender, all buffers are shared, total buffer size at the node is `B`, each pending message takes `h` bytes of storage, in addition to the storage for its data, and no storage is lost due to fragmentation. Then possible return values for MPI_GET_BUFFER_PARAMS are `numsend = totalnumsend-k, sizesend = totalsizesend = B-kh`, and `numrecv = totalnumrecv = sizerecv = totalsizerecv = INT_MAX`, for some $0 < k < B/h$.

2. Suppose that messages are always buffered at the receiver, and that a separate buffer with `C` bytes is allocated for each pair of communicating processes; entries for pending messages are kept in a seeparate table, with room for total of `m` pending messages at each receiving process; the total number of processes is `p`, and no storage is lost due to fragmentation. Then possible return values for MPI_GET_BUFFER_PARAMS are `sendcount = totalsendcount = sendsize = totalsendsize = INT_MAX`, `recvcount = totalrecvcount = m`, `recsize = C` and `totalrecvsize = (p-1)`$\times$`C`.

The call may return the value MPI_UNDEFINED in any parameter. If this is the case, then the MPI implementation makes no guarantee about buffering.

Some implementations may be able to adjust their buffering. In addition, since the parameters returned by MPI_GET_BUFFER_PARAMS represent a guarantee about a particular choice of number of buffers, it may not guarantee buffering that it can in fact perform. Thus, in example 1 above, the implementation picks an arbitrary value for `k`, trading off number of messages for total message size, rather than letting the user make this tradeoff. In order to allow the MPI user to inquire about a specific buffering need, the routine MPI_SUGGEST_BUFFER_PARAMS is provided.

MPI_SUGGEST_BUFFER_PARAMS(numsend, sizesend, numrecv, sizerecv, totalnumsend, totalsizesen, totalnumrecv, totalsizerecv, flag)

IN	numsend	Bound on number of pending messages with source at calling process and same destination
IN	sizesend	Bound on total size in bytes of pending messages with source at calling node and same destination
IN	numrecv	Bound on the number of pending messages with destination and calling process and same source
IN	sizerecv	Bound on total size in bytes of pending messages with destination at calling process and same source
IN	totalnumsend	Bound on the number of pending messages with source at the calling process and any destination
IN	totalsizesend	Bound on total size in bytes of pending messages with source at the calling process and any destination
IN	totalnumrecv	Bound on the number of pending messages from any source with the calling process as destination
IN	totalsizerecv	Bound on the total size in bytes of pending messages from any source with destination at the calling node
OUT	flag	TRUE if the suggestion was accepted, FALSE, otherwise

```
int MPI_SUGGEST_BUFFER_PARAMS(int numsend, int sizesend, int numrecv,
              int sizerecv, int totalnumsend, int totalsizesen,
              int totalnumrecv, int totalsizerecv, int *flag)
```

```
MPI_SUGGEST_BUFFER_PARAMS(NUMSEND, SIZESEND, NUMRECV, SIZERECV,
              TOTALNUMSEND, TOTALSIZESEN, TOTALNUMRECV, TOTALSIZERECV, FLAG,
              IERROR)
```

```
    INTEGER NUMSEND, SIZESEND, NUMRECV, SIZERECV, TOTALNUMSEND,
    TOTALSIZESEN, TOTALNUMRECV, TOTALSIZERECV, FLAG, IERROR
```

If a value of true is returned in const then it is as if the values provided by the user in MPI_SUGGEST_BUFFER_PARAMS were returned by a call to MPI_GET_BUFFER_PARAMS: the user is guaranteed that no deadlock will occur due to lack of buffers, provided it never exceeds the limitations on number and size of pending messages implied by these parameters.

This routines allows the user to tell the MPI implementation that a certain amount of buffering would be desireable, The MPI implementation is not required to provide such buffering.

It is expected that a common use of these routines will be

```
MPI_Get_buffer_params( &sendnum, &sendsize, ... );
if (sendsize < buffer_needed) {
    fprintf( stderr, "Program requires more buffering than is available\n" );
    MPI_Abort( 1 );
    }
else
    exploit buffering
```

This allows a program to fail rather than deadlock. It is also possible for an algorithm to switch to a different algorithm that requires less or no buffering.

Another use is to inform the MPI implementation of the buffer space desires of the application:

```
MPI_suggest_buffer_params( 1, buffer_needed, 1, buffer_needed, 0, 0, 0, 0  &flag );
if (flag) {
    exploit buffering
    }
else {
    fprintf( stderr, "Program requires more buffering than is available\n" );
    MPI_abort( 1 );
    }
```

Application provided buffering

An application program can optionally provide space for MPI's use in buffering message data via the following call:

MPI_USER_SPECIFIES_BUFFER(comm, buffer, size)

IN	comm	is a communicator with which the buffer is to be associated (handle)
IN	buffer	initial buffer address (choice)
IN	size	buffer size, in bytes (integer)

```
int MPI_USER_SPECIFIES_BUFFER(MPI_Comm comm, void* buffer, int size)
```

```
MPI_USER_SPECIFIES_BUFFER(COMM, BUFFER, SIZE, IERROR)
    <type> SENDBUF(*)
```

```
INTEGER COMM, BUFFER, SIZE, IERROR
```

If MPI_USER_SPECIFIES_BUFFER is called, then for subsequent regular sends using the specified communicator, MPI must provide as much safety *as if* outgoing message data were buffered by the sending process, in the provided buffer space, using a circular contiguous-space allocation policy. Following the call to MPI_USER_SPECIFIES_BUFFER, the application is not permitted to access the buffer until the corresponding communicator has been freed.

Access to the provided buffer space is not inherited by communicators derived from the initially specified one.

Implementation note: One approach is for MPI to simply transform all blocking sends on the specified communicator into

- copy data to (application-provided) buffer space
- issue non-blocking send from the buffer copy
- return to application
- check completion on subsequent MPI call(s)

No doubt many optimizations within MPI are possible—the proposal just requires that MPI provide at least this much safety.

Note that collective communications and other nonlocal operations will share suggested or user-supplied communication buffers with point to point communication.

7.2 Error handling

This section contains the suggested POSIX binding for error handling in MPI. MPI implementations that do not run in a POSIX-compliant environment may need to use alternative mechanisms.

An MPI implementation can not or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are not handled by MPI is implementation-dependent.

There are three possible error handling modes for those errors that MPI handles, which are identified by three constant values:

MPI_ERRORSFATAL The error is *fatal* and causes the program to abort.

MPI_ERRORSRETURN The error causes the MPI routine where it occurred to return an error code.

MPI_ERRORSCALLUSERROUTINE The error causes MPI to call a user-specified error handling routine. Such handler can be specified by a call to MPI_SET_HANDLER.

A user can associate an error handling mode with a communicator. The specified error handling mode will be used for any error that occurs during a call to MPI for a communication with this communicator. MPI calls that are not related to any communicator are considered to be attached to the communicator MPI_WORLD.

A user may also attach an error handling routine with a communicator; This attachement is purely local: different processes may attach different error handlers to the same communicator.

A newly created communicator inherits the error mode and the error handling routine, if any, that is associated with the "parent" communicator. In particular, the user can specify a "global" error-handling mode for all communicators by associating this mode with the communicator MPI_COMM_WORLD immediatly after initialization.

The default error handling mode for MPI_COMM_WORLD is MPI_ERRORSFATAL.

MPI_GET_ERRORMODE (comm, mode)

IN	comm	communicator to get error mode from (handle)
OUT	mode	current error handling mode (state)

```
int MPI_GET_ERRORMODE(MPI_Comm comm, MPI_Errmode mode)
```

```
MPI_GET_ERRORMODE(COMM, MODE, IERROR)
    INTEGER COMM, MODE, IERROR
```

Gets the error handling mode.

MPI_SET_ERRORMODE (comm, mode)

IN	comm	communicator to set error mode for (handle)
IN	mode	new error mode (state)

```
int MPI_SET_ERRORMODE(MPI_Comm comm, MPI_Errmode mode)
```

```
MPI_SET_ERRORMODE(COMM, MODE, IERROR)
    INTEGER COMM, MODE, IERROR
```

Sets the error handling mode.

MPI_SET_ERRHANDLER(comm, handler)

IN	comm	communicator to set the error handler for (handle)
IN	handler	user-defined error handler to call

```
int MPI_SET_ERRHANDLER(MPI_Comm comm, MPI_Errhandler handler)
```

```
MPI_SET_ERRHANDLER(COMM, HANDLER, IERROR)
    INTEGER COMM, HANDLER, IERROR
```

This routine returns the previous error handler as its value.

An error handler is a C function declared as

```
typedef int (MPI_ERRHANDLER)(MPICOMM, int *, ...);
```

The first argument is the communicator in use. The second is a pointer to the error code to be returned by the MPI routine. The remaining arguments are "varargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these parameters.

There are two predefined error handlers that the user may use instead of writing their own:

MPI_DEFAULTHANDLER aborts from within MPI.

MPI_NOHANDLER returns the error value and has no further effect.

MPI_ERROR_STRING(errorcode, len, string)

IN	errorcode	Error code returned by an MPI routine
IN	len	Length of string
OUT	string	Text that corresponds to the errorcode

```
int MPI_ERROR_STRING(int errorcode, int len, char *string)
```

```
MPI_ERROR_STRING(ERRORCODE, LEN, STRING, IERROR)
    INTEGER ERRORCODE, LEN, STRING, IERROR
```

7.3 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 routines are called. The version for ANSI C accepts the argc and argv that are provided by the arguments to main:

```
MPI_init( argc, argv );
```

The Fortran version takes no arguments.

MPI_FINALIZE()

```
int MPI_FINALIZE()
```

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

This routines cleans all MPI state. Once this routine is called, no MPI routine (even MPI_INIT) may be called.

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)
    INTEGER FLAG, 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.

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 accessable 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.

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

[1]Version as of July 30, 1993

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.

8.4 Examples

8.4.1 Profiler implementation

Suppose that the profiler wishes to accumulate the total number of times the MPI_SEND function is called, along with the total elapsed time spent in the function. This could trivially be achieved as follows.

```
static int totalSendCount;
static int totalTime;
extern int now(void);   /* Some suitable timer function ! */

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
{
   int tstart = now();
   /* Now call the real send, passing along all the arguments */
   int result = PMPI_Send(buf, count, datatype, dest, tag, comm);

   totalSendCount++;                   /* Accumulate byte count */
   totalTime  += now() - tstart;       /* and time              */

   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.

8.5 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 co-operation between the different profiling libraries than is required for the single level implementation detailed above.

Chapter 9

Initial Implementation Subset

9.1 Introduction

This chapter defines the minimal subset of MPI for initial implementation. This subset is being defined so that consistent implementations can appear more rapidly. It was recognized early in the process that MPI needed to appear as quickly as possible and practical. The creation of a subset will hopefully allow users earlier access to the standard and still allow for the writing of portable message passing codes.

This subset should not in any way be construed as a limitation on MPI implementations. It is strictly the minimum necessary to have an initial MPI subset conforming implementation. It is hoped that an officially sanctioned subset will encourage and allow implementors to introduce MPI in a more timely fashion. It should be noted, however, that implementation of the subset does not make an implementation conform with MPI. The subset is a potential first step in the process. All implementations shall conform to the entire standard; implementations are encouraged to do the full standard as rapidly as possible.

The subset presented is consistent with the complete MPI standard. This was an important goal so the additional MPI features could be added without changing any previous functionality from the user's perspective. Thus, users can be assured that programs written now for the subset will run without modification under the full MPI standard. Furthermore, using additional features of the full MPI standard in the future will not require changes to code already written. This requires that the MPI functions have their full calling arguments in the subset. To achieve this goal, the subset will generally limit access to full MPI functionality through complete elimination of functions or limitations on the creation mechanisms for certain MPI input arguments.

9.2 Criteria and Rationale

Having the subset be consistent with the entire MPI standard was considered critical to the effort. In addition to this, there were many possible criteria to use in determining the elements of MPI to include in the subset. The main criteria used were that the MPI subset should

[1] Version of Oct 3, 1993

1. contain routines that are as close as possible to current standard practice to minimize the effort to port codes.

2. contain new and important MPI features.

3. allow developers to be able to have the subset show up as rapidly as possible while still meeting the other criteria.

There were many rationales for these criteria. It was recognized that it is impossible to come up with an ideal subset, and compromises were necessary. It was felt that current users should be comfortable in migrating to MPI. Thus, the subset should contain the message passing elements that represent common practice today. Furthermore, it was felt that the initial (and later) version of MPI should maintain efficiency. Otherwise users could suffer from disenchantment with MPI on their first usage and may never give the standard a second try. Having the common practice routines in the subset should encourage developers to optimize these routines in the initial version of MPI. Also, though some felt that the portability offered by the MPI subset was sufficient to entice users, many felt that the subset should contain some new features over and above current common practice as an added incentive. The additional items selected were meant to represent some of the significant new features of MPI. Balanced against these first two goals is the need for the subset to show up in a timely fashion. Thus, each feature chosen for inclusion in the subset was deemed of sufficient importance to outweigh its added complexity in implementing the subset. Though some functions seem easy to implement, there are often overlooked costs in testing, documentation and development that were considered before a feature was added to the MPI subset. Inclusion of too many routines might lead to moderately efficient implementation of them all instead of a very efficient and possibly more useful implementation of a smaller subset.

9.3 Subset Functionality

The sections below contain a general summary and specific routines of what is included in/excluded from the subset for each chapter. The general requirements of each chapter also hold for the subset. As examples, the message envelope in subsection 3.2.2, the semantics in section 3.4, the type matching in section 3.5 and the communication objects in subsection 3.8.1 are all part of the subset.

9.3.1 Point to Point Functionality

The point to point routines are not only central to MPI but represent common practice today. As such, the majority of the functionality in this chapter is included in the subset. Derived datatypes represent a significant new feature in MPI and is included in the subset. However, only basic datatypes of section 3.2.1 will be allowed as input to the datatype construction routines described in section 3.13.1.

To reduce the difficulty of implementation, several features were left out of the subset. The ability to test for multiple completions of messages was removed. Also, the ability to cancel a message and use persistent communication objects has been removed. Finally, the routines associated with ready and synchronous communication modes were excluded.

The table below contains a listing by section of which routines from the point to point communication chapter are included and excluded from the subset.

Section	Title	Included	Excluded
3.2	Basic Send Operation	MPI_SEND	——
3.2.1	Message Data	All types	——
3.3	Basic Recv Operation	MPI_RECV	——
3.3.1	Return Status	MPI_GET_SOURCE MPI_GET_TAG MPI_GET_COUNT	——
3.7	Comm. Modes	Standard	MPI_RSEND MPI_SSEND
3.8.2	Comm. Initiation	MPI_ISEND MPI_IRECV	MPI_IRSEND MPI_ISSEND
3.8.3	Comm. Completion	MPI_WAIT MPI_TEST	——
3.8.3	Multiple Completion	——	MPI_WAITANY MPI_TESTANY MPI_WAITALL MPI_TESTALL
3.9	Probe & Cancel	MPI_PROBE MPI_IPROBE MPI_GET_COUNT MPI_GET_ELEMENT	MPI_CANCEL MPI_TEST_CANCELLED
3.10	Persistent Comm. Objects	——	MPI_CREATE_SEND MPI_CREATE_RSEND MPI_CREATE_SSEND MPI_CREATE_RECV MPI_START MPI_COMMOBJ_FREE
3.11	Send-receive	MPI_SENDRECV	MPI_EXCHANGE
3.12	Null process	MPI_PROCNULL	——
3.13.1	Derived Datatypes	MPI_TYPE_CONTIGUOUS MPI_TYPE_VECTOR MPI_TYPE_HVECTOR MPI_TYPE_INDEXED MPI_TYPE_HINDEXED MPI_TYPE_STRUCT	See text above
3.13.2	Address and extent functions	MPI_ADDRESS MPI_TYPE_EXTENT	
3.13.3	Lower-bound and upper-bound	MPI_LB MPI_UB	
3.13.4	Commit and free	MPI_TYPE_COMMIT MPI_TYPE_FREE	——

9.3.2 Collective Communication Functionality

As with point to point functionality, collective communication is common practice today, and the functionality is generally included in the subset. Furthermore, the ability to define arbitrary groups in the MPI subset, which are then associated with a communicator, allows the user to perform collective communication operations over an arbitrary set of processes. This increases the functionality of the collective communication routines over what is commonly available today.

To ease initial implementation, the user defined reduce functions are excluded from the subset. Note that all of the system provided reduce functions listed in section 4.5.1 are include in the subset. Also, the scan functions are excluded from the subset. Finally, the generalized "v" versions of routines are excluded from the subset as well as MPI_REDUCE_SCATTER.

The table below contains a listing by section of which routines from the collective communication chapter are included and excluded from the subset.

Section	Title	Included	Excluded
4.3	Barrier Synch.	MPI_BARRIER	——
4.4.1	Broadcast	MPI_BCAST	——
4.4.2	Gather	MPI_GATHER	MPI_GATHERV
4.4.4	Scatter	MPI_SCATTER	MPI_SCATTERV
4.4.6	All-to-all broadcast	MPI_ALLCAST	MPI_ALLCASTV
4.4.7	All-to-all scatter-gather	MPI_ALLTOALL	MPI_ALLTOALLV
4.5.1	Reduce	MPI_REDUCE	MPI_USER_REDUCE MPI_USER_REDUCEA
		MPI_ALLREDUCE	MPI_USER_ALLREDUCE MPI_USER_ALLREDUCEA MPI_REDUCE_SCATTER MPI_USER_REDUCE_SCATTER MPI_USER_REDUCE_SCATTERA
4.5.4	Scan	——	MPI_SCAN MPI_USER_SCAN MPI_USER_SCANA

9.3.3 Groups, Contexts, and Communicators Functionality

Groups, contexts and communicators are used throughout the MPI standard. Though these concepts are used in some packages today, their inclusion in MPI represents a significant new feature over what is generally available in common practice today. For these reasons, this functionality is included in the subset. However, implementation of groups, contexts and communicators is also a significant effort. As a compromise, the MPI intra-communicator functionality is included but the intra-communicator functionality is excluded. The caching facility was included to support the topology functionality and extensions.

The table below contains a listing by section of which routines from the "Groups, Contexts, and Communicators" chapter are included and excluded from the subset.

Section	Title	Included	Excluded
5.2.4	Predefined Comm.	MPI_COMM_WORLD	——
5.3.1	Group Accessors	MPI_GROUP_SIZE MPI_GROUP_RANK MPI_TRANSLATE_RANKS	——
5.3.2	Group Const.	MPI_GROUP_UNION MPI_GROUP_INTERSECTION MPI_GROUP_DIFFERENCE MPI_GROUP_INCL MPI_GROUP_EXCL MPI_GROUP_RANGE_INCL MPI_GROUP_RANGE_EXCL	——
5.3.3	Group Destruct.	MPI_GROUP_FREE	——
5.4.1	Comm. Access.	MPI_COMM_SIZE MPI_COMM_RANK MPI_COMM_GROUP	——
5.4.2	Comm. Const.	MPI_COMM_DUP MPI_COMM_MAKE MPI_COMM_SPLIT	——
5.4.3	Comm. Destruct.	MPI_COMM_FREE	——
5.5.1	Comm. Access.	——	MPI_COMM_STAT MPI_COMM_MERGE
5.5.2	InterComm. Const. & Destr.	——	MPI_INTERCOMM_MAKE MPI_INTERCOMM_START MPI_INTERCOMM_FINISH
5.5.3	Name Service	——	MPI_INTERCOMM_NAME MPI_INTERCOMM_NAME_START
5.6.1	Functionality	——	MPI_ATTRIBUTE_ALLOC MPI_ATTRIBUTE_FREE MPI_GET_ATTRIBUTE_KEY MPI_SET_ATTRIBUTE MPI_TEST_ATTRIBUTE MPI_DELETE_ATTRIBUTE

9.3.4 Topology Functionality

The capability presented in the process topology chapter represents a new functionality in MPI and is not generally common practice today. To make this available in the subset, the functionality associated with Cartesian topologies is included but the general graph topologies are excluded.

The table below contains a listing by section of which routines from the topology chapter are included and excluded from the subset.

Section	Title	Included	Excluded
6.5.4	Low–level topol. func.	MPI_MAP_CART	MPI_MAP_GRAPH
??	High–level topol. func.	MPI_CART	MPI_GRAPH
6.5.1	Topol. inquiry func.	MPI_INQMAP	
		MPI_INQCARTDIM	MPI_INQGRAPHDIMS
		MPI_INQCART	MPI_INQGRAPH
		MPI_INQRANK	
		MPI_INQLOC	
6.5.3	Partitioning of Cart.	MPI_PARTC	
		MPI_SHIFT_ID	
		MPI_MAKDIM	

9.3.5 Language Binding

The subset shall contain the Fortran 77 and C language bindings. All the routines included in the subset will conform to the rules stated in chapter 2 and annex A.

9.3.6 MPI Environmental Management

The ability for the user to make inquiries of the MPI system is an important feature in portable programming. For this reason most of the functions are included in the subset. However, the ability for the user to specify an error handler is not included in the subset nor is the ability to unpack derived datatypes. The first is consistent with not including user called functions in the subset and the second is of minimal impact since derived datatypes can only have basic datatypes in them.

The table below contains a listing by section of which routines from the MPI Environmental Management chapter are included and excluded from the subset.

Section	Title	Included	Excluded
7.1	Implem. Info.	MPI_GET_VALID_TAG_RANGE	
7.1.2	Buffering	MPI_GET_BUFFER_PARAMS	
		MPI_SUGGEST_BUFFER_PARAMS	
		MPI_USER_SPECIFIES_BUFFER	
7.2	Error handling	MPI_GET_ERRORMODE	
		MPI_SET_ERRORMODE	MPI_SET_ERRHANDLER
		MPI_ERROR_STRING	
??	"Unpacking"		MPI_TYPE_QUERY_CONSTRUCTOR
7.3	Startup	MPI_INIT	
		MPI_FINALIZE	
		MPI_INITIALIZED	
		MPI_ABORT	

9.3.7 Profiling Functionality

Profiling is a capability which will be of great utility in MPI for determining the performance of each implementation. Furthermore, though implementation of the profiling capabilities doubles the number of routines supplied, these routines can easily be generated from the routines required elsewhere in the subset. Given these considerations, the subset will contain the shadow routines "PMPI_" for each routine included in the subset.

9.4 Subset Testing

The test suite is expected to also include a subset which will verify the functionality of this subset.

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] 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.

[4] 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.

[5] R. Butler and E. Lusk. User's guide to the P4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992.

[6] Robert 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991.

[12] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992.

[13] D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991.

[14] 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.

[15] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990.

[16] H. Mühlenbein O. Krämer. Mapping strategies in message–based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989.

[17] Parasoft Corporation. *Express Version 1.0: A Communication Environment for Parallel Computers*, 1988.

[18] 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.

[19] 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.

[20] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992.

[21] 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

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 */
MPI_SUCCESS

/* assorted constants (both C and Fortran) */
MPI_BOTTOM
MPI_PROCNULL
MPI_ANY_SOURCE
MPI_ANY_TAG
MPI_UNDEFINED
MPI_UNDEFINED_RANK
MPI_UB
MPI_LB

/* constant array sizes (Fortran) */
MPI_STATUS_SIZE

/* attributes (C and Fortran) */
MPI_UNDEFINED_ATTR
MPI_FOUND
MPI_NOT_FOUND
MPI_ATTR_COPY_FREELY
MPI_ATTR_COPY_NEVER
MPI_ATTRCOPY_CALLBACK
```

[1] Version as of October 13, 1993

```
/* Error-handling specifiers (C and Fortran) */
MPI_ERRORSFATAL
MPI_ERRORSRETURN
MPI_ERRORSCALLUSERROUTINE

/* Pre-defined error handlers (C and Fortran) */
MPI_NOHANDLER
MPI_DEFAULTHANDLER

/* Maximum sizes for strings */
MPI_MAX_INTERCOMM_NAME_SIZE

/* elementary datatypes (C) */
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_BYTE
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED_LONG
MPI_UNSIGNED_INT

/* elementary datatypes (Fortran) */
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_COMPLEX
MPI_DOUBLE_COMPLEX
MPI_LOGICAL
MPI_CHARACTER
MPI_BYTE

/* reserved communicators (C and Fortran) */
MPI_COMM_WORLD

/* collective operations (C and Fortran) */
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_AND
MPI_OR
MPI_XOR
MPI_MAXLOC
```

```
MPI_MINLOC

/* topologies (C and Fortran) */
MPI_REORDER
MPI_NOREORDER
MPI_GRAPHTOP
MPI_CARTTOP
MPI_NOTOP
MPI_PER
MPI_NOPER
```

The following are defined C type definitions, also included in the file `mpi.h`.

```
/* opaque types (C) */
MPI_Persistence
MPI_Mode
MPI_Op
MPI_Comm_name
MPI_Addr
MPI_Errmode

/* assorted opaque structures (C) */
MPI_Status
MPI_Group
MPI_Communicator
MPI_Datatype
MPI_Comm_request
MPI_Attribute
MPI_Intercomm_request

/* Function types (C) */
MPI_Errhandler
```

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_source(MPI_Status status, int *source)

int MPI_Get_tag(MPI_Status status, int *tag)

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

int MPI_Rsend(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_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Comm_request *request)

int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Comm_request *request)

int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Comm_request *request)

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Comm_request *request)

int MPI_Wait(MPI_Comm_request request, MPI_Status *status)

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

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

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

int MPI_Waitall(int count, MPI_Comm_request *array_of_requests,
             MPI_Status *array_of_statuses)

int MPI_Testall(int count, MPI_Comm_request *array_of_requests, int *flag,
             MPI_Status *array_of_statuses)

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

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

int MPI_Cancel(MPI_Comm_request request)

int MPI_Test_cancelled(MPI_Status status, int *flag)

int MPI_Create_send(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Comm_request *request)

int MPI_Create_rsend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Comm_request *request)

int MPI_Create_ssend(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Comm_request *request)

int MPI_Create_recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Comm_request *request)

int MPI_Start(MPI_Comm_request request)

int MPI_Startall(int count, MPI_Comm_request *array_of_requests)

int MPI_Commreq_free(MPI_Comm_request *request)
```

```
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)

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)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
             MPI_Datatype *newtype)

int MPI_Type_vector(int count, MPI_Datatype oldtype, int stride,
             int blocklength, MPI_Datatype *newtype)

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

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

int MPI_Type_hindexed(int count, MPI_Datatype oldtype,
             int *array_of_displacements, int *array_of_blocklengths,
             MPI_Datatype *newtype)

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

int MPI_Address(void* location, MPI_ADDR address)

int MPI_Type_extent(MPI_Datatype datatype, int *extent)

int MPI_Type_size(MPI_Datatype datatype, int *size)

int MPI_Type_count(MPI_Datatype datatype, int *count)

int MPI_Type_commit(MPI_Datatype datatype)

int MPI_Type_free(MPI_Datatype datatype)

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

A.4 C Bindings for Collective Communication

```
int MPI_BARRIER(MPI_Comm comm )

int  MPI_BCAST(void* buf, int count, MPI_Datatype datatype, int root,
             MPI_Comm comm )

int MPI_GATHER(void* sendbuf, int sendcnt, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype, int root,
             MPI_Comm comm)

int MPI_GATHERV(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int *displs, int *recvcounts, MPI_Datatype,
             int root, MPI_Comm comm)

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

int MPI_SCATTERV(void* sendbuf, int *displs, int *sendcounts,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype, int root, MPI_Comm comm)

int MPI_ALLGATHER(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype, MPI_Comm comm)

int MPI_ALLGATHERV(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int *displs, int *recvcounts, MPI_Datatype,
             MPI_Comm comm)

int MPI_ALLTOALL(void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype, MPI_Comm comm)

int MPI_ALLTOALLV(void* sendbuf, int *sdispls, int *sendcounts,
             MPI_Datatype sendtype, void* recvbuf, int *rdispls,
             int *recvcounts, MPI_Datatype, MPI_Comm comm)

int MPI_REDUCE(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void op, int root, MPI_Comm comm)

int MPI_USER_REDUCE(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void function, int root, MPI_Comm comm)

int MPI_USER_REDUCEA(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void function, int root, MPI_Comm comm)

int MPI_ALLREDUCE(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void op, MPI_Comm comm)

int MPI_USER_ALLREDUCE(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void function, MPI_Comm comm)

   eint MPI_USER_ALLREDUCEA(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void function, MPI_Comm comm)
```

```
int MPI_REDUCE_SCATTER(void* sendbuf, void* recvbuf, int *recvcounts,
             MPI_Datatype datatype, void op, MPI_Comm comm)

int MPI_SCAN(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void op, MPI_Comm comm )

int MPI_USER_SCAN(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void function, MPI_Comm comm)

int MPI_USER_SCANA(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void function, 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_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,
             MPI_Group *newgroup)

int MPI_Group_range_excl(MPI_group group, int n, int **ranges,
             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_group(MPI_Comm comm, MPI_group *group)

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

int MPI_Comm_make(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_Intercomm_start(MPI_Comm local_comm, int local_leader,
             MPI_Comm peer_comm, int remote_leader, int tag,
             MPI_Intercomm_request *inter_request)

int MPI_Intercomm_finish(MPI_Intercomm_request *inter_request,
             MPI_Comm *newcomm)

int MPI_Intercomm_merge(MPI_Comm intercomm, int key, MPI_Comm *newintracomm)

int MPI_Intercomm_name_start(MPI_Comm comm, char *name,
             MPI_Intercomm_request *inter_request)

int MPI_Attr_get_key(copy_fn, delete_fn, int keyval, void* extra_state)

int MPI_Attr_free_key(int keyval)

int MPI_Attr_put_value(MPI_Comm comm, int keyval, void* attribute_val)

int MPI_Attr_get_value(MPI_Comm comm, int keyval, void* attribute_val,
             int *found)

int MPI_Attr_delete_value(MPI_Comm comm, int keyval)
```

A.6 C Bindings for Process Topologies

A.7 C bindings for Environmental Inquiry

```
int MPI_GET_VALID_TAG_RANGE(int *low, int *high)

int MPI_GET_PROCESSOR_NAME(int len, char *name)

int MPI_GET_HOST(MPI_Comm comm, int *rank)

int MPI_GET_BUFFER_PARAMS(int *numsend, int *sizesend, int *numrecv,
             int *sizerecv, int *totalnumsednd, int *totalsizesen,
             int *totalnumrecv, int *totalsizerecv)

int MPI_SUGGEST_BUFFER_PARAMS(int numsend, int sizesend, int numrecv,
             int sizerecv, int totalnumsend, int totalsizesen,
             int totalnumrecv, int totalsizerecv, int *flag)

int MPI_USER_SPECIFIES_BUFFER(MPI_Comm comm, void* buffer, int size)

int MPI_GET_ERRORMODE(MPI_Comm comm, MPI_Errmode mode)

int MPI_SET_ERRORMODE(MPI_Comm comm, MPI_Errmode mode)

int MPI_SET_ERRHANDLER(MPI_Comm comm, MPI_Errhandler handler)

int MPI_ERROR_STRING(int errorcode, int len, char *string)

int MPI_INIT(int argc, char **argv)

int MPI_FINALIZE()

int MPI_INITIALIZED(int *flag)
```

```
int MPI_ABORT(MPI_Comm comm, int errorcode)
```

A.8 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_SOURCE(STATUS, SOURCE, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), SOURCE, IERROR

MPI_GET_TAG(STATUS, TAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), TAG, IERROR

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

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

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

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

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

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

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

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

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
```

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

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

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

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

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

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

MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), FLAG, IERROR

MPI_CREATE_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    INTEGER REQUEST, BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_CREATE_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    INTEGER BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_CREATE_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    INTEGER BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_CREATE_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    INTEGER BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

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

MPI_COMMOBJ_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

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, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
```

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

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

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

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

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

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

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

MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR

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

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

MPI_TYPE_COUNT(DATATYPE, COUNT, IERROR)
    INTEGER DATATYPE, COUNT, 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
```

A.9 Fortran Bindings for Collective Communication

```
MPI_BARRIER(COMM)
    INTEGER COMM, IERROR

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

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

MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, DISPLS, RECVCOUNTS,
             RECVTYPE, ROOT, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DISPLS, RECVCOUNTS, RECVTYPE, ROOT, COMM,
    IERROR

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

MPI_SCATTERV(SENDBUF, DISPLS, SENDCOUNTS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER DISPLS, SENDCOUNTS, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM,
    IERROR

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

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, DISPLS, RECVCOUNTS,
             RECVTYPE, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DISPLS, RECVCOUNTS, RECVTYPE, COMM, IERROR

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

MPI_ALLTOALLV(SENDBUF, SDISPLS, SENDCOUNTS, SENDTYPE, RECVBUF, RDISPLS,
             RECVCOUNTS, RECVTYPE, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SDISPLS, SENDCOUNTS, SENDTYPE, RDISPLS, RECVCOUNTS, RECVTYPE,
    COMM, IERROR

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM)
```

```
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

MPI_USER_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, ROOT, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

MPI_USER_REDUCEA(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, ROOT, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

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

MPI_USER_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, COMM, IERROR

MPI_USER_ALLREDUCEA(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, COMM, IERROR

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

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

MPI_USER_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, COMM, IERROR

MPI_USER_SCANA(SENDBUF, RECVBUF, COUNT, DATATYPE, FUNCTION, COMM)
    <type> SENDBUF(*), RECVBUF(*)
    EXTERNAL FUNCTION
    INTEGER COUNT, DATATYPE, COMM, IERROR
```

A.10 Fortran Bindings for Groups, Contexts, and Communicators

```
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_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(*,*), NEWGROUP, IERROR

MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(*,*), 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_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR

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

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

MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, FLAG, IERROR

MPI_INTERCOMM_START(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
              NEW_COMM, INTER_REQUEST, IERROR)
```

```
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    INTER_REQUEST, IERROR

MPI_INTERCOMM_FINISH(INTER_REQUEST, IERROR)
    INTEGER INTER_REQUEST, NEWCOMM, IERROR

MPI_INTERCOMM_MERGE(INTERCOMM, KEY, NEWINTERCOMM, IERROR)
    INTEGER INTERCOMM, KEY, NEWINTRACOMM, IERROR

MPI_INTERCOMM_NAME_START(COMM, NAME, NEW_COMM, INTER_REQUEST, IERROR)
    INTEGER COMM, NAME, NEW_COMM, INTER_REQUEST, IERROR

MPI_ATTR_GET_KEY(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    INTEGER COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR

MPI_ATTR_FREE_KEY(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR

MPI_ATTR_PUT_VALUE(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

MPI_ATTR_GET_VALUE(COMM, KEYVAL, ATTRIBUTE_VAL, FOUND, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, FOUND, IERROR

MPI_ATTR_DELETE_VALUE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
```

A.11 Fortran Bindings for Process Topologies

A.12 Fortran Bindings for Environmental Inquiry

```
MPI_GET_VALID_TAG_RANGE(LOW, HIGH, IERROR)
    INTEGER LOW, HIGH, IERROR

MPI_GET_PROCESSOR_NAME(LEN, NAME, IERROR)
    INTEGER LEN, NAME, IERROR

MPI_GET_HOST(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR

MPI_GET_BUFFER_PARAMS(NUMSEND, SIZESEND, NUMRECV, SIZERECV, TOTALNUMSEDND,
              TOTALSIZESEN, TOTALNUMRECV, TOTALSIZERECV, IERROR)
    INTEGER NUMSEND, SIZESEND, NUMRECV, SIZERECV, TOTALNUMSEDND,
    TOTALSIZESEN, TOTALNUMRECV, TOTALSIZERECV, IERROR

MPI_SUGGEST_BUFFER_PARAMS(NUMSEND, SIZESEND, NUMRECV, SIZERECV,
              TOTALNUMSEND, TOTALSIZESEN, TOTALNUMRECV, TOTALSIZERECV, FLAG,
              IERROR)
    INTEGER NUMSEND, SIZESEND, NUMRECV, SIZERECV, TOTALNUMSEND,
    TOTALSIZESEN, TOTALNUMRECV, TOTALSIZERECV, FLAG, IERROR

MPI_USER_SPECIFIES_BUFFER(COMM, BUFFER, SIZE, IERROR)
    <type> SENDBUF(*)
```

```
    INTEGER COMM, BUFFER, SIZE, IERROR

MPI_GET_ERRORMODE(COMM, MODE, IERROR)
    INTEGER COMM, MODE, IERROR

MPI_SET_ERRORMODE(COMM, MODE, IERROR)
    INTEGER COMM, MODE, IERROR

MPI_SET_ERRHANDLER(COMM, HANDLER, IERROR)
    INTEGER COMM, HANDLER, IERROR

MPI_ERROR_STRING(ERRORCODE, LEN, STRING, IERROR)
    INTEGER ERRORCODE, LEN, STRING, IERROR

MPI_INIT(IERROR)
    INTEGER IERROR

MPI_FINALIZE(IERROR)
    INTEGER IERROR

MPI_INITIALIZED(FLAG, IERROR)
    INTEGER FLAG, IERROR

MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

MPI Function Index