D R A F T
Document for a Standard Message-Passing Interface

Message Passing Interface Forum

September 21, 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.

Contents

Abstract

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 September 21, 1993.

MPIF invites comments on the technical content of MPI, as well as on the editorial presentation in the document. Comments received before July 1, 1993 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.

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

Do we want an organization/company list here?

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

Ed Anderson	Joe Baron	Eric Barszcz
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	C. T. Howard Ho
Alex Ho	Steven Huss-Lederman	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		

The following organizations made the draft available by anonymous FTP access and/or mail servers: Oak Ridge National Laboratory, These outlets were instrumental in distributing the document.

Talk about financial support here.

Section 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 [15], Express [14], nCUBE's Vertex [12], P4 [4], and PARMACS [3, 5]. Other important contributions have come from Zipcode [16, 17], Chimp [8, 9], PVM [10, 18], and PICL [11].

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 [19]. 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 [?]. 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 as of September 21, 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. Although MPI1 and the MPI draft standard described in this paper have many features in common, they are distinct proposals, with MPI1 now being largely superseded by the MPI draft standard.

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 coprocessor, where available.
- Allow (but not mandate) extensions for use in 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.
- Define an interface that can be quickly implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- The interface should not contain more functions than are really necessary.
- Semantics of the interface should be language independent.

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.

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
- Model implementation
- Validation suite

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

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.

1.6 Organization of this Document

Describe where everything is located in the MPIF document.

Section 2

MPI Terms and Conventions

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

2.1 Document Conventions

Discussions that are not part of the standard are marked with **"Discussion"** and appear in small type:

Discussion: A "discussion" looks like this.

Advice to implementors of MPI will appear in a note like this:

Implementation note: Advice on implementation goes here.

2.2 Procedure Specification

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

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

If a parameter is a handle to an opaque object, and the object is updated by the procedure call, then the parameter 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 parameters, because such use is error-prone, especially for scalar parameters.

[1] Version as of September 20, 1993

A common occurrence for MPI functions is a parameter that is used as IN by some processes and OUT by other processes. Such parameters are, syntactically, an INOUT parameter, although, semantically, it is not used in one call both for input and for output.

Discussion: We might want to distinguish this use; e.g. IN/OUT vs INOUT.

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

2.3 Semantical Terms

A communication operation is termed to be:

nonblocking if it may return before the related communication operation completes;

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

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

An operation is **local** if its execution does not entail communication, and does not depend on the state of other processes; **global**, if its completion may depend on communication occurring; **collective** if several processes need to participate in its execution.

2.4 Data Types

2.4.1 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 (structured) 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 parameters.

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 parameters handles to such objects.

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

Discussion: This may need some clarification if "flatten/unflatten" is accepted in the context draft.

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 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 buffer descriptor objects, 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.

Discussion: Need a test to see if a handle is null. In C, if handles are pointers, a comparison to `NULL` can be used. In Fortran, we may either have a function MPI_IS_NULL (and make it available in C, too, for compatibility reasons), or have a `NULL` value that can be used in comparisons, in a manner similar to a C null pointer.

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

2.4.2 Array of handles

An MPI call may need a parameter 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 parameter is required to indicate the number of valid entries (unless this number can be derived otherwise).

2.4.3 State

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

2.4.4 Named constants

MPI procedures sometimes assign a special meaning to a special value of a basic type parameter; e.g. tag is an integer valued parameter of point-to-point communication operations, with a special wild-card value, MPI_ANY_TAG. Such parameters 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 environment inquiry or environment setting functions (Chapter 8).

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 will be flagged as an error.

2.4.5 Choice

MPI functions sometimes use parameters with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual parameters of different types. The mechanism for providing such parameters will differ from language to language.

Discussion: The Fortran 77 standard specifies that the type of actual arguments need to agree with the type of dummy arguments; no construct equivalent to C pointers is available. Thus, it would seem that there is no standard conforming mechanism to support choice parameters. However, most Fortran compilers either don't check type consistency of calls to external routines, or support a special mechanism to link foreign (e.g., C) routines. We accept this non-conformity with the Fortran 77 standard. I.e., we accept that the same routine may be passed an actual parameter of a different type at distinct calls.

Generic routines can be used in Fortran 90 to provide a standard conforming solution. This solution will be consistent with our nonstandard conforming Fortran 77 solution.

2.5 Language Binding

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

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

2.5.1 Data Types and Naming Conventions

All named constants, routine names, and variable names will begin with the letters "MPI". In the C binding, these will be upper case, with all remaining letters in lower case. In Fortran, the entire name will be lower case.

Handles are represented as integers in Fortran and as pointers to structures in C. Typed communication routines support **string**s, **double precision**, **real**, and **integer** data in Fortran. Null-terminated **string**s, **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 Fortran, there is an additional type **cbyte**, which identifies character data upon which no conversion is to be performed. In C, the definitions of named constants, function prototypes, and aggregate type information must be supplied in an include file `mpi.h`.

Discussion: Different Fortran implementation may pass characters in a manner different from the way in which numeric data is passed to a routine. For all datatypes *except* `MPI_byte`, the datatype may be used to decide how the parameter was passed. Since byte does not specify the type, Fortran requires a "numeric" byte (`MPI_byte`) and a "character" byte (`MPI_cbyte`).

2.5.2 Fortran 77 Binding Issues

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

- Identifiers are limited to thirty, not six, significant characters.
- Identifiers may contain underscores after the first character.
- The types of the formal argument list may not match the calling argument list. (An example is given in figure 2.5.2.) That is, the same MPI subroutine may be called with different argument types.
- Although not required, it is strongly suggested that named constants be provided in an include file.
- All Fortran subroutines return a return code in the last argument.

2.5.3 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 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.

```
double precision a
call foo(a)
    .
    .
    .
subroutine foo(b)
integer b
```

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

2.6 Processes

An MPI program is executed by several autonomous processes that each execute their own code, in a 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.

2.7 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementer 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.

Of course, MPI programs may still be erroneous. A **program error** can occur when an MPI call is called with an incorrect parameter (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. One can also write **safe** programs, that are not subject to resource errors.

All MPI calls return an error parameter that indicates successful completion of the operation, or the error condition that occurred, otherwise. However, MPI does not require the user to check error parameters. It is recommended to provide an implementation where the environment for an MPI execution 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.

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 parameter 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 parameter 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.

Implementation note: A possible mechanism is to have MPI calls generate a signal that can be ignored when an error is encountered. A more modular mechanism that allows one to handle an exception within its context is preferred.

The behavior of erroneous programs is not defined by this document. The desired behavior is that a relevant error code be returned; however, efficiency considerations may dictate that the system does not detect certain types of errors.

Section 3

Introduction to Groups, Contexts, and Communicators

This section serves as an introduction to the concepts of process groups, contexts, and communicators. We say enough here so that we can proceed with describing point to point and collective communications in the next two sections. After that, we revisit communicators in Section 6, "More on Groups, Contexts, and Communicators," and explain them in full detail.

3.1 Motivation

It is highly desirable that processes executing a parallel procedure use a "virtual process name space" local to the invocation. Thus, the code of the parallel procedure will look identical, irrespective of the absolute addresses of the executing processes. It is often the case that parallel application code is built by composing several parallel modules (*e.g.*, a numerical solver, and a graphic display module). Support of a virtual name space for each module will allow for the composition of modules that were developed separately without changing all message passing calls within each module. The set of processes that execute a parallel procedure may be fixed, or may be determined dynamically before the invocation. Thus, MPI has to provide a mechanism for dynamically creating sets of locally named processes. We always number processes that execute a parallel procedure consecutively, starting from zero, and call this numbering **rank in group**. Thus, a **group** is an ordered set of processes, where processes are identified by their ranks when communication occurs.

Communication `contexts` partition the message-passing space into separate, manageable "universes." Specifically, a send made in a context cannot be received in another context. Contexts are identified in MPI using opaque **contexts** that reside within `communicator` objects. The context mechanism is need to allow predictable behavior in subprograms, and to allow dynamicism in message usage that cannot be reasonably anticipated or managed. Normally, a parallel procedure is written so that all messages produced during its execution are also consumed by the processes that execute the parallel procedure. However, if one parallel procedure calls another, then it might be desirable to allow such call to proceed while messages are pending (the messages will be consumed by the procedure after the call returns). In such case, a new communication context is needed for the called parallel procedure, even if the transfer of control is synchronized.

[1]Version of August 13, 1993

The communication domain used by a parallel procedure is identified by a **communicator**. Communicators bring together the concepts of process group and communication context. A communicator is an explicit parameter in each point-to-point communication operation. The communicator identifies the communication context of that operation; it identifies the group of processes that can be involved in this communication; and it provides the translation from virtual process names, which are ranks within the group, into absolute addresses. Collective communication calls also take a communicator as parameter; it is expected that parallel libraries will be built to accept a communicator as parameter. Communicators are represented by opaque MPI objects.

3.2 Groups

A **group** is an ordered set of process identifiers (henceforth processes); process identifiers are implementation dependent; a group is an opaque object. 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.

3.3 Contexts

A **context** is the MPI mechanism for partitioning communication space. A defining property of a context is that a send made in a context cannot be received in another context. A context is an opaque object. Only one communicator in a process may bind the same context. Contexts have additional attributes for inter-communication, to be discussed below. For intra-communication, a context is essentially a hyper-tag needed to make a communicator safe for point-to-point and MPI-defined collective communication.

Discussion: Some implementations may make a context to be a pair of integers, each representing "hyper tags" – one for point-to-point and one for (MPI-defined) collective operations on a communicator. By making this concept opaque, we relieve the implementor of the requirement that this is the only way to implement contexts correctly for MPI.

Discussion: Among other reasons, including to address Jim Cownie's concerns about safety and to make both point-to-point and collective communication safer on an intra-communincator, we have opted to make contexts opaque, at the expense of upsetting those who want to be able to set context values. This change is crucial to abstracting MPI from specific implementations, and forces specific implementations to provide implementation-specific functions to make the connotation of contexts with specific integer values.

3.4 Communicators

All MPI communication (both point-to-point and collective) functions use **communicators** to provide a specific scope (context and group specifications) for the communication.

In short, communicators bring together the concepts of group and context; (furthermore, to support implementation-specific optimizations, and virtual topologies, they "cache" additional information opaquely). The source and destination of a message is identified by the rank of that process within the group; no a priori membership restrictions on the process sending or receiving the message are implied. For collective communication, the communicator specifies the set of processes that participate in the collective operation. 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.

Raison d'être for Contexts and Communicators Within a communicator, one or more contexts are cached. We want to make it possible for libraries quickly to achieve additional safe communication space without MPI-communicator-based synchronization. The only way to do this is to provide a means to preallocate many contexts, and bind them locally, as needed. This choice weakens the overall, inherent "safety" of MPI, if programmed in this way, but provides added performance which library designers will demand. We are convinced that the means we have chosen provides a good compromise between efficiency and safety.

3.4.1 Predefined Communicators

Initial communicators defined once MPI_INIT has been called are as follows:

- MPI_COMM_ALL, A communicator of all processes in the user program including a host process (if any).
- MPI_COMM_PEER, A communicator of all processes excluding a host process (if any). Note that in the absence of a host process this communicator will not be just the same as MPI_COMM_ALL since it will contain a different context.A

Discussion: In the host process, if there is one, what should MPI_COMM_PEER be? There are two consistent possibilities: a null communicator; a communicator in which the group contains only the host process.

MPI implementations are required to provide these communicators. The groups corresponding to these communicators do not appear as pre-defined constants, but they may be accessed using MPI_COMM_GROUP (see below).

Section 4

Point to Point Communication

4.1 Introduction

This chapter is a draft of the current proposal for point-to-point communication. Sending and receiving of messages by processes is the basic MPI communication mechanism. All other communication functions can be implemented on top of this basic communication layer (although a more direct implementation may lead to greater efficiency).

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.

4.2 Basic send operation

The syntax of the simplest send operation is given below.

[1]Version of September 11, 1993

MPI_SEND (start, count, datatype, dest, tag, comm)

IN	start	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* start, int count, MPI_Datatype datatype, int dest,
             MPI_Tag tag, MPI_Communicator comm)
```

```
mpi_send(start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer count, datatype, dest, tag, comm, rc
```

4.2.1 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 start. 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	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	

CHANGE

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.

Discussion:
One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit parameter. The need for such datatype information will become clear in Section 4.6.

4.2.2 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 8.

The context of the message sent is specified by the comm parameter. The message carries the context associated with this communicator (see Chapter 3).

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.

4.3 Basic receive operation

The syntax of the simplest receive operation is given below.

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

OUT	start	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

The receive buffer consists of the storage containing count consecutive elements of the type specified by datatype, starting at address start. 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 4.9 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 inter-communicators). 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.

4.3.1 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. For reasons that will become clear later, it is preferable to return this information in a separate **out** variable, rather then using the **count**, **source** and **tag** parameters both for input and output. The information is returned by the **status** parameter of the MPI_RECV function. This is a parameter of a special MPI-defined type. The status variable can be "decoded" to retrieve the **count**, **source** and **tag** fields, using the query functions listed below.

MPI_GET_SOURCE(status, source)

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

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)

Returns the tag of received message.

MPI_GET_COUNT(status, count)

IN	status	return status of receive operation (Status)
OUT	count	number of received elements (integer)

Returns the number of elements received. (Here, again, we count *elements*, not *bytes*.)

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.

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.

Discussion:

The use of a separate status parameter prevents errors that are often attached with INOUT parameters (e.g., passing the MPI_ANY_TAG constant as the actual argument for tag). The use of an explicit user object becomes important for nonblocking communication as it allows the receive operation to be stateless and, hence, reentrant. This prevents confusions in the case where multiple receives can be posted by a process.

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

All send and receive operations use the start, 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.

4.4 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 8.

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.

4.5 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.

Discussion:

The behavior of erroneous code is implementation-dependent. It is desirable but not required that errors are reported at run-time. An MPI implementation may decide to avoid type-checking for the sake of efficiency, in which case none of the three programs will report an error. On a system where real numbers occupy four bytes and both sender and receiver execute in the same environment, it is likely that all three codes will have the same behavior.

4.6 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.

Representation conversion is needed when data is moved across different environments, e.g. different machines that use different binary encodings for the same datatype, or codes compiled with different compiler options for datatype representation. Representation conversion preserves, to the greatest possible extent, the **value** represented. However, rounding errors and overflow and underflow exceptions may occur during floating point conversions; conversion of integers may also lead to exceptions when a value that can represented in one system cannot be represented in the other system. MPI does not specify rules for representation conversion.

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. 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. However, additional type information carried by messages can provide better error detection.

Discussion:
MPI does not handle inter-language communication because there seem to be no agreed standards for inter-language type conversions. The current design means that inter-language communication relies on the vendor specific rules for inter-language calls at each site.

4.7 Communication Modes

The basic send operation described in Section 4.2 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 **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 a blocking send 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. Thus, the basic send operation described in the previous section is **asynchronous**, since its return does not imply a synchronization with the (remote) receive operation, and does not imply **global** termination of the communication.

There are two additional communication modes:

A send that uses the **ready** communication mode may be started only if a 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 the sender provides additional information to the communication subsystem, when using the ready mode.

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.

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 (start, count, datatype, dest, tag, comm)

IN	start	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)

Send in synchronous mode

MPI_SSEND (start, count, datatype, dest, tag, comm)

IN	start	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)

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 modify the send buffer until the send completes and not to access or modify the receive buffer until the receive 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.

4.8 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 that returns as soon as possible. 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.

4.8.1 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.

4.8.2 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(handle, start, count, datatype, dest, tag, comm)

OUT	handle	communication object (handle)
IN	start	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)

Initiate a ready mode nonblocking communication.

MPI_IRSEND(handle, start, count, datatype, dest, tag, comm)

OUT	handle	communication object (handle)
IN	start	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)

Initiate a synchronous mode nonblocking communication.

MPI_ISSEND(handle, start, count, datatype, dest, tag, comm)

OUT	handle	communication object (handle)
IN	start	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)

Initiate a nonblocking receive.

MPI_IRECV (handle, start, count, datatype, source, tag, comm)

OUT	handle	communication object (handle)
OUT	start	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)

These calls allocate a communication object and associates it with the handle. The handle 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 start copying data out of the send buffer. The sender should not update 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 or update any part of the receive buffer after a nonblocking receive operation returns, until the receive completes.

4.8.3 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 (handle, status)

IN	handle	handle to communication object
OUT	status	status object

A call to MPI_WAIT returns when the send operation identified by handle is complete. If the communication object associated with this handle was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI_WAIT and the handle 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 4.3.1. 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 (handle, flag, status)

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

A call to MPI_TEST returns flag = true if the operation identified by handle 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 handle 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 4.3.1. 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(handle, a(1), 10, MPI_REAL, 1, tag, comm, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(handle, status, ierr)
  ELSE
    CALL MPI_IRECV(handle, a(1), 15, MPI_REAL, 0, tag, comm, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(handle, 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.

4.8.4 Semantics of Nonblocking Communications

The semantics of nonblocking communication is defined by suitably extending the definitions CHANGE
in Section 4.4 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.

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.

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(handle1, a(1), count, MPI_REAL, 1, tag1, comm, ierr)
    CALL MPI_ISEND(handle2, a(2), count, MPI_REAL, 1, tag2, comm, ierr)
    CALL MPI_WAIT(handle1, status, ierr)
    CALL MPI_WAIT(handle2, status, ierr)
  ELSE
    CALL MPI_IRECV(handle2, a(2), count, MPI_REAL, 0, tag2, comm, ierr)
    CALL MPI_IRECV(handle1, a(1), count, MPI_REAL, 0, tag1, comm, ierr)
    CALL MPI_WAIT(handle2, status, ierr)
    CALL MPI_WAIT(handle1, 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.

4.8.5 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_handles, index, status)

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

Blocks until one of the operations associated with the communication handles in the array has completed. Returns the index of that handle 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_handles, index, status) has the same effect as the execution of MPI_WAIT(array_of_handles[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_handles 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_handles, index, status)

IN	count	list length (integer)
IN	array_of_handles	array of handles to communication objects
OUT	index	index of handle for operation that completed, or MPI_UNDEFINED if none completed (integer)
OUT	status	return status object

Causes either one or none of the operations associated with the communication handles to return. In the former case, it returns in `index` the index of its handle in the array, returns in `status` the status of that operation, deallocates the associated communication object, and sets the handle 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_handles, index, status) has the same effect as the execution of MPI_TEST(array_of_handles[i], flag, status), for all `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. CHANGE

MPI_WAITALL(count, array_of_handles, array_of_status)

IN	count	lists length (integer)
IN	array_of_handles	array of handles to communication objects
OUT	array_of_status	array of status objects

Blocks until all communication operations associated with handles 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_status is set to the return status of the i-th operation. All communication objects are deallocated and all handles are set to null.

CHANGE

The execution of MPI_WAITALL(count, array_of_handles, array_of_status has the same effect as the execution of MPI_WAIT(array_of_handle[i], array_of_status[i]), for `i=0, ... , count-1`, in some arbitrary order.

MPI_TESTALL(count, array_of_handles, flag, array_of_status)

IN	count	lists length (integer)
IN	array_of_handles	array of handles to communication objects
OUT	flag	(logical)
OUT	array_of_status	array of status objects

Causes either all or none of the operations associated with the communication handles to complete. It returns flag = true if all communications associated with handles 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 handles are set to null. Otherwise, flag = false is returned, no handle 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(handle, a, n, MPI_REAL, 2, tag, comm, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(handle, status, ierr)
  ELSE         ! rank=2
    CALL MPI_IRECV(handle_list(0), a, n, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_IRECV(handle_list(1), b, n, MPI_REAL, 1, tag, comm, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAITANY(2, handle_list, index, status, ierr)
    IF(index.EQ.0)
      THEN
        **** handle message from process 0 ****
        CALL MPI_WAIT(handle_list(1), status, ierr)
        **** handle message from process 1 ****
      ELSE
        **** handle message from process 1 ****
        CALL MPI_WAIT(handle_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_status 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_handles are send operations, then an arbitrary value can be passed to the **status** (resp. **array_of_status**) argument.

An **array_of_handles** 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 handles containing only the non-null entries. As a handle in **array_of_handles** 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: CHANGE

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank < 2)
  THEN         ! rank=0 or rank=1
    CALL MPI_ISEND(handle, a, n, MPI_REAL, 2, tag, comm, ierr)
    CALL MPI_WAIT(handle, status, ierr)
  ELSE         ! rank=2
    CALL MPI_IRECV(handle_list(0), a, n, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_IRECV(handle_list(1), b, n, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_WAITANY(2, handle_list, index, status, ierr)
        **** handle first message  ****
    CALL MPI_WAITANY(2, handle_list, index, status, ierr)
        **** handle second message  ****
```

If MPI_WAITANY or MPI_TESTANY are called with an empty **array_of_handles** 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_handles** argument has no effect; MPI_TESTALL with an empty **array_of_handles** argument returns **flag=true**.

4.9 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

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 source and tag of the message matched are returned in the status object. These are the same values that would have been returned by a call to MPI_RECV executed at the same point in the program.

These values can be decoded from the return status object using the status query functions described in Section 4.3.1. The return status object is undefined if flag = false.

MPI_GET_COUNT cannot be used after a call to MPI_IPROBE to query on the length of the probed message. (The reason is that MPI_IPROBE does not carry a datatype argument, and the message type may not be known when the probe is executed. On the other hand, the datatype information is needed to compute the number of elements in a message.) Instead, a different status decoding function that provides the datatype as argument can be used in this case.

MPI_PROBE_COUNT(status, datatype, count)

IN	status	status object
IN	datatype	message datatype
OUT	count	number of entries in message (integer)

Returns the number of elements in the message that was last probed with status as an argument. The call is erroneous if datatype does not match the type of the message. It is erroneous to invoke MPI_PROBE_COUNT with a status argument that was set by a regular receive operation.

A subsequent receive executed with the same context, and the source and tag returned by the call to 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.

Discussion:
MPI guarantees that successive messages sent from a source to a destination within the same context are received in the order they are sent. Thus, MPI must support, either explicitly or implicitly, a FIFO structure to manage messages between each pair of processes, for each context. MPI_PROBE returns information on the first matching message in this FIFO; this will also be the message received by the first subsequent receive with the same source, tag and context as the message matched by MPI_PROBE.

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

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(handle)

IN	**handle**	communication object (handle)

A call to MPI_CANCEL marks for cancellation a pending nonblocking communication operation (send or receive). The cancel call is non-blocking, 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 about 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; 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)

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.

4.10 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(handle, start, count, datatype, dest, tag, comm)

OUT	handle	communication object (handle)
IN	start	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)

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

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

MPI_CREATE_RSEND(handle, start, count, datatype, dest, tag, comm)

OUT	handle	communication object (handle)
IN	start	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)

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

MPI_CREATE_SSEND(handle, start, count, datatype, dest, tag, comm)

OUT	handle	communication object (handle)
IN	start	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)

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

MPI_CREATE_RECV(handle, start, count, datatype, source, tag, comm)

OUT	handle	communication object (handle)
OUT	start	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)

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

MPI_START(handle)

INOUT	handle	communication object (handle)

The communication object associated with handle 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 4.8.

A communication started with a call to MPI_START is completed by a call to MPI_WAIT, MPI_TEST, or one of the derived functions described in Section 4.8.5. 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(handle)

INOUT	handle	communication object (handle)

Marks the communication object for deallocation. The object will be deallocated when there are no pending communications involving this object, at which point the handle becomes null.

The call is nonblocking. It is permissible to call MPI_COMMOBJ_FREE(handle) after a communication that uses handle 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 handle that has been freed.

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

$$\textbf{Create}\ (\textbf{Start}\ \textbf{Complete})^{*}\ \textbf{Free}\ ,$$

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.

4.11 Send-receive

The **send-receive** operations combines in one call the sending of a message to one destination and the receiving of another message, from another destination, possibly the same. The **exchange** operations are the same as send-receive, except that the send buffer and the receive buffer are identical. 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 or exchange operation is used, the communication subsystem takes care of these issues. Also, a send-receive operation is useful for implementing remote procedure calls.

A message sent by a send-receive or exchange operation can be received by a regular receive operation, and vice versa.

MPI_SENDRECV(send_start, send_count, send_type, dest, send_tag, recv_start, recv_count, recv_type, source, recv_tag, comm, status)

IN	send_start	initial address of send buffer (choice)
IN	send_count	number of elements in send buffer (integer)
IN	send_type	type of elements in send buffer
IN	dest	rank of destination (integer)
IN	send_tag	send tag (integer)
OUT	recv_start	initial address of receive buffer (choice)
IN	recv_count	number of elements in receive buffer (integer)
IN	recv_type	type of elements in receive buffer
IN	source	rank of source (integer)
IN	recv_tag	receive tag (integer)
IN	comm	communicator (handle)
OUT	status	status object

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_EXCHANGE(start, count, datatype, dest, send_tag, source, recv_tag, comm, status)

IN	start	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	send_tag	send message tag (integer)
IN	source	rank of source (integer)
IN	recv_tag	receive message tag (integer)
IN	comm	communicator (handle)
OUT	status	status object

Execute a blocking send and receive; the same buffer is used both for the send and for the receive.

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. 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. Note that some system buffering is required for a correct implementation of MPI_EXCHANGE. (Consider the last example, with send-receive replaced by exchange.)

4.12 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.

4.13 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**. 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. This type map, together with a base address $start$, specifies a communication buffer: the communication buffer that consists of n entries, where the i-th entry is at address $start + disp_i$ and has type $type_i$. A message assembled from such communication buffer will contain a sequence of n values, where value i has type $type_i$. The message will carry n values of types $(type_0, ..., type_{n-1})$.

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(start, 1, datatype,...) will use the send buffer defined by the base address start and the general datatype associated with datatype; it will generate a message with the type signature determined by the datatype argument. MPI_RECV(start, 1, datatype,...) will use the receive buffer defined by the base address start and the general datatype associated with datatype.

General datatypes can be used in all send and receive operations. We address later in Section 4.13.5 the case where the second argument count has value > 1.

The basic datatypes presented in Section 4.2.1 are particular cases of a general datatype, and are predefined. Thus, MPI_INT is a predefined handle to a datatype with type map $\{(\texttt{int}, 0)\}$, 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

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j disp_j + sizeof(type_j), \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap) + \epsilon. \end{aligned} \tag{4.1}$$

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 $Type = \{(\texttt{double}, 0), (\texttt{char}, 8)\}$ (a `double` at displacement zero, followed by a `char` at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are 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.

Discussion:

The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfil alignment constraints. This condition would be violated by a compiler that always align structures in an array to double word boundaries, even if they do not contain double word components. On a machine with such hypothetical compiler some of the functions introduced below become less useful. This is further discussed in Section 4.13.6.

4.13.1 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)

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)

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 4.13.6. (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)

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 be 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)

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

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_blocklength, 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_blocklength	number of elements in each block (array of nonnegative integers)
OUT	newtype	new datatype (handle)

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

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)

Example: Let type have type map

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

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

4.13.2 Address and extent functions

The displacements in a general datatype are relative to some start 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 start 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)

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.
```

Another useful auxiliary function is MPI_TYPE_EXTENT, that returns the extent of a datatype – where extent is as defined in Eq. 4.1.

MPI_TYPE_EXTENT(datatype, extent)

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

4.13.3 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 4.1. 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 4.13.6. 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 ($extent($MP_LB$) = extent($MP_UB$) = 0$). They do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

Example: Let `D = (-3, 0, 6)`; `T = (MPI_LB, MPI_INT, MPI_UB)`, and `B = (1, 1, 1)`. Then a call to MPI_TYPE_STRUC(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

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

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

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

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**.

4.13.4 Commit and free

CHANGE

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)

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

MPI_TYPE_FREE(datatype)

INOUT	datatype	handle to datatype object

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.

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.

4.13.5 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(start, count, datatype, ...), where count = 1 uses the send buffer defined by start and datatype; i.e. if datatype is a handle to the datatype with type map $\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}$, (with size zero entries deleted) then the send buffer consists of the entries of types $type_i$, at addresses start + $disp_i$, $i = 0, ..., n-1$. The message generated consists of a sequence of n entries, of types $type_0, ..., type_{n-1}$. The receive buffer for a call MPI_RECV(start, count, datatype , ...), is similarly defined, for count =1. The same applies to all other send and receive operations that have a datatype parameter.

A call of the form MPI_SEND(start, 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(start, count, datatype, dest, tag, comm) is equivalent to

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

If a communication operation uses as argument a general datatype type then each entry in the type map should match the datatype of the corresponding variable in the caller memory (see Section 4.5). If a send operation using a general datatype type1 is matched by a receive operation using a general datatype type2 then the sequence of (nonempty) basic type entries in each type map should be identical. However, the sequences of displacements may differ. Thus, type matching is defined by the type map, and does not depend on the construction of the datatype.

Example:

```
...
MPI_TYPE_CONTIGUOUS( MPI_REAL, 2, type2)
MPI_TYPE_CONTIGUOUS( MPI_REAL, 4, type4)
MPI_TYPE_CONTIGUOUS( type2, 2, type22)
```

A message sent with datatype type4 can be received with datatype type22.

A datatype used in a receive operation may not have overlapping 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.

The value returned in count by a call to MPI_GET_COUNT(status, count) is equal to the number of basic elements received by the operation that set status; this is the number of nonempty entries in the type map used to create the message. Consider again the datatypes defined in the last example. If a message is sent using either datatype type22 or type4 and is received using either of these two datatypes, then the number of elements received will be four.

Discussion:
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 prevents the obvious optimization of copying the structure, together with the padding, as one contiguous block (the additional communication overhead of sending the padding is likely to be smaller then the computation overhead of disassembling, next reassembling the structure components). However, this optimization is not possible in general, since it is not known when the send is executed that the receive is going to store the incoming message in a structure of the same type; also, across heterogeneous machines, similarly defined structures may have a different memory layout. In a homogeneous environment, when communication is known to involve structures with the same layout, one can avoid assembly and disassembly by sending the data using the datatype MPI_BYTE. This problem may be addressed by future extensions of MPI.

4.13.6 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_STRUC( 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_communicator     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++);
MPI_address( (void *)particle[0].b, disp);
for (i=0; i <2; 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++);
MPI_address( (void *)particle[0].b, disp1++);
MPI_address( (void *)(particle+1), disp1);
for (i=0; i <3; 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++);
MPI_address( (void *)(particle+1), disp2);
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);
```

4.13.7 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 start 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_STRUC are absolute addresses (computed by MPI_ADDR), 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 start and datatype, then either start = MPI_BOTTOM and datatype is a handle to an absolute datatype, or start 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.

Discussion:
It is not expected that MPI implementations will be able to detect erroneous, "out of bound" displacements – unless those overflow the user address space – since the MPI call may not know the extent of the arrays and records in the host program.

Integer arguments of type constructor functions that represent displacements should be of a type that can be used to represent an address on the host system (e.g., 64 bits on a 64 bit address machine).

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.

4.14 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 start argument that is not MPI_BOTTOM can be reduced to communication with start = MPI_BOTTOM by creating a suitable displaced new datatype. Thus:

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

```
Type[1] = datatype
MPI_ADDRESS( start, 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 start = 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(handle, datatype, source-dest, tag, comm, op-mode, persistence)

OUT	handle	new communication object (handle)
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

4.14.1 Persistent communication objects

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

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

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

The functions MPI_START and MPI_COMMOBJ_FREE are primitive.

4.14.2 Nonblocking communication initiation

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

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

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

4.14.3 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-handles argument of length one.

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

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

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

4.14.4 Blocking communication

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

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

4.14.5 Probe and cancel

The functions MPI_PROBE and MPI_CANCEL are primitive.

4.14.6 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.

4.14.7 send-receive and exchange

MPI_SENDRECV(send_start, send_count, send_type, dest, recv_start, recv_count, recv_type, source, tag, comm, status) is

```
MPI_ISEND(handle[0], send_start, send_count, send_type, dest, tag, comm)
MPI_IRECV(handle[1], recv_start, recv_count, recv_type, source, tag, comm)
MPI_WAITALL(2, handle, status_array)
status = status_array[1]
```

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

The function MPI_EXCHANGE 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_EXTENT.

4.14.8 Derived datatypes

We have outlined in Section 4.13.1 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 and MPI_ADDRESS are primitive.

4.14.9 Summary

In order to implement the MPI point-to-point communication one needs 16 functions; about half of those are trivial macros:

1. MPI_COMM_INIT
2. MPI_START
3. MPI_WAITALL
4. MPI_TESTANY
5. MPI_PROBE
6. MPI_CANCEL
7. MPI_GET_SOURCE
8. MPI_GET_TAG
9. MPI_GET_LEN
10. MPI_PROBE_LEN
11. MPI_IS_CANCELLED
12. MPI_TYPE_STRUCT
13. MPI_TYPE_EXTENT
14. MPI_ADDRESS
15. MPI_TYPE_FREE
16. MPI_COMM_FREE

On the other hand, we expect most implementations to use more primitives, in order to reduce communication overheads, reduce buffering, and improve error reporting.

Section 5

Collective Communication

5.1 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 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 (or Gather) data from all members to all members of a group (also called complete exchange or all-to-all).

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

The syntax and semantics of the collective operations are defined so as to be consistent with the syntax and semantics of the point-to-point operations. A collective operation is executed by having all processes in the group call the communication routine, with matching parameters. One of the key parameters is a communicator that defines the group of participating processes and provides a context for the operation. The reader is referred to chapter 4 for information concerning communication buffers and their manipulations and type matching rules; and to chapter 3 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

[1]Version of September 13, 1993

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. A more detailed discussion of the correct use of the collective routines can be found at the end of this chapter.

Discussion: The collective operations do not accept a message tag parameter. The rationale for not using tags is that the need for distinguishing collective operations with the same context seldom arises (since the operations are blocking); the tag field can be used by the point-to-point messages that implement the collective communication.

5.2 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 parameter. Instead, there is a communicator parameter. In this chapter a communicator can be thought of as a group identifier merged with a context.

5.3 Barrier synchronization

MPI_BARRIER(comm)

IN	comm	communicator handle

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.

5.4 Data move functions

Figure 5.1 illustrates the the different collective move functions supported by MPI.

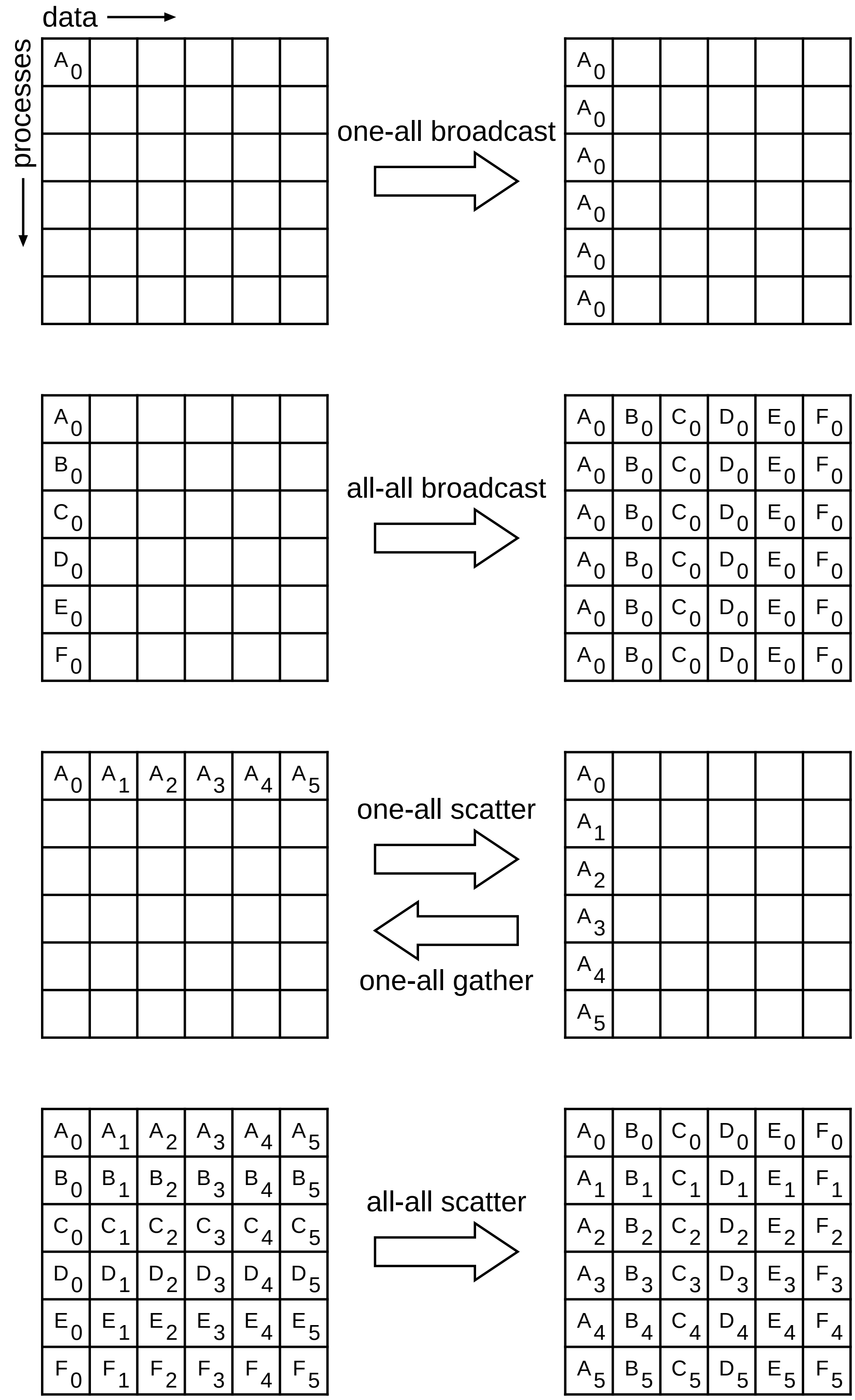

Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the one-all broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

5.4.1 Broadcast

MPI_BCAST(buffer, cnt, type, root, comm)

IN/OUT	buffer	starting address of buffer
IN	cnt	number of entries in buffer
IN	type	data type of buffer (possibly general)
IN	root	rank of broadcast root
IN	comm	communicator handle

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 cnt, type, comm, and root. On return the contents of the buffer of the process with rank root is contained in the buffer of the calling process.

5.4.2 Gather

MPI_GATHER(sendbuf, sendcnt, sendtype, recvbuf, maxrecvcnt, recvcnts, recvtype, root, comm)

IN	sendbuf	starting address of send buffer
IN	sendcnt	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer – significant only at root
IN	maxrecvcnt	maximum number of elements in receive buffer – significant only at root
OUT	recvcnts	integer array of size MPI_GSIZE returning the number of elements sent by each processor – 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

Each process (including the root process) sends the contents of its send buffer to the root process. The root process places all the incoming messages in the locations specified by the recvbuf and recvtype. The receive buffer is ignored for all non-root processes. The receive buffer of the root process is assumed contiguous and partitioned into MPI_GSIZE consecutive blocks, The data sent from process with rank i is stored in the i-th block. sendcnt can be different for each group member and these values, which are the size of the blocks, are returned in the array recvcnts on the root process. recvcnts[i] = sendcnt on the process with rank i. The function is called with the same values for sendtype, root, and comm at all participating processes.

5.4.3 Scatter

MPI_SCATTER(sendbuf, sendcnts, sendtype, recvbuf, maxrecvcnt, recvcnt, recvtype, root, comm)

IN	sendbuf	address of send buffer – significant only at root
IN	sendcnts	integer array of size MPI_GSIZE specifying the number of elements to send to each processor – significant only at root.
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer.
IN	maxrecvcnt	maximum number of elements in receive buffer (integer)
OUT	recvcnt	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

The root process sends the `i`-th portion of its send buffer to the process with rank `i`; each process (including the root process) stores the incoming message in its receive buffer. The send buffer of the root process is assumed contiguous and partitioned into MPI_GSIZE consecutive blocks. The `i`-th block consists of sendcnts[i] elements. The `i`-th block is sent to the process with rank i in the group and stored in its receive buffer. The routine is called by all members of the group using the same arguments for recvtype, root, and comm.

Note that MPI_SCATTER is the reverse operation to MPI_GATHER.

5.4.4 All-to-all broadcast

MPI_ALLCAST(sendbuf, sendcnt, sendtype, recvbuf, maxrecvcnt, recvcnts, recvtype, comm)

IN	sendbuf	starting address of send buffer
IN	sendcnt	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer.
IN	maxrecvcnt	maximum number of elements in receive buffer
OUT	recvcnts	integer array of size MPI_GSIZE returning the number of elements sent by each processor.
IN	recvtype	data type of receive buffer elements
IN	comm	communicator handle

Each process in the group broadcasts its entire send buffer to all processes (including itself); Each send buffer can have a different number of elements. Each process concatenates the incoming messages, in the order of the senders' ranks, and stores them in its receive buffer. The number of elements in the i-th sender's contribution is returned in recvcnts[i]. The routine is called by all members of the group using the same arguments for sendtype, maxrecvcnt, and comm.

MPI_ALLCAST is equivalent to n executions of MPI_BCAST, with each process once the root.

5.4.5 All-to-all scatter-gather

MPI_ALLTOALL(sendbuf, sendcnts, sendtype, recvbuf, maxrecvcnt, recvcnts, recvtype, comm)

IN	sendbuf	starting address of send buffer
IN	sendcnts	integer array of size MPI_GSIZE specifying the number of elements to send to each processor
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	maxrecvcnt	maximum number of elements in receive buffer
OUT	recvcnts	integer array of size MPI_GSIZE returning the number of elements received from each processor
IN	recvtype	data type of receive buffer elements
IN	comm	communicator handle

The send buffer of each process is partitioned into MPI_GSIZE consecutive blocks. The number of elelments in the i-th block is given by sendcnts[i]. Each process in the group sends the i-th block of its send buffer to the process with rank i (itself included). Each process concatenates the incoming messages, in the order of the senders' ranks, and stores them in its receive buffer. The number of elelments in the received blocks are returned in the array recvcnts such that process with rank i gets recvcnts[k] equal to sendcnts[i] on process with rank k. The routine is called by all members of the group using the same arguments for sendtype, and comm.

An all-to-all scatter-gather is the equivalent of n scatters (or n gathers) executed with each process being once the root.

5.5 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

global max and who (rank) has it

global min and who (rank) has it

user defined (associative) operation

user defined (associative and commutative) operation

5.5.1 Reduce

MPI_REDUCE(sendbuf, recvbuf, cnt, type, op, root, comm)

IN	sendbuf	address of send buffer
OUT	recvbuf	address of receive buffer – significant only at root
IN	cnt	number of elements in input buffer (integer)
IN	type	data type of elements of input buffer
IN	op	operation
IN	root	rank of root process (integer)
IN	comm	communicator handle

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 point-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 one of those allowed for operands of op. For all operations except MPI_MINLOC and MPI_MAXLOC the number and type of elements in the send buffer are the same as for the receive buffers. For MPI_MINLOC and MPI_MAXLOC, the receive buffer will contain cnt elements of the same type as the elements in the input buffer, followed by cnt integers (ranks).

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 cnt, type, 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_AND	and (logical or bit-wise integer)
MPI_OR	or (logical or bit-wise integer)
MPI_XOR	xor (logical or bit-wise integer)
MPI_MAXLOC	maximum value and rank of process with maximum value (rank of first process with maximum value, in case of ties)
MPI_MINLOC	minimum value and rank of process with minimum value (rank of first process with minimum value, in case of ties)

All operations, with the exception of MPI_MAXLOC and MPI_MINLOC return a value which has the same datatype as the operands. Each operand of MPI_MAXLOC and MPI_MINLOC can be thought as a pair `(v, i)`: `i` is the rank of the calling process that is passed implicitly, and `v` is the value that is explicitly passed to the call. MPI_MAXLOC and MPI_MINLOC return (explicitly) a pair `(value, rank)`.

When MPI_MINLOC or MPI_MAXLOC are invoked, the input buffer should contain m elements of the same type to which the operation MPI_MIN or MPI_MAX can be applied. The operation returns at the root m elements of the same type as the inputs, followed by m integers (ranks). The output buffer should be defined accordingly.

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 this operation is associative and commutative.

A similar definition can be given for MPI_MINLOC.

Discussion:
We define MPI_MINLOC to return a vector of values, followed by a vector of ranks. The alternative is for MPI_MINLOC to return a vector of (value, rank) pairs, i.e., a vector of structures. This second choice is less convenient for Fortran. Another alternative is to have MPI_MINLOC return two output buffers, but then it need be invoked differently than the other operations.

The computation can still be pipelined, provided that the location of the first rank entry in the output buffer can be computed up front.

Implementation note:
The operations can be applied to operands of different types, in different calls: e.g., MPI_SUM may require an integer sum in one call, and a complex sum in another. Since we require that all elements be of the same datatype, it is not necessary to store a full signature with each buffer: It is only necessary to store the datatype of the elements when all elements are of the same type, and store a flag indicating that the buffer is not homogeneous, otherwise.

Missing:

Need to define the types compatible with each operation. This includes MPI_BYTE for the logical operations, and whatever Fortran/C allow for all operations.

MPI_USER_REDUCE(sendbuf, recvbuf, cnt, type, function, root, comm)

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

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 `f( invec, inoutvec, *len)`. Both invec and inoutvec are arrays with *len entries. The function computes point-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.

MPI_USER_REDUCEA(sendbuf, recvbuf, cnt, type, function, root, comm)

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

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.

Implementation note:

The code for MPI_USER_REDUCEA can be used to provide an identical implementation for MPI_USER_REDUCE.

Discussion:

The addition of the third parameter, *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, cnt, type, op, comm)

IN	sendbuf	starting address of send buffer
OUT	recvbuf	starting address of receive buffer
IN	cnt	number of elements in input buffer (integer)
IN	type	data type of elements of input buffer
IN	op	operation
IN	comm	communicator handle

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, cnt, type, function, comm)

IN	sendbuf	starting address of send buffer
OUT	recvbuf	starting address of receive buffer
IN	cnt	number of elements in input buffer (integer)
IN	type	data type of elements of input buffer
IN	function	user defined function
IN	comm	communicator handle

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, cnt, type, function, comm)

IN	sendbuf	starting address of send buffer
OUT	recvbuf	starting address of receive buffer
IN	cnt	number of elements in input buffer (integer)
IN	type	data type of elements of input buffer
IN	function	user defined function
IN	comm	communicator handle

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

Implementation note:

The allreduce operations can be implemented as a reduce, followed by a broadcast. However, a direct implementation can lead to better performance.

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, distcnts, type, op, comm)

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

MPI_REDUCE_SCATTER first does a componentwise reduction on vectors provided by the processes. Next, the resulting vector of results is split into disjoint segments, where segment `i` has length discnts[i]; the `i`-th segment is sent to process with rank `i`.

This routine is functionally equivalent to: A MPI_REDUCE operation function with cnt equal to the sum of distcnts[i] followed by MPI_SCATTER with mpiargsendcnts equal to mpiargdistcnts. However, it can be implemented to run substantially faster.

MPI_USER_REDUCE_SCATTER(sendbuf, recvbuf, distcnts, type, function, comm)

IN	sendbuf	starting address of send buffer
OUT	recvbuf	starting address of receive buffer
IN	distcnts	integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	type	data type of elements of input buffer
IN	function	user defined function
IN	comm	communicator handle

Same as the MPI_REDUCE_SCATTER operation function except that the user specifies the reduction operation as in MPI_USER_REDUCE.

MPI_USER_REDUCE_SCATTERA(sendbuf, recvbuf, distcnts, type, function, comm)

IN	sendbuf	starting address of send buffer
OUT	recvbuf	starting address of receive buffer
IN	distcnts	integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	type	data type of elements of input buffer
IN	function	user defined function
IN	comm	communicator handle

Same as the MPI_USER_REDUCE_SCATTER operation function except that the user specified reduction operation only need to be associative as in MPI_USER_REDUCEA.

Implementation note:

The REDUCE_SCATTER operations can be implemented as a reduce, followed by a scatter. However, a direct implementation can lead to better performance.

5.5.2 Scan

MPI_SCAN(sendbuf, recvbuf, cnt, type, op, comm)

IN	sendbuf	starting address of send buffer
IN	recvbuf	starting address of receive buffer
IN	cnt	number of elements in input buffer (integer)
IN	type	data type of elements of input buffer
IN	op	operation
IN	comm	communicator handle

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, cnt, type, function, comm)

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

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, cnt, type, function, comm)

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

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

Implementation note:
MPI_USER_SCAN can be implemented as MPI_USER_SCANA.

5.6 Correctness

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

```
/* Example A */
switch(MPI_rank(comm,rank); rank)
   {
   case 0: { MPI_bcast(var1, cnt, type, 0, comm);
             MPI_send(var2, cnt, type, 1, tag, comm);
             break;
           }
   case 1: { MPI_recv(var2, cnt, type, 0, tag, comm);
             MPI_bcast(var1, cnt, 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 blocks on the receive and never executes the broadcast.

The following example is correct, but nondeterministic:

```
/* Example B */
switch(MPI_rank(comm,rank); rank)
   {
    case 0: { MPI_bcast(var1, cnt, type, 0, comm);
              MPI_send(var2, cnt, type, 1, tag, comm);
              break;
            }
    case 1: { MPI_recv(var2, cnt, type, MPI_SOURCE_ANY, tag, comm);
              MPI_bcast(var1, cnt, type, 0, comm);
              MPI_recv(var2, cnt, type, MPI_SOURCE_ANY, tag, comm);
              break;
            }
    case 2: { MPI_send(var2, cnt, type, 1, tag, comm);
              MPI_bcast(var1, cnt, 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 after the broadcast. Process 1 receives before and after the broadcast, with a wildcard source parameter.

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.

Discussion:
An alternative design is to require that all collective communication calls are synchronizing. In this case, the second program is deterministic and only the first execution may occur. This will make a difference only for collective operations where not all processes both send and receive (broadcast, reduce, scatter, gather).

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, cnt, type, 0, comm);
 MPI_bcast(var2, cnt, 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 implementer 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.

An alternative choice is to require that a communicator is quiescent when used in a collective communication: No messages using a context can be pending at a process when this process starts executing a collective communication with this context, nor can any new message with this context arrive during the execution of this collective communication, unless they were sent as part of the execution of the collective call itself.

This approach has the advantage of simplifying the layering of collective communications on top of point-to-point communication (no need for hidden contexts). Also, it imposes on the collective collective communication library the same restrictions that hold for any other collective library. It has the disadvantage of restricting the use of collective communications.

The question is whether we want to view the collective communication operations as part of the basic communication services of MPI, or whether we want to see them as a library layered on top of these basic services.

Section 6

More on Groups, Contexts, and Communicators

6.1 Introduction

We start by discussing intra-communication in full detail; then we discuss inter-communication, which builds on the data structures and requirements of the intra-communication sections. We follow with discussion of formalizations of the loosely synchronous model of computing (vis a vis message passing) and offer examples.

6.2 Communicators

All MPI communication (both point-to-point and collective) functions use **communicators** to provide a specific scope (context and group specifications) for the communication. In short, communicators bring together the concepts of group and context; (furthermore, to support implementation-specific optimizations, and virtual topologies, they "cache" additional information opaquely). The source and destination of a message is identified by the rank of that process within the group; no a priori membership restrictions on the process sending or receiving the message are implied. For collective communication, the communicator specifies the set of processes that participate in the collective operation. 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.

Raison d'être for Contexts and Communicators Within a communicator, one or more contexts are cached. We want to make it possible for libraries quickly to achieve additional safe communication space without MPI-communicator-based synchronization. The only way to do this is to provide a means to preallocate many contexts, and bind them locally, as needed. This choice weakens the overall, inherent "safety" of MPI, if programmed in this way, but provides added performance which library designers will demand. We are convinced that the means we have chosen provides a good compromise between efficiency and safety.

---

[1]Version of August 13, 1993

6.2.1 Predefined Communicators

Initial communicators defined once MPI_INIT has been called are as follows:

- MPI_COMM_ALL, A communicator of all processes in the user program including a host process (if any).
- MPI_COMM_PEER, A communicator of all processes excluding a host process (if any). Note that in the absence of a host process this communicator will not be just the same as MPI_COMM_ALL since it will contain a different context.A

Discussion: In the host process, if there is one, what should MPI_COMM_PEER be? There are two consistent possibilities: a null communicator; a communicator in which the group contains only the host process.

MPI implementations are required to provide these communicators. The groups corresponding to these communicators do not appear as pre-defined constants, but they may be accessed using MPI_COMM_GROUP (see below).

6.2.2 Group Management

This section describes the manipulation of process groups in MPI. The execution of the these operations do not require interprocess communication.

6.2.3 Group Accessors

MPI_GROUP_SIZE(group, size)

IN	group	handle to group object.
OUT	size	is the integer number of processes in the group.

MPI_GROUP_RANK(group, rank)

IN	group	handle to group object.
OUT	rank	is the integer rank of the calling process in group, or MPI_UNDEFINED if the process is not a member.

MPI_TRANSLATE_RANKS (group_a, n, ranks_a, group_b, ranks_b)

IN	group_a	handle to group object "A"
IN	n	number of ranks in ranks_a array
IN	ranks_a	array of zero or more valid ranks in group "A"
IN	group_b	handle to group object "B"
OUT	ranks_b	array of corresponding ranks in group "B," MPI_UNDEFINED when no correspondence exists.

6.2.4 Group Constructors

MPI_LOCAL_GROUP_UNION(group1, group2, group_out)

IN	group1	first group object handle
IN	group2	second group object handle
OUT	group_out	group object handle

MPI_LOCAL_GROUP_INTERSECT(group1, group2, group_out)

IN	group1	first group object handle
IN	group2	second group object handle
OUT	group_out	group object handle

MPI_LOCAL_GROUP_DIFFERENCE(group1, group2, group_out)

IN	group1	first group object handle
IN	group2	second group object handle
OUT	group_out	group object handle

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 which are also in the second group

difference all elements of the first group which are not in the second group

Note that for these operations the order of processes in the output group is determined first by order in the first group (if possible) and then by order in the second group (if necessary).

MPI_LOCAL_SUBGROUP(group, n, ranks, new_group)

IN	group	handle to group object
IN	n	number of elements in array ranks (and size of new_group)
IN	ranks	array of integer ranks in group to appear in new_group.
OUT	new_group	new group derived from above, preserving the order defined by ranks.

The function MPI_LOCAL_SUBGROUP creates a group `new_group` which is a subgroup of `old_group` such that the process with rank `i` in `new_group` is the process with rank `ranks[i]` in `old_group`, and the size of `new_group` is `n`. Each of the `n` elements of `ranks` must be a valid rank in `old_group` and all elements must be distinct, else the program is erroneous. If no ranks are specified, new_group has no members.

MPI_LOCAL_EXCL_SUBGROUP(group, n, ranks, new_group)

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

The function MPI_LOCAL_EXCL_SUBGROUP creates a group `new_group` which is a subgroup of `old_group` such that `new_group` contains all process in `old_group` except the `n` processes with ranks `ranks[i] , i = 0, ..., n - 1`, and the ordering of processes in `new_group` is identical to the ordering in `old_group`. Each of the `n` elements of `ranks` must be a valid rank in `old_group` and all elements must be distinct, else the program is erroneous. If no ranks are specified, new_group is identical to group.

MPI_LOCAL_SUBGROUP_RANGES(group, n, ranges, new_group)

IN	group	handle to group object
IN	n	number of triplets in array `ranges`
IN	ranges	a one-dimensional array of integer triplets: pairs of ranks (form: beginning through end, inclusive) to be included in the output group new_group, plus a constant stride (often 1 or -1).
OUT	new_group	new group derived from above, preserving the order defined by ranges.

If any of the rank sets overlap, then call is erroneous. If no ranges are specified, then the output group has no members.

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_LOCAL_SUBGROUP.

MPI_LOCAL_SUBGROUP_EXCL_RANGES(group, n, ranges, new_group)

IN	group	handle to group object
IN	n	number of elements in array ranks (and size of new_group)
IN	ranges	a one-dimensional array of (three integer) consisting of pairs of ranks (form: beginning through end, inclusive) to be excluded from the output group new_group, plus a constant stride (often 1 or -1).
OUT	new_group	new group derived from above, preserving the order defined by ranges.

If any of the rank sets overlap, then call is erroneous. If there are no ranges specified, the output group is the same as the original group.

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_LOCAL_EXCL_SUBGROUP.

Discussion: Please propose additional subgroup functions, before the second reading...Virtual Topologies support?

Discussion: What do people think about these local operations? More? Less? Note: these operations do not explicitly enumerate ranks, and therefore are more scalable if implemented efficiently...

MPI_GROUP_DUP(group, new_group)

IN	group	extant group object handle
OUT	new_group	new group object handle

MPI_GROUP_DUP duplicates a group with all its cached information, replacing nothing. This function is essential to the support of virtual topologies.

6.2.5 Group Destructors

MPI_GROUP_FREE(group)

IN	group	frees group previously defined.

This operation frees a handle group which is not currently bound to a communicator. It is erroneous to attempt to free a group currently bound to a communicator.

6.3 Context Management

6.3.1 Context Allocation

MPI_CONTEXTS_ALLOC(comm, n)

IN	comm	a communicator
IN	n	number of contexts to allocate

This function allocates an array of contexts, which are unique within the group bound to the communicator `comm`, and stores them in the unused context store of `comm`. The function call is global (i.e. collective) in `comm`.

Note: MPI provides special contextual space to the collective operations (including MPI_CONTEXTS_ALLOC) so that, despite any on-going of point-to-point communication on the communicator argument, the collcetive operation can execute safely. MPI collective functions will have to lock out multiple threads, so it will evidently have capabilities unavailable to the user program.

Discussion:

We have to retain the chicken-and-egg aspect of MPI (*i.e.*, use a communicator to get context(s) or a communicator), to get thread safety. Yet, we want libraries to control their own fate regarding safety, not to rely on the caller to provide a quiescent context. We achieve this by adding the *quiescent property* for MPI collective communication functions. We must, in fact, push this requirement to the collective chapter, but we demonstrate here why our particular collective routines need this property. In a multi-threaded environment, it is clear that each temporally overlapping call to a collective operation must be with a different communicator. If this has not been made explicit, it must be.

One can have on-going point-to-point and collective communications on a single communicator. A context is defined to be sufficiently powerful to keep both point-to-point and collective operations distinct. Hence, it is always safe to call MPI_COMM_MAKE and MPI_CONTEXTS_ALLOC, even if pending asynchronous point-to-point operations are on-going, or messages have not been received but are on the receipt queue. With these rules, no quiescent communicator is required in order to get new contexts. We have added demands on the MPI implementation while making contexts opaque to make this simpler to realize without saying how it must be done.

In summary, libraries have to get the communicator as a base argument to retain thread safety, but they can always safely get communication contexts to do further work. The concept of quiescence is banished to be a small detail of implementation, rather than a central tenet of library design. Users must still worry about temporal safety, which is not guaranteed by contexts alone (see example below in section 6.8.6).

6.3.2 Context Deallocation

MPI_CONTEXTS_FREE(comm, n)

IN	comm	a communicator
IN	n	number of contexts to free

This function frees (i.e., returns to the MPI system) up to `n` contexts from the unused context store of the communicator `comm`, which were previously allocated by calling MPI_CONTEXTS_ALLOC (below). The function call is local but **must** be called as though it is collective.

6.4 Intra-Communication Communicator Management

6.4.1 Communicator Accessors

The following are all local (non-communicating) operations.

MPI_COMM_SIZE(comm, size)

IN	comm	handle to communicator object.
OUT	size	is the integer number of processes in the group of comm.

MPI_COMM_RANK(comm, rank)

IN	comm	handle to communicator object.
OUT	rank	is the integer rank of the calling process in group of comm, or MPI_UNDEFINED if the process is not a member.

MPI_COMM_GROUP(comm, group)

IN	comm	communicator object handle
OUT	group	group object handle corresponding to `comm`

MPI_COMM_CONTEXTS(comm, n)

IN	comm	handle to communicator object.
OUT	n	the number of contexts available in the unused context store associated with `comm`

6.4.2 Communicator Constructors

MPI_COMM_DUP(comm, new_comm)

IN	comm	communicator object handle
OUT	new_comm	communicator object handle

MPI_COMM_LOCAL_DUP(comm, new_comm)

IN	comm	communicator object handle
OUT	new_comm	communicator object handle

MPI_COMM_COLL_DUP(comm, new_comm)

IN	comm	communicator object handle
OUT	new_comm	communicator object handle

Each of these three functions duplicates the existing communicator `comm` with all its cached information, replacing just the context, and creates the new communicator `new_comm`.

MPI_COMM_DUP is a collective operation which will take a context from the unused context store associated with comm if such a context is available, else it will allocate a new context for the communicator new_comm. MPI_COMM_LOCAL_DUP is a local operation which will take a context from the unused context store associated with comm if such a context is available, else it will fail and return the null communicator MPI_COMM_NULL in new_comm. The function is local but **must** be called as thought it is collective. MPI_COMM_COLL_DUP is a collective operation which will allocate a new context for the communicator new_comm. See also the considerations of program correctness below.

MPI_COMM_MAKE(sync_comm, comm_group, comm_new)

IN	sync_comm	Communicator within which context is unique
IN	comm_group	Group which is a subset of the group of sync_comm
OUT	comm_new	The new communicator.

MPI_COMM_LOCAL_MAKE(sync_comm, comm_group, comm_new)

IN	sync_comm	Communicator within which context is unique
IN	comm_group	Group which is a subset of the group of sync_comm
OUT	comm_new	The new communicator.

MPI_COMM_COLL_MAKE(sync_comm, comm_group, comm_new)

IN	sync_comm	Communicator within which context is unique
IN	comm_group	Group which is a subset of the group of sync_comm
OUT	comm_new	The new communicator.

Each of these three functions creates a new communicator new_comm which combines a context unique within the group of the existing communicator sync_comm, and the group comm_group. No cached information propagates from sync_comm to comm_new. For processes in sync_group which are not in comm_group, these functions return comm_new as the null communicator MPI_COMM_NULL. If the set of processes in comm_group is not a subset of, or equal to, the set of processes in the group of sync_group then the program is erroneous. This error condition can be checked. If the group comm_group is not identical in all processes calling the function, then the program is again erroneous. This error condition cannot be checked. MPI_COMM_MAKE is a collective operation which will take a context from the unused context store associated with sync_comm if such a context is available, else it will allocate a new context for the communicator new_comm. MPI_COMM_LOCAL_MAKE is a local operation which will take a context from the unused context store associated with sync_comm if such a context is available, else it will fail and return the null communicator MPI_COMM_NULL in new_comm. The function is local but **must** be called as though it iscollective. MPI_COMM_COLL_MAKE is a collective operation which will allocate a new context for the communicator new_comm. See also the considerations of program correctness below.

MPI_COMM_COLL_SUBSET(comm, colour, key, comm_new)

IN	comm	An existing communicator
IN	colour	control of subset assignment
IN	key	control of rank assigment
OUT	comm_new	A new communicator.

Discussion: Since this function must be collective we do not need to consider whether it synchronises to allocate a new context when the unused context store is empty. However we should probably say whether it takes a context from the store or not. I have said that it allocates a new context, as opposed to using the store, but we may wish to change this.

6.4.3 Communicator Destructors

MPI_COMM_FREE(comm)

IN	comm	the communicator to be destroyed.

This function destroys the communicator comm. The function call is local but **must** be called as though it is collective. If the context of comm was pre-allocated by the user calling MPI_CONTEXTS_ALLOC then it is returned to the context store of the communicator from which it was taken. Otherwise the context is returned to the system. It is erroneous to free a communicator if contexts from the context sore of the communicator are in use. If comm was created by calling one of MPI_COMM_MAKE, MPI_COMM_LOCAL_MAKE or MPI_COMM_COLL_MAKE then the underlying group is not destroyed. Otherwise the group is destroyed.

Discussion: Do we want a collective version of this function? .

6.5 Inter-Communication Communicator Management

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs which contain user-level servers. It also describes a name service which simplifies writing programs containing inter-communication.

6.5.1 Discussion

Definitions of Inter-Communication and Inter-Communicators

All point-to-point communication described thus far has involved communication between processes that are members of the same group. The source process in send or the destination process in receive (the "target" process) is specified using a (communicator, rank) pair. The target process is that process with the given rank within the group of the given communicator. 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 different 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 which uses the services of one or more servers. In these applications it is again most natural to specify the target process by rank within the target group. 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 the remote group.

One additional needed concept is the "group leader." The process with rank 0 in a process group is designated "group leader." This concept is used in support of user-level servers, and elsewhere.

Properties of Inter-Communication and Inter-Communicators

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax is the same for both inter- and intra-communication. The same communicator can be used for both send and receive operations.
- A target process is addressed by its rank in the remote group.
- 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.
- Once constructed, the remote group of an inter-communicator may not be changed. Communication with any process outside of the remote group is not allowed.

The routine MPI_COMM_STAT() 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 inquiry routines (defined above). Inter-communicators cannot be used as input to any of the local constructor routines for intra-communicators or the context management routines.

Intercommunicator Constructors and Destructors

Construction of an inter-communicator requires two separate collective operations (one in the local group and one in the remote group) and a point-to-point operation between the two group leaders. These operations may be performed with explicit synchronization of the two groups by calling MPI_COMM_PEER_MAKE(). The explicit synchronization can cause deadlock in modular programs with cyclic communication graphs. So, the local and remote operations can be decoupled and the construction performed "loosely synchronously" by calling the two routines MPI_COMM_PEER_MAKE_START() and MPI_COMM_PEER_MAKE_FINISH().

Discussion: MPI_COMM_PEER_MAKE_START() and MPI_COMM_PEER_MAKE_FINISH() are both collective operations in the local group. They may leave a non-blocking send and receive active between the two calls, where the group leaders exchange local communicator information as necessary. However, they are not a non-blocking collective operation. These routines can

construct multiple inter-communicators with a single call. This improves performance by allowing amortization of the synchronization overhead.

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

Support for User-Level Servers

We consider the primary feature of user-level servers that can require additional support is that the server cannot a priori know the identification of the clients, whereas the clients must a priori know the identification of the servers. In addition, a user-level server is a dedicated process group which, after some initialization, provides a given service until termination.

The support for user-level servers takes into account the prevailing view that all processes (possibly excepting a host process) are initially equivalent members of the group of all processes. This group is described by pre-defined intra-communicator MPI_COMM_ALL. The user splits this group such that processes in each parallel server are placed within a specific sub-group. The non-server processes are placed in a group of all non-servers. Provided that the user can determine the ranks of the server group leaders (*i.e.,* rank zero) and assign some tags for clients to send a message to the group leaders, then a group leader can at any time notify a server that it wishes to become a client.

MPI provides a routine, MPI_COMM_SPLITL(), that splits a parent group, creates sub-groups (intra-communicators) according to supplied keys, and returns the rank of each sub-group leader (relative to the parent group). This allows a process that does not know about a sub-group to contact that sub-group via the sub-group leader, using the parent communicator. The keys may be used as unique tags. This information may also be used as input to MPI_COMM_PEER_MAKE(), for example.

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 synchronizing version is provided by routine MPI_COMM_NAME_MAKE(). A loosely synchronous version is provided by routines MPI_COMM_NAME_MAKE_START() and MPI_COMM_NAME_MAKE_FINISH().

6.5.2 Communicator Accessors

MPI_COMM_STAT(comm, status)

IN	comm	communicator
OUT	status	integer status

This local routine allows the calling process to determine if a communicator is an inter-communicator or an intra-communicator. This returns the status of communicator comm. Valid status values are MPI_INTRA, MPI_INTER, MPI_INVALID.

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 MPI_UNDEFINED
MPI_COMM_CONTEXTS()	erroneous

Discussion: Should there be any other status values?

Discussion: Does this routine really belong under inter-communication?

6.5.3 Communicator Constructors

MPI_COMM_PEER_MAKE(my_comm, my_leader, peer_comm, peer_rank, tag, num_comms, new_comms)

IN	my_comm	local intra-communicator
IN	my_leader	rank of local group leader in my_comm
IN	peer_comm	"parent" intra-communicator
IN	peer_rank	rank of remote group leader in peer_comm
IN	tag	"safe" tag
IN	num_comms	number of new inter-communicators to construct
OUT	new_comms	array of new inter-communicators

This routine constructs an array of inter-communicators and stores it in new_comms. Intra-communicator my_comm describes the local group. Intra-communicator peer_comm describes a group that contains the leaders (*i.e.,* members with rank zero) of both the local and remote groups. Integer peer_rank is the rank of the remote leader in peer_comm. Integer tag is used to distinguish this operation from others with the same peer. Integer num_comms is the number of new inter-communicators constructed. This routine is collective in the local group and synchronizes with the remote group. Each of the inter-communicators produced providesinter-communication with the remote group.

MPI_COMM_PEER_MAKE_START(my_comm, my_leader, peer_comm, peer_rank, tag, num_comms, make_id)

IN	my_comm	local intra-communicator
IN	my_leader	rank of local group leader in my_comm
IN	peer_comm	"parent" intra-communicator
IN	peer_rank	rank of remote group leader in peer_comm
IN	tag	"safe" tag
IN	num_comms	number of new inter-communicators to construct
OUT	make_id	handle for MPI_COMM_PEER_MAKE_FINISH()

This starts off a _PEER_MAKE operation, returning a handle for the operation in "make_id." It is collective in my_comm. It does not wait for the remote group to do MPI_COMM_MAKE_START(). The "make_id" handle is conceptually similar to the communication handle used by non-blocking point-to-point routines. A make_id 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 again with the same peer_comm, peer_rank and tag, without calling MPI_COMM_MAKE_FINISH() to finish the first call.

MPI_COMM_PEER_MAKE_FINISH(make_id, new_comms)

IN	make_id	handle from MPI_COMM_PEER_MAKE_START()
OUT	new_comms	array of new inter-communicators

This completes a _PEER_MAKE operation, returning an array of num_comms new inter-communicators in new_comms. (Note that num_comms was specified in the corresponding call to MPI_COMM_PEER_MAKE_START().) This routine is collective in the my_comm of the corresponding call to MPI_COMM_PEER_MAKE_START(). It waits for the remote group to call MPI_COMM_PEER_MAKE_START() but does not wait for the remote group to call MPI_COMM_PEER_MAKE_FINISH().

6.5.4 Communicator Destructors

Inter-communicator objects are destroyed in the same fashion as intra-communiation objects, using MPI_COMM_FREE described above.

6.5.5 Support for User-Level Servers

Discussion: Does this routine really belong under inter-communication?

This collective routine is used to make it easier for processes to form sub-groups and contact user-level servers.

MPI_COMM_SPLITL(comm, key, nkeys, leaders, sub_comm)

IN	comm	extant intra-communicator to be "split"
IN	key	key for sub-group membership
IN	nkeys	number of keys (number of sub-groups)
OUT	leaders	ranks of sub-group leaders in comm
OUT	sub_comm	intra-communicator describing sub-group of calling process

This routine splits the group described by intra-communicator comm into nkeys sub-groups. Each calling process must specify a value of key in the range [0...(nkeys-1)]. Processes specifying the same key are placed in the same sub-group. Ranks of the leaders of each sub-group (relative to comm) are returned in integer array leaders. This routine returns a new intra-communicator, sub_comm, that describes the sub-group to which the calling process belongs.

6.5.6 Communicator Constructors using Name Service

MPI_COMM_NAME_MAKE(my_comm, name, num_comms, new_comms)

IN	my_comm	local intra-communicator
IN	name	"name known to both local and remote group leaders"
IN	num_comms	number of new inter-communicators to construct
OUT	new_comms	Array of new inter-communicators

This is the name-served equivalent of MPI_COMM_PEER_MAKE in which the caller need only know a name for the peer connection. The same name is supplied by both local and remote groups. The name is removed from the internal name-server database after both groups have completed MPI_COMM_NAME_MAKE().

MPI_COMM_NAME_MAKE_START(my_comm, name, num_comms, make_id)

IN	my_comm	local intra-communicator
IN	name	"name" known to both local and remote group leaders
IN	num_comms	number of new inter-communicators to construct
OUT	make_id	handle for MPI_COMM_NAME_MAKE_FINISH()

MPI_COMM_NAME_MAKE_FINISH(make_id, new_comms)

IN	make_id	handle from MPI_COMM_NAME_MAKE_START()
OUT	new_comms	array of new inter-communicators

These are the START/FINISH versions of MPI_COMM_NAME_MAKE(). They have synchronization properties analogous to the corresponding _PEER_ routines.

6.5.7 Further Discussion

Security and Performance Issues

The routines in this section do not introduce insecurity into the basic usage of MPI. Specifically, they do not allow contexts to be bound in multiple usable communicators.

The provision of inter-communication does not adversely affect the (potential) performance of intra-communication.

"Under the Hood" Implementation Notes

A possible implementation of a communicator contains a single group, a send context, a receive context, and a source (for the message envelope). This structure makes intra- and inter-communicators basically the same object.

The intra-communicator has the properties that: the send-context and the receive-context are identical; the process is a member of the group; the source is the rank of the process in the group.

The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process **P** in group G which has an inter-communicator `Cp`, and a process **Q** in group H which has an inter-communicator `Cq`. (Note that G and H do not have to be distinct.) The inter-communicators have the properties that: the send context of `Cp` is identical to the receive context of `Cq`, and is unique in H; the receive context of `Cp` is identical to the send context of `Cq`, and is unique in G; the group of `Cp` is H; the group of `Cq` is G; the source of `Cp` is the rank of **P** in G, which is the group of `Cq`; the source of `Cq` is the rank of Q in H, which is the group of `Cp`.

It is easy to see that in terms of these fields, the intra-communicator is a special case of the inter-communicator. It has $\mathcal{G} = \mathcal{H}$ and both contexts the same. This ensures that the point-to-point communication implementation for intra-communication and inter-communication can be identical.

6.6 Cacheing

MPI provides a "cacheing" facility that allows an application to attach arbitrary pieces of information, called *attributes*, to context, group, and communicator descriptors; it provides this facility to user programs as well. Attributes are local to the process and are not included if the descriptor were somehow sent to another process[1]. This facility is intended to support optimizations such as saving persistent communication handles and recording topology-based decisions by adaptive algorithms. However, attributes are propagated intentionally by specific MPI routines.

To summarize, cacheing is, in particular, the process by which implementation-defined data (and virtual topology data) is propagated in groups and communicators.

Discussion: Attribute propagation must be discussed carefully.

[1] The deletion of flatten/unflatten makes this point moot.

6.6.1 Functionality

MPI provides the following services related to cacheing. They are all process-local.

MPI_ATTRIBUTE_ALLOC(n,handle_array,len)

IN	n	number of handles to allocate
OUT	handle_array	pointer to array of opaque attribute handling structure
OUT	len	length of each opaque structure

Allocates a new attribute, so user programs and functionality layered on top of MPI can access attribute technology.

MPI_ATTRIBUTE_FREE(handle_array,n)

IN	handle_array	array of pointers to opaque attribute handling structures
IN	n	number of handles to deallocate

Frees attribute handle.

MPI_GET_ATTRIBUTE_KEY(keyval)

OUT	keyval	Provide the integer key value for future storing.

Generates a new cache key.

MPI_SET_ATTRIBUTE(handle, keyval, attribute_val, attribute_len, attribute_destructor_routine)

IN	handle	opaque attribute handle
IN	keyval	The integer key value for future storing.
IN	attribute_val	attribute value (opaque pointer)
IN	attribute_len	length of attribute (in bytes)
IN	attribute_destructor_routine	What one calls to get rid of this attribute later

Stores attribute in cache by key.

MPI_TEST_ATTRIBUTE(handle,keyval,attribute_ptr,len)

IN	handle	opaque attribute handle
IN	keyval	The integer key value for future storing.
OUT	attribute_ptr	void pointer to attribute, or NULL if not found
OUT	len	length in bytes of attribute, if found.

Retrieve attribute from cache by key.

MPI_DELETE_ATTRIBUTE(handle, keyval)

IN	handle	opaque attribute handle
IN	keyval	The integer key value for future storing.

Delete attribute from cache by key.

Example

Each attribute consists of a pointer or a value of the same size as a pointer, and would typically be a reference to a larger block of storage managed by the module. As an example, a global operation using cacheing to be more efficient for all contexts of a group after the first call might look like this:

```
static int gop_key_assigned = 0;    /* 0 only on first entry */
static int gop_key;        /* key for this module's stuff */

efficient_global_op (comm, ...)
void *comm;
{
  struct gop_stuff_type *gop_stuff;   /* whatever we need */
  void  *group = mpi_comm_group(comm);

  if (!gop_key_assigned)     /* get a key on first call ever */
  { gop_key_assigned = 1;
    if ( ! (gop_key = mpi_Get_Attribute_Key()) ) {
      mpi_abort ("Insufficient keys available");
    }
  }
  if (mpi_Test_Attribute (mpi_group_attr(group),gop_key,&gop_stuff))
  { /* 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.
     */
    gop_stuff = /* malloc a gop_stuff_type */

    /* ... fill in *gop_stuff with whatever we want ... */

    mpi_set_attribute (mpi_group_attr(group), gop_key, gop_stuff,
                               gop_stuff_destructor);
  }
  /* ... use contents of *gop_stuff to do the global op ... */
 }

 gop_stuff_destructor (gop_stuff)   /* called by MPI on group delete */
```

```
struct gop_stuff_type *gop_stuff;
{
  /* ... free storage pointed to by gop_stuff ... */
}
```

Discussion: The cache facility could also be provided for other descriptors, but it is less clear how such provision would be useful. It is suggested that this issue be reviewed in reference to Virtual Topologies.

6.7 Formalizing the Loosely Synchronous Model (Usage, Safety)

6.7.1 Basic Statements

When a caller passes a communicator (which 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 the creation of new communicators without synchronization (assuming preallocated contexts), 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.

6.7.2 Models of Execution

We say that a parallel procedure is *active* at 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 at 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.

Nonreentrant 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 build 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 preallocated context. If the parallel procedures can be organized into libraries, so that only one procedure of each library can be concurrently active at each processor, then it is sufficient to allocate one context per library.

Parallel procedures that are nonreentrant 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 at 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 need be implemented so as to avoid confusion. 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 preservation of message order by MPI. With such an approach, we may be gaining performance, but we loose modularity. It is not sufficient 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 need 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 to allocate first a large number of context's (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.

6.8 Motivating Examples

Discussion: The intra-communication examples were first presented at the June MPI meeting; the inter-communication routines (when added) are new.

6.8.1 Current Practice #1

Example #1a:

```
int me, size;
...
mpi_init();
mpi_comm_rank(MPI_COMM_ALL, &me);
mpi_comm_size(MPI_COMM_ALL, &size);

printf("Process %d size %d\n", me, size);
...
```

Example #1a is a do-nothing program that initializes itself legally, and refers to the the "all" communicator, and prints a message. This example does not imply that MPI supports printf-like communication itself.
Example #1b:

```
int me, size;
...
mpi_init();
mpi_comm_rank(MPI_COMM_ALL, &me);   /* local */
mpi_comm_size(MPI_COMM_ALL, &size); /* local */

if((me % 2) == 0)
   mpi_send(..., ((me + 1) % size), MPI_COMM_ALL);
else
   mpi_recv(..., ((me - 1 + size) % size), MPI_COMM_ALL);
...
```

Example #1b schematically illustrates message exchanges between "even" and "odd" processes in the "all" communicator.

6.8.2 Current Practice #2

```
void *data;
int me;
int count;
...
mpi_init();
mpi_comm_rank(MPI_COMM_ALL, &me);
```

```
if(me == 0)
{
    /* get input, create buffer ``data'' */
    ...
}

mpi_broadcast(data, count, MPI_BYTE, 0, MPI_COMM_ALL);

...
```

This example illustrates the use of a collective communication.

6.8.3 (Approximate) Current Practice #3

```
int me, count, count2;
void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
MPI_Group MPI_GROUP_ALL, grp0, grprem;
MPI_Comm commslave;
static int ranks[] = {0};
...
mpi_init();
mpi_comm_group(MPI_COMM_ALL, &MPI_GROUP_ALL);
mpi_comm_rank(MPI_COMM_ALL, &me);  /* local */
mpi_local_subgroup(group_all, 1, ranks, &grp0); /* local */
mpi_group_difference(group_all, grp0, &grprem); /* local */
mpi_comm_make(MPI_COMM_ALL, grprem, &commslave);

if(me != 0)
{
    /* compute on slave */
    ...
    mpi_reduce(send_buf, recv_buff, count, MPI_BYTE, commslave);
    ...
}
/* zero falls through immediately to this reduce, others do later... */
mpi_reduce(send_buf2, recv_buff2, count2, MPI_BYTE, MPI_COMM_ALL);
```

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_ALL context. This example illustrates how the two communicators (which possess distinct contexts) protect communication. That is, communication in MPI_COMM_ALL is insulated from communication in commslave, and vice versa.

In summary, for communication with "group safety," contexts within communicators must be distinct.

6.8.4 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
    int me;
    MPI_Group MPI_GROUP_ALL, subgroup;
    int ranks[] = {2, 4, 6, 8};
    ...
    mpi_init();
    mpi_comm_group(MPI_COMM_ALL, &MPI_GROUP_ALL);
    mpi_local_subgroup(MPI_GROUP_ALL, 4, ranks, &subgroup); /* local */
    mpi_group_rank(subgroup, &me);     /* local */

    mpi_comm_make(MPI_COMM_ALL, subgroup, &the_comm);

    if(me != MPI_UNDEFINED)
    {
        /* asynchronous receive: */
        mpi_irecv(handle, start, count, MPI_DOUBLE,
                      MPI_SRC_ANY, TAG_ARBITRARY, the_comm);
    }

    for(i = 0; i < SOME_COUNT, i++)
        mpi_reduce(..., the_comm);
```

6.8.5 Library Example #1

The main program:

```
    int done = 0;
    user_lib_t *libh_a, *libh_b;
    void *dataset1, *dataset2;
    ...
    mpi_init();
    ...
    init_user_lib(MPI_COMM_ALL, &libh_a);
    init_user_lib(MPI_COMM_ALL, &libh_b);
    ...
    user_start_op(libh_a, dataset1);
    user_start_op(libh_a, dataset2);
    ...
    while(!done)
    {
        /* work */
        ...
        mpi_reduce(..., MPI_COMM_ALL);
        ...
        /* see if done */
        ...
```

```
    }
    user_end_op(libh_a);
    user_end_op(libh_b);
```

The user library initialization code:

```
void init_user_lib(void *comm, user_lib_t **handle)
{
   user_lib_t *save;
   void *context;
   void *group;
   int len;

   user_lib_initsave(&save); /* local */
   mpi_contexts_alloc(comm, 1);
   mpi_comm_dup(comm, &(save -> comm));

   /* other inits */
   *handle = save;
}
```

Notice that the communicator comm passed to the library *is* needed to allocate new contexts.
User start-up code:

```
void user_start_op(user_lib_t *handle, void *data)
{
    mpi_irecv(&(handle -> irecv_handle), ..., handle -> comm);
    mpi_isend(&(handle -> isend_handle), ..., handle -> comm);
}
```

User clean-up code:

```
void user_end_op(user_lib_t *handle)
{
    mpi_wait(handle -> isend_handle);
    mpi_wait(handle -> irecv_handle);
}
```

6.8.6 Library Example #2

The main program:

```
    int ma, mb;
    ...
    list_a := ``[0,1]'';
    list_b := ``[0,2{,3}]'';

    mpi_local_subgroup(MPI_GROUP_ALL, 2, list_a, &group_a);
    mpi_local_subgroup(MPI_GROUP_ALL, 2(3), list_b, &group_b);
```

```
    mpi_comm_make(MPI_COMM_ALL, group_a, &comm_a);
    mpi_comm_make(MPI_COMM_ALL, group_b, &comm_b);

    mpi_comm_rank(comm_a, &ma);
    mpi_comm_rank(comm_b, &mb);

    if(ma != MPI_UNDEFINED)
       lib_call(comm_a);
    if(mb != MPI_UNDEFINED)
    {
       lib_call(comm_b);
       lib_call(comm_b);
    }
```

The library:

```
void lib_call(void *comm)
{
    int me, done = 0;
    mpi_comm_rank(comm, &me);
    if(me == 0)
        while(!done)
        {
            mpi_recv(..., comm, MPI_SRC_ANY);
            ...
        }
    else
    {
        /* work */
        mpi_send(..., comm, 0);
        ....
    }
    MPI_SYNC(comm);   /* include/no safety for safety/no safety */
}
```

The above example is really two examples, depending on whether or not you include rank 3 in `list_b`. This example illustrates that, despite contexts, subsequent calls to `lib_call` with the same context need not be safe from one another ("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.

Algorithms like "combine" have strong enough source selectivity so that they are inherently OK. 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 break. Therefore, such algorithms would have to utilize the tag to keep things straight. All of the foregoing is a discussion 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.

6.8.7 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. Note that the synchronous inter-communicator constructor (MPI_COMM_PEER_MAKE()) can be safely used here since there is no cyclic communication.

```
void * myComm;                    /* intra-communicator of local sub-group */
void * myFirstComm;                                  /* inter-communicator */
void * mySecondComm;           /* second inter-communicator (group B only) */
int membershipKey;
int subGroupLeaders[3];

MPI_INIT();
...

/* User code must generate membershipKey in the range [0, 1, 2] */
membershipKey = ... ;

/* Build intra-communicator for local sub-group and get group leaders */
/* of each sub-group (relative to MPI_COMM_ALL). */
MPI_COMM_SPLITL(MPI_COMM_ALL, membershipKey, 3,
                      subGroupLeaders, &myComm);

/* Build inter-communicators.  Tags are hard-coded. */
if (membershipKey == 0)
    {                             /* Group 0 communicates with group 1. */
    MPI_COMM_PEER_MAKE(myComm, 0, MPI_COMM_ALL,
        subGroupLeaders[1], 10, 1, &myFirstComm);
    }
else if (membershipKey == 1)
    {                      /* Group 1 communicates with groups 0 and 2. */
    MPI_COMM_PEER_MAKE(myComm, 0, MPI_COMM_ALL,
        subGroupLeaders[0], 10, 1, &myFirstComm);
    MPI_COMM_PEER_MAKE(myComm, 0, MPI_COMM_ALL,
        subGroupLeaders[2], 21, 1, &mySecondComm);
    }
else if (membershipKey == 2)
    {                             /* Group 2 communicates with group 1. */
```

```
        MPI_COMM_PEER_MAKE(myComm, 0, MPI_COMM_ALL,
            subGroupLeaders[1], 21, 1, &myFirstComm);
        }
```

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. Note that the "loosely synchronous" inter-communicator constructor (MPI_COMM_PEER_MAKE_START() and MPI_COMM_PEER_MAKE_FINISH()) is the best choice here due to the cyclic communication.

```
MPI_Comm  myComm;              /* intra-communicator of local sub-group */
MPI_Comm myFirstComm;                            /* inter-communicators */
MPI_Comm mySecondComm;
make_id firstMakeID, secondMakeID;              /* handles for "FINISH" */
int membershipKey;
int subGroupLeaders[3];

MPI_INIT();
...

/* User code must generate membershipKey in the range [0, 1, 2] */
membershipKey = ... ;

/* Build intra-communicator for local sub-group and get group leaders */
/* of each sub-group (relative to MPI_COMM_ALL). */
MPI_COMM_SPLITL(MPI_COMM_ALL, membershipKey, 3, subGroupLeaders, &myComm);

/* Build inter-communicators.  Tags are hard-coded. */
if (membershipKey == 0)
    {                      /* Group 0 communicates with groups 1 and 2. */
    MPI_COMM_PEER_MAKE_START(myComm, 0, MPI_COMM_ALL, subGroupLeaders[2], 20,
                             1, &firstMakeID);
    MPI_COMM_PEER_MAKE_START(myComm, 0, MPI_COMM_ALL, subGroupLeaders[1], 10,
                             1, &secondMakeID);
    }
else if (membershipKey == 1)
    {                      /* Group 1 communicates with groups 0 and 2. */
    MPI_COMM_PEER_MAKE_START(myComm, 0, MPI_COMM_ALL, subGroupLeaders[0], 10,
                             1, &firstMakeID);
```

```
        MPI_COMM_PEER_MAKE_START(myComm, 0, MPI_COMM_ALL, subGroupLeaders[2], 21,
                                 1, &secondMakeID);
        }
    else if (membershipKey == 2)
        {                       /* Group 2 communicates with groups 0 and 1. */
        MPI_COMM_PEER_MAKE_START(myComm, 0, MPI_COMM_ALL, subGroupLeaders[1], 21,
                                 1, &firstMakeID);
        MPI_COMM_PEER_MAKE_START(myComm, 0, MPI_COMM_ALL, subGroupLeaders[0], 20,
                                 1, &secondMakeID);
        }

    /* Everyone has the same "FINISH" code... */
    MPI_COMM_PEER_MAKE_FINISH(firstMakeID, &myFirstComm);
    MPI_COMM_PEER_MAKE_FINISH(secondMakeID, &mySecondComm);
```

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. Note that the synchronous inter-communicator constructor (MPI_COMM_NAME_MAKE()) can be safely used here since there is no cyclic communication.

```
    MPI_Comm myComm;                /* intra-communicator of local sub-group */
    MPI_Comm myFirstComm;                               /* inter-communicator */
    MPI_Comm mySecondComm;       /* second inter-communicator (group B only) */

    MPI_INIT();
    ...

    /* User builds intra-communicator myComm describing the local sub-group */
    /* using any appropriate MPI routine(s).  (For example, myComm could */
    /* have been passed in as an argument to a user subroutine.) */
    myComm = ... ;

    /* Build inter-communicators.  Group membership conditions must be */
    /* provided by the user. */
    if (<I am in group 0>)
        {                              /* Group 0 communicates with group 1. */
        MPI_COMM_NAME_MAKE(myComm, "Connect 10", 1, &myFirstComm);
        }
    else if (<I am in group 1>)
        {                       /* Group 1 communicates with groups 0 and 2. */
```

```
        MPI_COMM_NAME_MAKE(myComm, "Connect 10", 1, &myFirstComm);
        MPI_COMM_NAME_MAKE(myComm, "Connect 21", 1, &mySecondComm);
        }
    else if (<I am in group 2>)
        {                              /* Group 2 communicates with group 1. */
        MPI_COMM_NAME_MAKE(myComm, "Connect 21", 1, &myFirstComm);
        }
```

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. Note that the ”loosely synchronous” inter-communicator constructor (MPI_COMM_NAME_MAKE_START() and MPI_COMM_NAME_MAKE_FINISH()) is the best choice here due to the cyclic communication.

```
    MPI_Comm  myComm;             /* intra-communicator of local sub-group */
    MPI_Comm  myFirstComm;                          /* inter-communicators */
    MPI_Comm  mySecondComm;
    make_id firstMakeID, secondMakeID;             /* handles for "FINISH" */

    MPI_INIT();
    ...

    /* User builds intra-communicator myComm describing the local sub-group */
    /* using any appropriate MPI routine(s).  (For example, myComm could */
    /* have been passed in as an argument to a user subroutine.) */
    myComm = ... ;

    /* Build inter-communicators.  Group membership conditions must be */
    /* provided by the user. */
    if (<I am in group 0>)
        {                       /* Group 0 communicates with groups 1 and 2. */
        MPI_COMM_NAME_MAKE_START(myComm, "Connect 20", 1, &firstMakeID);
        MPI_COMM_NAME_MAKE_START(myComm, "Connect 10", 1, &secondMakeID);
        }
    else if (<I am in group 1>)
        {                       /* Group 1 communicates with groups 0 and 2. */
        MPI_COMM_NAME_MAKE_START(myComm, "Connect 10", 1, &firstMakeID);
        MPI_COMM_NAME_MAKE_START(myComm, "Connect 21", 1, &secondMakeID);
        }
```

```
else if (<I am in group 2>)
    {                      /* Group 2 communicates with groups 0 and 1. */
    MPI_COMM_NAME_MAKE_START(myComm, "Connect 21", 1, &firstMakeID);
    MPI_COMM_NAME_MAKE_START(myComm, "Connect 20", 1, &secondMakeID);
    }

/* Everyone has the same "FINISH" code... */
MPI_COMM_NAME_MAKE_FINISH(firstMakeID, &myFirstComm);
MPI_COMM_NAME_MAKE_FINISH(secondMakeID, &mySecondComm);
```

Section 7

Process Topologies

7.1 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 the majority of parallel applications this linear ordering does not reflect the communication structure 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. This process structure is henceforth called "virtual topology".

For the user who programs an application on such a process structure the most natural way of referring to individual processes is to specify their coordinates in the process topology. Usually a process does not have to know the identifiers of *all* processes in a grid, because communication only takes place with processes in a certain neighborhood, and perhaps with a few distant processes. Non–scalable features like the explicit listing of all participating processes in the application could, therefore, be avoided, if there were a means to specify the information which is really required.

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 the hardware 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 this standardization proposal, since it is too much dependent on the particular hardware. 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. But even for more complicated graph structures good heuristics often yield near–optimal results [13]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology", this information is most difficult to recover later. A random mapping is most likely the consequence. On some machines this will lead to unnecessary contention in the interconnect network. 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

[1]Version as of Sept 16, 1993

[7, 6].

7.2 Virtual Topologies

In the most general case the connection structure of a set of processes can 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 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. As a consequence, an automatic mapping tool will not try to assign the processes to neighboring processors.

The graph definition function MPI_GRAPH presented here does not include any weighting of the communication volume along the different edges. Processes are either neighbors or not. Experience with similar techniques in PARMACS [3, 5] 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, or 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 do the mapping as grids and tori rather than obscuring the information in a graph.

Therefore, in addition to the general graph, Cartesian topologies are supported. The topology definition function MPI_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 [14] 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.

Based on the positions in the virtual topology a process can look up the ranks of other processes by calling MPI_INQRANK. The processes are specified by their rectangular process coordinates. Thus, it is possible to inquire for the ranks of only those (few) processes with which messages have to be exchanged. The inverse function MPI_INQLOC translates a rank into the corresponding Cartesian coordinates.

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is fully consistent with other parts of MPI, and, whenever possible, it makes use of functions which are defined elsewhere. In particular, the virtual topology functions are based on the general management of process groups and communicators, and they complement the functions for collective communication. The current design of the process topology functions is based on the following assumptions:

- 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.

- One important feature of MPI are the process groups. The treatment of virtual topologies has to take this concept into account. The functions presented in this chapter are not only compatible with the process groups, they even take advantage of them for the definition of mapping overlays and dynamic re–mapping during execution time. Virtual process topologies help in the subdivision of process groups which in many applications has to be done along process coordinate lines. Furthermore, the definition of collective gather–type communication calls benefits from the natural ordering of processes in a virtual topology.

The virtual topology handling follows the general MPI rule of functional layering. At the bottom it is based on the general communicator creation function MPI_COMM_COLL_SUBSET plus a few topology specific mapping functions. In principle this is sufficient for defining topologies and using them in applications.

A number of higher–level functions provide an interface which is similar to what users of packages like Express and PARMACS are familiar with. This level can be written in a portable way in terms of the low–level functions. Implementations of the high–level functions which bypass the low level may be advantageous on some systems in terms of memory and cpu time usage, but this is not a requirement.

7.4 Overview of the proposed MPI functions

The features presented in this chapter are based on MPI functions for process group and communicator management which are defined in chapter 3. In particular, the function MPI_COMM_COLL_SUBSET is used internally for subdividing and re–ranking process groups. Since the topology group creation functions may require internal communication, communicators are passed to them instead of group identifiers. For consistency the topology group creation routines MPI_CART and MPI_GRAPH also return communicators instead of group identifiers for the newly created topological process groups. The topology routines MPI_MAP_CART, MPI_MAP_GRAPH, MPI_CART, MPI_GRAPH, and MPI_PARTC must be called as if they contain a barrier, but they may not actually have one in the implementation. This is analogous to the behaviour of the collective communication functions described in chapter 5.

Also in chapter 3, a mechanism for internally storing data with a communicator (cacheing) is defined. The topology functions can use this facility to store and retrieve process structure information.

There has been a general decision in MPI to start all numbering with 0 instead of 1. Thus, coordinates in a Cartesian structure are 0,1, ..., dim-1. In the same way, nodes in a general graph are numbered zero up. Row major numbering is used for the processes in a cartesian structure. This means that, for example, the four processes in a (2×2) grid are assigned the ranks

Process (0,0): rank 0
Process (0,1): rank 1
Process (1,0): rank 2
Process (1,1): rank 3

7.4.1 Low–level topology functions

On the lowest level groups (as part of a communicator) are created by the general function MPI_COMM_COLL_SUBSET. Using the input argument *key* the caller can change the process ranks in the new group as compared to the parent group. This feature can be used to efficiently map the processes in the group to the positions in the topology. For each class of topologies, therefore, there is a function which computes the rank of the calling process in the new group, based on the size and structure of the topology. Note that it is a valid implementation if the new ranks are identical to the old ranks. So, if for a given machine the placement of processes makes no difference, the implementation of the rank computation functions is trivial.

The functions in this section do not change the state of the calling process, but they provide the information used in a following group creation for process reordering.

For Cartesian structures the function is

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(*)	number of processes in each coordinate direction (integer)
IN	period(*)	periodicity specification in each coordinate direction, either set to MPI_PER or MPI_NOPER (integer)
OUT	newrank	reordered rank of the calling process (integer)

The corresponding function for general graph structures is

MPI_MAP_GRAPH(comm_old, nodes, index, nglist, newrank)

IN	comm_old	communicator for parent group (handle)
IN	nodes	number of graph nodes (integer)
IN	index(*)	define graph structure, see text (integer)
IN	nglist(*)	define graph structure, see text (integer)
OUT	newrank	reordered rank of the calling process (integer)

Definition of graph structure: for each graph node its set of neighbors is entered, i. e., the set of processes with which the node process is going to communicate most frequently. Since there is no weighting of node connections, it might be sensible to specify only those neighbors with which there is a considerable amount of communication.

For each node i $(0 \leq i \leq nodes - 1)$ the list of the neighbor nodes is stored at $nglist(j)$, $index(i) \leq j \leq index(i+1) - 1$. For the last process $index(i+1)$ points to the first free position in vector nglist. Note that the matrix is symmetric, i. e., if node i is a neighbor of node j, than node j is also a neighbor of node i. This property can be used for an internal checking of the input arguments.

The definitions of the arguments `nodes`, `index`, and `nglist` 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:

nodes =	4
index =	0, 2, 3, 4, 6
nglist =	1, 3, 0, 3, 0, 2

If the topology contains fewer processes than there are processes in the parent group, the newrank values for the remaining processes are set to MPI_PROCNULL. It is an error if the topology defines more processes than are contained in the parent group.

7.4.2 High–level topology functions

In principle, all that is needed to create an MPI process group with topological ordering are the two mapping functions and the general routine for subgroup communicator construction. However, this is not what the applications programmer expects to see on his level of abstraction. Therefore, another layer of functions is defined on top of the low–level functions. This higher level can be portably implemented in terms of the level below. Of course MPI implementors are free to bypass the lower level for the sake of better performance or resource usage.

For Cartesian and graph structures the functions MPI_CART and MPI_GRAPH create a process group with topological structure. They combine the call to the mapping function with the call to the group/communicator creation routine. The user can decide whether the ranking in the new group may be reordered (for a more efficient mapping) or not. In the latter case the rank of the calling process is copied from the parent group.

The option of not reordering the process ranks can be useful in the case of an overlay of different topologies. The user can then fully control which location in the first topology coincides with which location in the second one. The price which one obviously has to pay in this case is that for only one topology the process arrangement can be optimized.

As specified for the low–level functions it is an error if one attempts to define a topology with more locations than there are processes in the parent process group. If the number of locations is less than the number of available processes, the remaining processes of the parent group will not be members of the new group. For those processes the rank inquiry function MPI_COMM_RANK returns MPI_PROCNULL if called with the identifier of the new group.

MPI_CART is defined as

MPI_CART(comm_old, comm_cart, ndims, dims, period, reorder)

IN	comm_old	communicator for parent group (handle)
OUT	comm_cart	communicator for Cartesian group (handle)
IN	ndims, dims(*), period(*)	definition of Cartesian structure, (for details see the definition of MPI_MAP_CART)
IN	reorder	ranking may be reordered (MPI_REORDER) or not (MPI_NOREORDER) (integer)

Apart from some local computations, MPI_CART is based on the two function calls
MPI_MAP_CART (comm_old, ndims, dims, period, newrank)
and
MPI_COMM_COLL_SUBSET (comm_old, 0, newrank, comm_cart).

For graph structures the corresponding routine is

MPI_GRAPH(comm_old, comm_graph, nodes, index, nglist, reorder)

IN	comm_old	communicator for parent group (handle)
OUT	comm_graph	communicator for graph group (handle)
IN	nodes, index(*), nglist(*)	definition of graph structure, (for details see the definition of MPI_MAP_GRAPH)
IN	reorder	ranking may be reordered (MPI_REORDER) or not (MPI_NOREORDER) (integer)

The following topology information is likely to be stored with the process group by a high–level topology definition function:

1. Type of topology (Cartesian/graph)
2. For a Cartesian topology:
   - ndims (number of dimensions),
   - dims(*) (numbers of processes per coordinate direction),
   - period(*) (periodicity information),
   - own_position(*) (own position in grid, could also be computed from rank and dims(*))
3. For a graph topology:
   - index(*), nglist(*) (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.

The actual set of topology data stored with a process group is implementation dependent. It can contain more information than given above, whereas in other cases some items might be redundant.

7.4.3 Topology inquiry functions

If a topology has been defined with one of the above high–level 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_INQMAP returns the type of mapping which is assigned to a group.

MPI_INQMAP(comm, map)

IN	comm	communicator (handle)
OUT	map	topology type of group within communicator `comm` (integer)

The output value `map` is one of the following:

MPI_GRAPHTOP	graph topology
MPI_CARTTOP	Cartesian topology
MPI_NOTOP	no topology

For a process group with a graph structure which has been created by MPI_GRAPH, functions MPI_INQGRAPHDIMS and MPI_INQGRAPH look up topology information:

MPI_INQGRAPHDIMS(comm, indexlen, nglistlen)

IN	comm	communicator for group with graph structure (handle)
OUT	indexlen	number of entries in vector `index` (integer)
OUT	nglistlen	number of entries in vector `nglist` (integer)

The information provided by MPI_INQGRAPHDIMS can be used to dimension the vectors `index` and `nglist` long enough in the following call to MPI_INQGRAPH. The number of nodes in the graph is `indexlen - 1`.

MPI_INQGRAPH(comm, indexdim, nglistdim, index, nglist)

IN	comm	communicator for group with graph structure (handle)
IN	indexdim	length of vector index(*) in the calling program (integer)
IN	nglistdim	length of vector nglist(*) in the calling program (integer)
OUT	index(*), nglist(*)	vectors containing the graph structure (for details see the definition of MPI_MAP_GRAPH)

For a process group with Cartesian ordering which has been created by MPI_CART, functions MPI_INQCARTDIM and MPI_INQCART look up topology information:

MPI_INQCARTDIM(comm, dim)

IN	comm	communicator for group with Cartesian structure (handle)
OUT	dim	dimension of the Cartesian structure (integer)

MPI_INQCART(comm, maxdim, size, period, pos)

IN	comm	communicator for group with Cartesian structure (handle)
IN	maxdim	length of vectors size(*), period(*), and pos(*) in the calling program (integer)
OUT	size(*)	number of processes for each coordinate direction (integer)
OUT	period(*)	periodicity (MPI_PER/MPI_NOPER) for each coordinate direction (integer)
OUT	pos(*)	position of calling process in Cartesian structure (integer)

The inquiry for Cartesian topology information is thus split up into two calls. The reason is that otherwise the user might not know a safe value for the dimensioning of vectors `size, period` and `pos` before calling MPI_INQCART (input argument `maxdim`).

For a process group with Cartesian structure, the function MPI_INQRANK translates the logical process coordinates to process ranks as they are used by the point–to–point routines:

MPI_INQRANK(comm, location, rank)

IN	comm	communicator for group with Cartesian structure (handle)
IN	location(*)	coordinate vector of process (integer)
OUT	rank	process rank (integer)

In a sense, the inverse operation to MPI_INQRANK is provided by the function:

MPI_INQLOC(comm, rank, location, maxdim)

IN	comm	communicator for group with Cartesian structure (handle)
IN	rank	rank of a process within group `gid` (integer)
OUT	location(*)	coordinate vector of process (integer)
IN	maxdim	length of vector `location` in the calling program (only for error checking) (integer)

MPI_INQLOC translates a rank in the group into a topology location.

7.4.4 Partitioning of Cartesian structures

If a group has been created with MPI_CART, the function MPI_PARTC can be used to partition the group into subgroups which form hyper–planes of the original process structure. As an example, assume that MPI_CART has defined a two–dimensional grid structure on a process group. With a call to MPI_PARTC this group can then be split up into disjoint subgroups which represent rows or columns of the original group. The function syntax is:

MPI_PARTC(comm, lostdim, comm_new)

IN	comm	communicator for group with Cartesian structure (handle)
IN	lostdim	direction perpendicular to resulting hyper–planes (integer)
OUT	comm_new	communicator containing the hyper–plane group which includes the calling process (handle)

MPI_PARTC uses the function MPI_COMM_COLL_SUBSET to partition the parent group, using the process coordinate in direction `lostdim` as the key value. After a group has been partitioned by MPI_PARTC, the inquiry functions MPI_INQCART and MPI_INQRANK consistently refer to the grid sections if applied to a subgroup.

If the process topology is a Cartesian structure, a collective shift operation most likely is used along a coordinate direction. In MPI shift operations are handled by the more general send–receive functions. Those take as input the rank of a source process for the receive, and the rank of a destination process for the send.

If the function MPI_SHIFT_ID 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:

MPI_SHIFT_ID(comm, direction, steps, rank_source, rank_dest)

IN	comm	communicator for group with Cartesian structure (handle)
IN	direction	coordinate direction of shift (integer)
IN	steps	step length (>0: upwards shift, <0: downwards shift) (integer)
OUT	rank_source	rank of source process (integer)
OUT	rank_dest	rank of destination process (integer)

Depending of the periodicity of the Cartesian group in the specified coordinate direction, MPI_SHIFT_ID provides the identifiers for a circular or an end–off shift. In the case of an end–off shift, at the boundary process one of the identifiers is set to MPI_PROCNULL, indicating that there is no process in that direction to communicate with.

In a portable program the number of available processors usually is not known at compile time. In the case of a Cartesian topology the function MPI_MAKDIM 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_MAKDIM(nprocs, ndims, dims)

IN	nprocs	number of processes in the group (integer)
IN	ndims	number of Cartesian dimensions (integer)
INOUT	dims(*)	number of processes in each coordinate direction (integer)

The `nprocs` processes are divided as evenly as possible among `ndims` dimensions, subject to the constraint that all processes are used. 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 fix the number of processes in coordinate direction `i` to `dims(i)`. Processes will then be distributed among directions where `dims(i)` is set to 0, subject to the above constraints.

Note that an error will occur if `nprocs` 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.

When the function returns, `dims(i)` will be set to the number of processes in coordinate direction `i`. For `dims(i)` not set by the user, `dims(i)` will be ordered (MAXIMUM, ..., MINIMUM). Array `dims(*)` is suitable for use as input to routine MPI_CART.

7.5 A simple example

The example in figure 7.1 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), period(ndims), i, j, rank
      real*8 u(0:101,0:101), f(0:101,0:101)
      data dims / ndims * 0 /
      comm = MPI_COMM_ALL
C     Set process grid size and periodicity
      call MPI_MAKDIM(comm, ndims, dims)
      period(1) = MPI_PER
      period(2) = MPI_PER
C     Create a grid structure in ALL group and inquire about own position
      call MPI_CART (comm, comm_cart, ndims, dims, period, reorder, rank)
      call MPI_INQCART (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_INQRANK (comm_cart, neigh_def, neigh_rank(1))
      neigh_def(1) = i+1
      neigh_def(2) = j
      call MPI_INQRANK (comm_cart, neigh_def, neigh_rank(2))
      neigh_def(1) = i
      neigh_def(2) = j-1
      call MPI_INQRANK (comm_cart, neigh_def, neigh_rank(3))
      neigh_def(1) = i
      neigh_def(2) = j+1
      call MPI_INQRANK (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 7.1: Set–up of process structure for two–dimensional parallel Poisson solver.

Section 8

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation (such as error handling and the number of available communicators). In addition, a discussion of MPI startup through the use of an initialization routine is included, as well as the relationship of signals to an MPI implementation.

8.1 Implementation information

MPI contain routines for managing implementation limits in MPI. These routines will take a requested value and return the actual value, possibly including `MPI_NA` for "not applicable."

[1] Version as of September 21, 1993

MPI_ValidTags (low, high)

out	low	Smallest valid tag value
out	high	Largest valid tag value

MPI_NumCommunicator(n)

out	n	Number of Communicators that can be created

MPI_Valid_Communicator(comm, flag)

in	comm	Communicator
out	flag	True if comm is a valid communicator and false otherwise.

MPI_Collective_Ops_Sync(flag)

out	flag	True if collective operations perform an implicit barrier

Discussion: Someone needs to say what this does

MPI_GetMachineName(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.

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 return value of `MPI_NA` is allowed but strongly discouraged.

8.1.1 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

```
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 `size` 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`.

Proposal I

MPI_GetBufferParams(b, n, bt, nt)

out	b	Maximum size of buffered messages between any pair of tasks.
out	n	Maximum number of buffered messages between any pair of tasks.
out	bt	Maximum size of buffered messages between calling task and all other tasks.
out	nt	Maximum number of buffered messages between calling task and all other tasks.

The value of any of these may be `MPI_NA`. If this is the case, the MPI implementation makes no guarantee about buffering.

MPI_SuggestBufferParams(b, n, bt, nt)

in	b	Maximum size of buffered messages between any pair of tasks.
in	n	Maximum number of buffered messages between any pair of tasks.
in	bt	Maximum size of buffered messages between calling task and all other tasks.
in	nt	Maximum number of buffered messages between calling task and all other tasks.

This routine allows the user to tell the MPI implementation that a certain amount of buffering would be desirable. The MPI implementation is not required to provide such buffering.

If the value of any of the parameters is negative, the parameter is ignored. the value with the value that the implementation supports. The value may be

It is expected that a common use of these routines will be

```
MPI_GetBufferParams( &b, &n, &bt, &nt );
if (b < buffer_needed)
    switch to no-buffering algorithm
else
    exploit buffering
```

Another use is to inform the MPI implementation of the buffer space desires of the application:

```
bt = b = buffer_needed;
nt = n = 1;
MPI_SuggestBufferParams( b, n, bt, nt );
MPI_GetBufferParams( b, n, bt, nt );
if (b < buffer_needed)
    switch to no-buffering algorithm
else
    exploit buffering
```

The value of `MPI_NA` is guaranteed to be negative.

Discussion: These need to refer to a diagram of possible buffer usages. We must be careful to handle the case of (a) no buffering and (b) buffers on the sender.

An implementation may provide for more detailed buffer management with

MPI_ExclusiveBuffer(size, partner, comm)

in	size	Size (in bytes) of buffer to dedicate
in	partner	Rank of task that buffer can accept messages from
in	comm	Communicator

This routine can return failure to indicate that the buffer could not be provided or that the feature is not supported.

Discussion: Another approach is to use a single routine that takes a query parameter and returns the requested data (or a "not available").

The routines returning the maximum number of things (like groups) could instead return the maximum number of *new* things that could be created. All of these routines are meant to be advisory, particularly in a multithreaded environment, since they may be allocated dynamically.

Proposal II

An application program can optionally provide space for MPI's use in buffering message data via the following call:

MPI_User_Provides_Buffer(comm, len, buffer)

in	comm	is a communicator with which the buffer is to be associated
in	len	the length of 'buffer', in bytes
out	buffer	a scratch array of len bytes for MPI's use in buffering messages

Discussion: Should `buffer` be an "in" parameter?

By default, application programs are "unsafe" if they assume buffering. However, an application can make it safe to assume a specified amount of data buffering, by providing buffer space via the `MPI_User_Provides_Buffer` call.

If `MPI_User_Provides_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.

That is, user-provided buffer space may be consumed by outgoing messages, but not by incoming ones, and MPI is not required to split any message if it must be copied to the buffer.

Following the call to `MPI_User_Provides_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.

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.

Discussion: This proposal does not provide guarantees on the *number* of outstanding sends, only their aggregate message length. Not guaranteeing number is consistent with MPI's general stance of treating limits as a quality-of-implementation issue. Guaranteeing the length is nominally at odds with MPI's general stance, but is felt to be an important compromise in order to support programs that assume current "common practice." It is unreasonable to simply declare existing programs to be "unsafe," when a simple method exists to make them safe.

8.1.2 I/O

MPI provides no way to determine if I/O operations are collective or not; that is, whether or not a `printf` is an implicit barrier. This was an explicit choice of the MPI committee. User programs must use vendor-specific mechanisms to manage I/O.

Discussion: Here is a slightly different version with some explanation Consider the code

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

This simple program fragment can cause a program to deadlock if the interpretation of `printf` is as a collective operation. The replacement

```
printf( "Starting program\n" );
```

may print once or p times if there are p tasks running. To allow MPI programmers to write portable programs that use conventional I/O operations (which are *not* required to be supported), the routine `MPI_GetIOmode` is provided. MPI_GetIOmode (iomode, comm)

out	iomode	The I/O mode
in	comm	Communicator

Returns whether I/O is per task (`MPI_IOPERTASK`, per Communicator(`MPI_IOPERCOMM`), or undefined (`MPI_NA`).

Discussion: a counterpart that sets the I/O mode could be provided

8.2 Error handling

There are three possible actions that MPI can take when an error is encountered.

`MPI_ERRORSFATAL` All errors are *fatal* and cause the program to abort.

`MPI_ERRORSRETURN` All routines return an error code to the user as the result of the erroneous call.

`MPI_ERRORSCALLUSERROUTINE` User can specify an error handler for a communicator; this error handler is inherited by derived communicators. An MPI standard error handler which prints a message and calls `MPI_Abort` can be used. If this mode is used, `MPI_SetErrhandler` must be called to specify the error handler.

MPI_GetErrormode (oldmode, comm)

out	oldmode	The current error handling mode
in	comm	Communicator

Gets the error handling mode.

MPI_SetErrormode (newmode, comm)

in	newmode	The desired new error mode
in	comm	Communicator

Sets the error handling mode.

To set the error handler, use

```
typedef int ()(MPICOMM, int *, ...) MPI_ERRHANDLER;
```

The return value is the value to return as the MPI routine's "error return."

Discussion: What arguments should be provided to the error handler?

MPI_ERRHANDLER MPI_Set_ErrHandler(MPI_COMM comm, MPI_ERRHANDLER handler)

in	comm	Communicator
in	handler	User-defined error handler to call

This routine returns the previous error handler as its value.

Discussion: Despite the fact that this routine has a communicator as an argument, it is a purely local routine (that is, the communicator is *not* used for communication!

There are two predefined error handlers that the user may use instead of writing their own:

MPI_NOHANDLER returns a status to the user.

MPI_DEFAULTHANDLER aborts from within MPI.

char * MPI_ErrorString(errorcode)

in	errorcode	Error code returned by an MPI routine

This returns a pointer to a string that contains text corresponding to the error message.

Discussion: The Fortran binding should probably copy into a string passed in.

8.3 Examining MPI objects

It is useful to be able to unpack the various opaque objects into individual variables, such as buffer descriptors and contexts, and the message queue.

Discussion: These still need specific output formats (how is the output structure laid out? What is the Fortran binding? What information is provided?).

MPI_DumpReceivedQueue(len, lenout, info)

in	len	Size of `info` array
out	lenout	Number of entries set in info
out	info	Array of structures, with each structure containing the tag, source, comm, and length (bytes? elements?) of each message in the received queue.

Discussion: This routine is provided to allow codes to examine the queue. As will all such routines, if another thread can access the receive queue, a message whose information is returned by this routine may be received by another thread. There is no requirement that these messages can then be received by the task that called `MPI_DumpReceivedQueue`.

Discussion: An alternate approach for the queue and context inquiry functions (at least for C) is for the routine to allocate the space rather than having it passed in.

MPI_DumpDatatype(datatype, lenin, lenout info)

in	datatype	Datatype to expand
in	len	Size of `info` array
out	lenout	Number of entries set in info
out	info	Array of structures, with each structure containing the datatype, extent, and address of each element of a datatype.

MPI_DumpPostedSends(len, lenout, info)

in	len	Size of `info` array
out	lenout	Number of entries set in info
out	info	Array of structures, with each structure containing the tag, destination, comm, and length (bytes? elements?) of each message for which there is a posted send.

MPI_DumpPostedReceives(len, lenout, info)

in	len	Size of `info` array
out	lenout	Number of entries set in info
out	info	Array of structures, with each structure containing the tag, source, comm, and length (elements) of each message for which there is a posted receive.

8.4 Interaction with signals

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 does 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.

Finally, an MPI implementation should not appropriate signals from the user. For example, an MPI implementation that did not allow the user to use `SIGALRM` in a POSIX environment would not conform to this standard. An exception is made for `SIGUSR1` and `SIGUSR2`.

Discussion: Should there be a way to find out what signals the MPI implementation uses?

8.5 Information on CPU load

Discussion: Submitted for consideration Currently, most parallelized algorithms for distributed

memory machines assume that they will run on a homogeneous set of processors. (I.e. that each node in the parallel system has approximately the same computing power and I/O speed.) The main focus in parallel computing has thus far been to minimize communication over-head and achieve a load-balance for the particular algorithm or application.

However, as each computational node a parallel system become more complex, their individual processor/node performance characteristics will tend to deviate more as some nodes perform I/O while others concentrate on system tasks, etc.

This behavior has so far been addressed through multi-tasking on large shared-memory systems such as a network of IBM 3090s, however, no simple, straight-forward solutions exist currently for the distributed memory cases.

It is, however, this authors belief that load-balancing application at the "load-info" level (i.e. based on each node's dynamic load at run-time) will become very important in the future – especially if there were mechanisms built into MPI that would facilitate such implementations.

Currently, these is no portable way of determining the load; even for those systems that support some load information, determining the information is often expensive or difficult.

MPI_CPU_Load_Info(n)

out	n	the number of jobs the scheduler has outstanding.

This is a local operation that applies only to the processor running the calling task. A value of `MPI_NA` is permissible. The definition of load is left up to the implementation.

This function is intended to provide very basic information about the load; methods for getting more detailed information (available memory, CPU speed, etc.) are not specified by MPI.

8.6 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`.

Discussion: The inability to specify or discover I/O behavior causes MPI to fail to meet the source-code-portability goal.

All MPI programs must contain a call to `MPI_init`; this routine must be called before any other MPI routines are called. The format is:

```
MPI_init( );
```

Discussion: I removed `args` from `MPI_init` since the committee felt that penalizing Fortran's inability to access the environment was too punitive. Note that POSIX environments are required to make the command line accessible to Fortran.

MPI_init()

This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous (see `MPI_Initialized`).

MPI_finalize()

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.

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(errorcode)

in	errorcode	Error code to return to invoking environment.

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)`.

Discussion: Passing the command-line arguments to `MPI_init` should be considered in the C binding.

Note that POSIX includes a *requirement* that access to the command line be provided to Fortran programs (see 8.9 in Std 1003.9–1992).

Section 9

Profiling

9.1 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.

9.2 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).

9.3 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.

9.4 Examples

9.4.1 Profiler implementation

Suppose that the profiler wishes to accumulate the total amount of data sent by the MPI_SEND function, along with the total elapsed time spent in the function. This could trivially be achieved thus :–

WARNING : syntax correction needed in examples when MPI syntax is defined.

```
static int totalBytes;
static int totalTime;
extern int now(void);   /* Some suitable timer function ! */

int MPI_SEND(const int tag, const int bytes, void * buffer, const int who)
{
   int tstart = now();
   int result = PMPI_SEND(tag,bytes,buffer,who);   /* Pass on all the arguments */

   totalBytes += bytes;                /* Accumulate byte count */
   totalTime  += now() - tstart;       /* and time              */

   return result;
}
```

9.4.2 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.

9.5 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.

Section 10

Initial Implementation Subset

10.1 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 only a potential first step in the process. All implementations must ultimately 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 limit access to full MPI functionality through complete elimination of functions or limitations on the creation mechanisms for certain MPI input arguments. Any argument capable of being constructed within the subset can be used as an argument to a subset function where it would be allowed in the full standard. For example, any communicator that can be created with the subset is an allowed argument in any subset function where a communicator is an appropriate argument. However, some of the more complex mechanisms for communicator creation have been removed from the subset.

Users may utilize MPI features outside this subset that are offered by various implementors. However, people who require portability during the early development of MPI may experience some difficulties until later in the development process. This subset is meant to offer the opportunity for a greater level of portability until the full MPI standard is

[1] Version of September 16, 1993

implemented on multiple platforms.

10.2 Criteria and Rational

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. contain routines that are as close as possible to current standard practice to minimize the effort to port codes.
2. contain as many new and important MPI features as possible.
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 optimized 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 was 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.

10.3 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 4.2.2, the semantics in section 4.4, the type matching in section 4.5 and the communication objects in subsection 4.8.1 are all part of the subset.

10.3.1 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 inter-communicator functionality is not included in the subset. Also, the caching facility is not included. Finally, MPI_CONTEXTS_ALLOC and MPI_COMM_MAKE are excluded from the subset. This effectively means that the user cannot create a new context. The user can still manipulate the contexts provided to create new communicators. The full functionality on groups is provided.

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
6.2.1	Predefined Comm.	MPI_COMM_ALL MPI_COMM_HOST MPI_COMM_PARENT MPI_COMM_SELF	——
??	Local Operations	MPI_GROUP_SIZE MPI_GROUP_RANK MPI_TRANSLATE_RANKS	——
??	Local Group Constructor	MPI_LOCAL_SUBGROUP MPI_LOCAL_EXCL_SUBGROUP MPI_LOCAL_SUBGROUP_RANGES MPI_LOCAL_SUBGROUP_EXCL_RANGES MPI_LOCAL_GROUP_UNION MPI_LOCAL_GROUP_INTERSECT MPI_LOCAL_GROUP_DIFFERENCE MPI_GROUP_FREE MPI_GROUP_DUP	——
??	Coll. Group Const.	MPI_COLL_SUBGROUP	——
??	Local Operations	MPI_CONTEXTS_FREE	——
??	Collective Ops.	——	MPI_CONTEXTS_ALLOC
??	Local Comm. Ops.	MPI_COMM_SIZE MPI_COMM_RANK	——
??	Local Const.	MPI_COMM_GROUP MPI_COMM_CONTEXT MPI_COMM_BIND MPI_COMM_UNBIND MPI_COMM_DUP	——
??	Coll. Comm. Const.	——	MPI_COMM_MAKE
??	Synchronous Inter-Comm. Const.	——	MPI_COMM_PEER_MAKE MPI_COMM_NAME_MAKE MPI_COMM_PEER_MAKE_START MPI_COMM_PEER_MAKE_FINISH MPI_COMM_NAME_MAKE_START MPI_COMM_NAME_MAKE_FINISH

6.6.1	Functionality	——	MPI_COMM_STAT MPI_COMM_SPLITL MPI_GET_ATTRIBUTE MPI_SET_ATTRIBUTE MPI_TEST_ATTRIBUTE MPI_DELETE_ATTRIBUTE

10.3.2 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 include in the subset. It was felt that the added complexity of implementation was outweighed by the importance of derived datatypes.

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
4.2	Basic Send Operation	MPI_SEND	——
4.2.1	Message Data	All types	——
4.3	Basic Recv Operation	MPI_RECV	——
4.3.1	Return Status	MPI_GET_SOURCE MPI_GET_TAG MPI_GET_COUNT	——
4.7	Comm. Modes	Standard	MPI_RSEND MPI_SSEND
4.8.2	Comm. Initiation	MPI_ISEND MPI_IRECV	MPI_IRSEND MPI_ISSEND
4.8.3	Comm. Completion	MPI_WAIT MPI_TEST	——
4.8.3	Multiple Completion	——	MPI_WAITANY MPI_TESTANY MPI_WAITALL MPI_TESTALL
4.9	Probe & Cancel	MPI_PROBE MPI_IPROBE MPI_PROBE_COUNT	MPI_CANCEL MPI_TEST_CANCEL
4.10	Persistent Comm. Objects	——	MPI_CREATE_SEND MPI_CREATE_RSEND MPI_CREATE_SSEND MPI_CREATE_RECV

			MPI_START MPI_FREE
4.11	Send-receive	MPI_SENDRECV	MPI_EXCHANGE
4.12	Null process	MPI_PROCNULL	——
4.13.1	Derived Datatypes	MPI_TYPE_CONTIGUOUS MPI_TYPE_VECTOR MPI_TYPE_HVECTOR MPI_TYPE_INDEXED MPI_TYPE_HINDEXED MPI_TYPE_STRUCT	——
4.13.2	Additional Functions	MPI_TYPE_COMMIT MPI_FREE MPI_ADDRESS MPI_TYPE_EXTENT	——

10.3.3 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. Also, the scan functions are excluded from the subset.

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
5.3	Barrier Synch.	MPI_BARRIER	——
5.4.1	Broadcast	MPI_BCAST	——
5.4.2	Gather	MPI_GATHER	——
5.4.3	Scatter	MPI_SCATTER	——
5.4.4	All-to-all broadcast	MPI_ALLCAST	——
5.4.5	All-to-all scatter-gather	MPI_ALLTOALL	——
5.5.1	Reduce	MPI_REDUCE	MPI_USER_REDUCE MPI_USER_REDUCEA
		MPI_ALLREDUCE	MPI_USER_ALLREDUCE MPI_USER_ALLREDUCEA
5.5.2	Scan	——	MPI_SCAN MPI_USER_SCAN MPI_USER_SCANA

10.3.4 Topology Functionality

The capability presented in the process topology chapter represents a new functionality in MPI and is not generally common practice today. Though this new functionality will undoubtedly be of use to many users of MPI, process topologies are being excluded from the subset in the interest of minimizing the difficulty of implementation. Thus, all of the functionality in chapter 7 is excluded from the subset.

10.3.5 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 2.5.

10.3.6 Environmental Management and Inquiry

Discussion:
This section is vague because it is still not clear to me what will be in this chapter. Also, we did not have any detailed discussion at the last meeting about what to include/exclude. There was only a discussion that the subset should contain some of the routines. This section mostly serves as a list of routines we need to decide on.

A general discussion will go here once we decide what is going on.

The table below contains a listing by section of which routines from the Environmental Management and Inquiry chapter are included and excluded from the subset.

Section	Title	Included	Excluded
??	MPI Program Startup Prop.	MPI_init	——
??	MPI-Specific Enquiry Functions	——	MPI_ValidTags
			MPI_NumGroups
			MPI_NumCntxs
			MPI_BufferSize
			MPI_BufferManagement
			MPI_BufferSizePairs
			MPI_BufferSizeAll
			MPI_ExclusiveBuffer
			MPI_IOmode
			MPI_ErrorMode
			MPI_Has_Ready_Receiver
			MPI_Has_Heterogeneous
			MPI_UnpackBufferDescriptor
			MPI_DumpMessageQueue
			MPI_StatusOfHandles
??	Parallel Programming	——	MPI_GetNbrs
			MPI_GetPhysNode
			MPI_GetTopology
			MPI_GetDistance
			MPI_GetDiameter

10.3.7 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.

10.3.8 Subset Testing

Discussion:
There has been discussion of creating a test suite for MPI. It was agreed that a subset of this testing suite would be available for testing the subset specification.

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] 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.

[4] R. Butler and E. Lusk. User's guide to the P4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992.

[5] 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.

[6] 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.

[7] 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.

[8] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991.

[9] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992.

[10] G. Geist and V. Sunderam. Network based concurrent computing on the PVM system. Technical Report TM-11760, Oak Ridge National Laboratory, June 1991.

[11] 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.

[12] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990.

[13] H. Mühlenbein O. Krämer. Mapping strategies in message–based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989.

[14] Parasoft Corporation. *Express Version 1.0: A Communication Environment for Parallel Computers*, 1988.

[15] 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.

[16] 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.

[17] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992.

[18] V. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[19] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992.

Annex A

Language Bindings

A.1 Introduction

In this section we summarize the specific bindings for both Fortran and C. We present first the C bindings, then the Fortran bindings. Listings are alphabetical within chapter.

A.2 Defined Constants for C and Fortran

These are required defined constants, to be defined in the files `mpi.h` and `mpif.h`.

```
/* return codes */
MPI_SUCCESS

/* communication modes */
MPI_STANDARD
MPI_READY
MPI_SYNCHRONOUS
MPI_RECV

/* assorted constants */
MPI_BOTTOM
MPI_PROCNULL
MPI_ANY_PROC
MPI_ANY_TAG
MPI_NA

/* Error-handling specifiers */
MPI_ERRORSFATAL
MPI_ERRORSRETURN
MPI_ERRORSCALLUSERROUTINE

/* Pre-defined error handlers */
MPI_NOHANDLER
MPI_DEFAULTHANDLER
```

[1]Version as of September 21, 1993

```
/* persistence values */
MPI_PERSISTENT
MPI_EPHEMERAL

/* elementary datatypes */
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_BYTE
MPI_UCHAR
MPI_USHORT
MPI_ULONG
MPI_UINT

/* reserved communicators */
MPI_COMM_ALL
MPI_COMM_HOST
MPI_COMM_PARENT
MPI_COMM_SELF

/* global ops */
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_AND
MPI_OR
MPI_XOR
MPI_MAXLOC
MPI_MINLOC
```

The following are defined C type definitions, also included in the file `mpi.h`.

```
/* opaque types */
MPI_Persistence
MPI_Mode
MPI_Tag
MPI_Op
MPI_Comm_name

/* assorted structures */
MPI_Status
MPI_Group
```

```
MPI_Communicator
MPI_Datatype
MPI_Handle
MPI_Attribute

/* Function types */
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* start, int count, MPI_Datatype datatype, int dest,
             MPI_Tag tag, MPI_Communicator comm)

int MPI_recv(void* start, int count, MPI_Datatype datatype, int source,
             MPI_Tag tag, MPI_Communicator comm, MPI_Status *status)

int MPI_get_source(MPI_Status *status, int source)

int MPI_get_tag(MPI_Status *status, MPI_Tag tag)

int MPI_get_count(MPI_Status *status, int count)

int MPI_rsend(void* start, int count, MPI_Datatype datatype, int dest,
             MPI_Tag tag, MPI_Communicator comm)

int MPI_ssend(void* start, int count, MPI_Datatype datatype, int dest,
             MPI_Tag tag, MPI_Communicator comm)

int MPI_isend(MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int dest, MPI_Tag tag,
             MPI_Communicator comm)

int MPI_irsend(MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int dest, MPI_Tag tag,
             MPI_Communicator comm)

int MPI_issend(MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int dest, MPI_Tag tag,
             MPI_Communicator comm)

int MPI_irecv(MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int source, MPI_Tag tag,
             MPI_Communicator comm)

int MPI_wait(MPI_Handle handle, MPI_Status *status)

int MPI_test(MPI_Handle handle, int flag, MPI_Status *status)

int MPI_waitany(int count, MPI_Handle *array_of_handles, int index,
             MPI_Status *status)

int MPI_testany(int count, MPI_Handle *array_of_handles, int index,
             MPI_Status *status)
```

```
int MPI_waitall(int count, MPI_Handle *array_of_handles, array_of_status)

int MPI_testall(int count, MPI_Handle *array_of_handles, int flag,
             array_of_status)

int MPI_iprobe(int source, MPI_Tag tag, MPI_Communicator comm, int flag,
             MPI_Status *status)

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

int MPI_probe(int source, MPI_Tag tag, MPI_Communicator comm,
             MPI_Status *status)

int MPI_cancel(MPI_Handle handle)

int MPI_test_cancelled(MPI_Status *status, int flag)

int MPI_create_send(MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int dest, MPI_Tag tag,
             MPI_Communicator comm)

int MPI_create_rsend(MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int dest, MPI_Tag tag,
             MPI_Communicator comm)

int MPI_create_ssend(MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int dest, MPI_Tag tag,
             MPI_Communicator comm)

int MPI_create_recv(MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int source, MPI_Tag tag,
             MPI_Communicator comm)

int MPI_start(MPI_Handle handle)

int MPI_commobj_free(MPI_Handle handle)

int MPI_sendrecv(void* send_start, int send_count, MPI_Datatype send_type,
             int dest, MPI_Tag send_tag, void* recv_start, int recv_count,
             MPI_Datatype recv_type, int source, MPI_Tag recv_tag,
             MPI_Communicator comm, MPI_Status *status)

int MPI_exchange(void* start, int count, MPI_Datatype datatype, int dest,
             MPI_Tag send_tag, int source, MPI_Tag recv_tag,
             MPI_Communicator 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, array_of_types, int *array_of_displacements,
            int *array_of_blocklengths, MPI_Datatype newtype)

int MPI_address(void* location, int address)

int MPI_type_extent(MPI_Datatype datatype, int extent)

int MPI_type_commit(MPI_Datatype datatype)

int MPI_type_free(MPI_Datatype datatype)

int MPI_comm_init(MPI_Handle handle, MPI_Datatype datatype, int source_dest,
            MPI_Tag tag, MPI_Communicator comm, MPI_Mode op_mode,
            MPI_Persistence persistence)
```

A.4 C Bindings for Collective Communication

```
int MPI_barrier(MPI_Communicator comm)

int MPI_bcast(void* buffer, int count, MPI_Datatype datatype, int root,
            MPI_Communicator comm)

int MPI_gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
            void* recvbuf, int maxrecvcnt, int *recvcounts,
            MPI_Datatype recvtype, int root, MPI_Communicator comm)

int MPI_scatter(void* sendbuf, int *sendcounts, MPI_Datatype sendtype,
            void* recvbuf, int maxrecvcnt, recvcnt, MPI_Datatype recvtype,
            int root, MPI_Communicator comm)

int MPI_allcast(void* sendbuf, int sendcount, MPI_Datatype sendtype,
            void* recvbuf, int maxrecvcnt, int *recvcounts,
            MPI_Datatype recvtype, MPI_Communicator comm)

int MPI_alltoall(void* sendbuf, int *sendcounts, MPI_Datatype sendtype,
            void* recvbuf, int maxrecvcnt, int *recvcounts,
            MPI_Datatype recvtype, MPI_Communicator comm)

int MPI_reduce(void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, int root,
            MPI_Communicator comm)

int MPI_user_reduce(void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, void(*)(void*,void*,int*) function,
            int root, MPI_Communicator comm)
```

```
int MPI_user_reducea(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void(*)(void*,void*,int*) function,
             int root, MPI_Communicator comm)

int MPI_allreduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Communicator comm)

int MPI_user_allreduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void(*)(void*,void*,int*) function,
             MPI_Communicator comm)

int MPI_user_allreducea(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void(*)(void*,void*,int*) function,
             MPI_Communicator comm)

int MPI_reduce_scatter(void* sendbuf, void* recvbuf, int *distcounts,
             MPI_Datatype datatype, MPI_Op op, MPI_Communicator comm)

int MPI_user_reduce_scatter(void* sendbuf, void* recvbuf, int *distcounts,
             MPI_Datatype datatype, void(*)(void*,void*,int*) function,
             MPI_Communicator comm)

int MPI_user_reduce_scattera(void* sendbuf, void* recvbuf, int *distcounts,
             MPI_Datatype datatype, void(*)(void*,void*,int*) function,
             MPI_Communicator comm)

int MPI_scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Communicator comm)

int MPI_user_scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void(*)(void*,void*,int*) function,
             MPI_Communicator comm)

int MPI_user_scana(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, void(*)(void*,void*,int*) function,
             MPI_Communicator 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_translate_ranks(MPI_Group group_a, int n, int *ranks_a,
             MPI_Group group_b, int *ranks_b)

int MPI_local_group_union(MPI_Group group1, MPI_Group group2,
             MPI_Group group_out)

int MPI_local_group_intersect(MPI_Group group1, MPI_Group group2,
             MPI_Group group_out)

int MPI_local_group_difference(MPI_Group group1, MPI_Group group2,
```

```
            MPI_Group group_out)

int MPI_local_subgroup(MPI_Group group, int n, int *ranks,
            MPI_Group new_group)

int MPI_local_excl_subgroup(MPI_Group group, int n, int *ranks,
            MPI_Group new_group)

int MPI_local_subgroup_ranges(MPI_Group group, int n, int *ranges,
            MPI_Group new_group)

int MPI_local_subgroup_excl_ranges(MPI_Group group, int n, int *ranges,
            MPI_Group new_group)

int MPI_group_dup(MPI_Group group, MPI_Group new_group)

int MPI_group_free(MPI_Group group)

int MPI_contexts_alloc(MPI_Communicator comm, int n)

int MPI_contexts_free(MPI_Communicator comm, int n)

int MPI_comm_size(MPI_Communicator comm, int *size)

int MPI_comm_rank(MPI_Communicator comm, int *rank)

int MPI_comm_group(MPI_Communicator comm, MPI_Group group)

int MPI_comm_contexts(MPI_Communicator comm, int *n)

int MPI_comm_dup(MPI_Communicator comm, MPI_Communicator new_comm)

int MPI_comm_local_dup(MPI_Communicator comm, MPI_Communicator new_comm)

int MPI_comm_coll_dup(MPI_Communicator comm, MPI_Communicator new_comm)

int MPI_comm_make(MPI_Communicator sync_comm, MPI_Communicator group_comm,
            MPI_Communicator new_comm)

int MPI_comm_local_make(MPI_Communicator sync_comm,
            MPI_Communicator group_comm, MPI_Communicator new_comm)

int MPI_comm_coll_make(MPI_Communicator sync_comm,
            MPI_Communicator group_comm, MPI_Communicator new_comm)

int MPI_comm_coll_subset(MPI_Communicator comm, int color, int key,
            MPI_Communicator new_comm)

int MPI_comm_free(MPI_Communicator comm)

int MPI_comm_stat(MPI_Communicator comm, MPI_Status *status)

int MPI_comm_peer_make(MPI_Communicator my_comm, int my_leader,
            MPI_Communicator peer_comm, int peer_rank, MPI_Tag tag,
            int num_comms, MPI_Communicator *new_comms)

int MPI_comm_peer_make_start(my_comm, int my_leader,
            MPI_Communicator peer_comm, int peer_rank, MPI_Tag tag,
```

```
              int num_comms, MPI_Comm_name make_id)

int MPI_comm_peer_make_finish(MPI_Comm_name make_id,
              MPI_Communicator *new_comms)

int MPI_comm_splitl(MPI_Communicator comm, int key, int nkeys, leaders,
              MPI_Communicator sub_comm)

int MPI_comm_name_make(MPI_Communicator my_comm, char *name, int num_comms,
              MPI_Communicator *new_comms)

int MPI_comm_name_make_start(MPI_Communicator my_comm, char *name,
              int num_comms, MPI_Comm_name make_id)

int MPI_comm_name_make_finish(MPI_Comm_name make_id,
              MPI_Communicator *new_comms)

int MPI_attribute_alloc(int n, MPI_Attribute *handle_array, int len)

int MPI_attribute_free(MPI_Attribute *handle_array, int n)

int MPI_get_attribute_key(int keyval)

int MPI_set_attribute(MPI_Attribute handle, int keyval, void* attribute_val,
              int attribute_len, attribute_destructor_routine)

int MPI_test_attribute(MPI_Attribute handle, int keyval,
              MPI_Attribute *attribute_ptr, int len)

int MPI_delete_attribute(MPI_Attribute handle, int keyval)
```

A.6 C Bindings for Process Topologies

```
int MPI_map_cart(MPI_Communicator comm_old, int ndims, int *dims,
              int *period, int *newrank)

int MPI_map_graph(MPI_Communicator comm_old, int nodes, int *index,
              int *nglist, int *newrank)

int MPI_cart(MPI_Communicator comm_old, MPI_Communicator comm_cart, ndims,
              int *dims, int *period, int reorder)

int MPI_graph(MPI_Communicator comm_old, MPI_Communicator comm_graph, nodes,
              int index, int *nglist,  reorder)

int MPI_inqmap(MPI_Communicator comm, map)

int MPI_inqgraphdims(MPI_Communicator comm, int indexlen, int nglistlen)

int MPI_inqgraph(MPI_Communicator comm, int indexdim, int nglistdim,
              int index, int *nglist)

int MPI_inqcartdim(MPI_Communicator comm, int dim)

int MPI_inqcart(MPI_Communicator comm, int maxdim, int size, int period,
              pos)
```

```
int MPI_inqrank(MPI_Communicator comm, location, int rank)

int MPI_inqloc(MPI_Communicator comm, int rank, location, int maxdim)

int MPI_partc(MPI_Communicator comm, int lostdim, MPI_Communicator new_comm)

int MPI_shift_id(MPI_Communicator comm, direction, int steps,
             int rank_source, int rank_dest)

int MPI_makdim(int nprocs, int ndims, int *dims)
```

A.7 C bindings for Environmental Inquiry

```
int MPI_validtags(int low, int high)

int MPI_num_comm(int n)

int MPI_bufferparams(int b, int n, int bt, int nt)

int MPI_exclusivebuffer(int size, int partner, MPI_Communicator comm)

int MPI_user_provides_buffer(MPI_Communicator comm, int len, void* buffer)

int MPI_errormode(newmode, MPI_Communicator comm)

MPI_Errhandler mpi_set_handler(MPI_Communicator comm, mpi_Errhandler handler)
int MPI_errhandler mpi_set_handler(MPI_Communicator comm, mpi_errhandler
             handler)

char *mpi_errorstring(errorcode)
int MPI_dumpreceivedqueue(int len, int lenout, info)

int MPI_getmachinename(int len, name)

int MPI_init()

int MPI_finalize()

int MPI_initialized()

int MPI_abort(int errorcode)
```

A.8 Fortran Bindings for Point-to-Point Communication

```
mpi_send(start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer count, datatype, dest, tag, comm, rc

mpi_recv(start, count, datatype, source, tag, comm, status, rc)
    real start(*)
    integer count, datatype, source, tag, comm, status, rc

mpi_get_source(status, source, rc)
    integer status, source, rc
```

```
mpi_get_tag(status, tag, rc)
    integer status, tag, rc

mpi_get_count(status, count, rc)
    integer status, count, rc

mpi_rsend(start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer count, datatype, dest, tag, comm, rc

mpi_ssend(start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer count, datatype, dest, tag, comm, rc

mpi_isend(handle, start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer handle, count, datatype, dest, tag, comm, rc

mpi_irsend(handle, start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer handle, count, datatype, dest, tag, comm, rc

mpi_issend(handle, start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer handle, count, datatype, dest, tag, comm, rc

mpi_irecv(handle, start, count, datatype, source, tag, comm, rc)
    real start(*)
    integer handle, count, datatype, source, tag, comm, rc

mpi_wait(handle, status, rc)
    integer handle, status, rc

mpi_test(handle, flag, status, rc)
    integer handle, flag, status, rc

mpi_waitany(count, array_of_handles, index, status, rc)
    integer count, array_of_handles(*), index, status, rc

mpi_testany(count, array_of_handles, index, status, rc)
    integer count, array_of_handles(*), index, status, rc

mpi_waitall(count, array_of_handles, array_of_status, rc)
    integer count, array_of_handles(*), array_of_status, rc

mpi_testall(count, array_of_handles, flag, array_of_status, rc)
    integer count, array_of_handles(*), flag, array_of_status, rc

mpi_iprobe(source, tag, comm, flag, status, rc)
    integer source, tag, comm, flag, status, rc

mpi_probe_count(status, datatype, count, rc)
    integer status, datatype, count, rc

mpi_probe(source, tag, comm, status, rc)
```

```
    integer source, tag, comm, status, rc

mpi_cancel(handle, rc)
    integer handle, rc

mpi_test_cancelled(status, flag, rc)
    integer status, flag, rc

mpi_create_send(handle, start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer handle, count, datatype, dest, tag, comm, rc

mpi_create_rsend(handle, start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer handle, count, datatype, dest, tag, comm, rc

mpi_create_ssend(handle, start, count, datatype, dest, tag, comm, rc)
    real start(*)
    integer handle, count, datatype, dest, tag, comm, rc

mpi_create_recv(handle, start, count, datatype, source, tag, comm, rc)
    real start(*)
    integer handle, count, datatype, source, tag, comm, rc

mpi_start(handle, rc)
    integer handle, rc

mpi_commobj_free(handle, rc)
    integer handle, rc

mpi_sendrecv(send_start, send_count, send_type, dest, send_tag, recv_start,
             recv_count, recv_type, source, recv_tag, comm, status, rc)
    integer send_start, send_count, send_type, dest, send_tag, recv_start,
    recv_count, recv_type, source, recv_tag, comm, status, rc

mpi_exchange(start, count, datatype, dest, send_tag, source, recv_tag, comm,
             status, rc)
    real start(*)
    integer count, datatype, dest, send_tag, source, recv_tag, comm,
    status, rc

mpi_type_contiguous(count, oldtype, newtype, rc)
    integer count, oldtype, newtype, rc

mpi_type_vector(count, oldtype, stride, blocklength, newtype, rc)
    integer count, oldtype, stride, blocklength, newtype, rc

mpi_type_hvector(count, oldtype, stride, blocklength, newtype, rc)
    integer count, oldtype, stride, blocklength, newtype, rc

mpi_type_indexed(count, oldtype, array_of_displacements,
             array_of_blocklengths, newtype, rc)
    integer count, oldtype, array_of_displacements(*),
    array_of_blocklengths(*), newtype, rc
```

```
mpi_type_hindexed(count, oldtype, array_of_displacements,
             array_of_blocklengths, newtype, rc)
    integer count, oldtype, array_of_displacements(*),
    array_of_blocklengths(*), newtype, rc

mpi_type_struct(count, array_of_types, array_of_displacements,
             array_of_blocklengths, newtype, rc)
    integer count, array_of_types(*), array_of_displacements(*),
    array_of_blocklengths(*), newtype, rc

mpi_address(location, address, rc)
    integer location, address, rc

mpi_type_extent(datatype, extent, rc)
    integer datatype, extent, rc

mpi_type_commit(datatype, rc)
    integer datatype, rc

mpi_type_free(datatype, rc)
    integer datatype, rc

mpi_comm_init(handle, datatype, source-dest, tag, comm, op-mode,
             persistence, rc)
    integer handle, datatype, source-dest, tag, comm, op-mode, persistence,
    rc
```

A.9 Fortran Bindings for Collective Communication

```
mpi_barrier(comm, rc)
    integer comm, rc

mpi_bcast(buffer, cnt, type, root, comm, rc)
    integer buffer, cnt, type, root, comm, rc

mpi_gather(sendbuf, sendcnt, sendtype, recvbuf, maxrecvcnt, recvcnts(*),
             recvtype, root, comm, rc)
    real(*) sendbuf, recvbuf
    integer sendcnt, sendtype, maxrecvcnt, recvcnts(*), recvtype, root,
    comm, rc

mpi_scatter(sendbuf, sendcnts(*), sendtype, recvbuf, maxrecvcnt, recvcnt,
             recvtype, root, comm, rc)
    real(*) sendbuf, recvbuf
    integer sendcnts(*), sendtype, maxrecvcnt, recvcnt, recvtype, root,
    comm, rc

mpi_allcast(sendbuf, sendcnt, sendtype, recvbuf, maxrecvcnt, recvcnts(*),
             recvtype, comm, rc)
    real(*) sendbuf, recvbuf
    integer sendcnt, sendtype, maxrecvcnt, recvcnts(*), recvtype, comm, rc
```

```
mpi_alltoall(sendbuf, sendcnts(*), sendtype, recvbuf, maxrecvcnt,
             recvcnts(*), recvtype, comm, rc)
    real(*) sendbuf, recvbuf
    integer sendcnts(*), sendtype, maxrecvcnt, recvcnts(*), recvtype, comm,
    rc

mpi_reduce(sendbuf, recvbuf, cnt, type, op, root, comm, rc)
    real(*) sendbuf, recvbuf
    integer cnt, type, op, root, comm, rc

mpi_user_reduce(sendbuf, recvbuf, cnt, type, function, root, comm, rc)
    real(*) sendbuf, recvbuf
    external function
    integer cnt, type, root, comm, rc

mpi_user_reducea(sendbuf, recvbuf, cnt, type, function, root, comm, rc)
    real(*) sendbuf, recvbuf
    external function
    integer cnt, type, root, comm, rc

mpi_allreduce(sendbuf, recvbuf, cnt, type, op, comm, rc)
    real(*) sendbuf, recvbuf
    integer cnt, type, op, comm, rc

mpi_user_allreduce(sendbuf, recvbuf, cnt, type, function, comm, rc)
    real(*) sendbuf, recvbuf
    external function
    integer cnt, type, comm, rc

mpi_user_allreducea(sendbuf, recvbuf, cnt, type, function, comm, rc)
    real(*) sendbuf, recvbuf
    external function
    integer cnt, type, comm, rc

mpi_reduce_scatter(sendbuf, recvbuf, distcnts(*), type, op, comm, rc)
    real(*) sendbuf, recvbuf
    integer distcnts(*), type, op, comm, rc

mpi_user_reduce_scatter(sendbuf, recvbuf, distcnts(*), type, function, comm,
             rc)
    real(*) sendbuf, recvbuf
    external function
    integer distcnts(*), type, comm, rc

mpi_user_reduce_scattera(sendbuf, recvbuf, distcnts(*), type, function,
             comm, rc)
    real(*) sendbuf, recvbuf
    external function
    integer distcnts(*), type, comm, rc

mpi_scan(sendbuf, recvbuf, cnt, type, op, comm, rc)
    real(*) sendbuf, recvbuf
```

```
    integer cnt, type, op, comm, rc

mpi_user_scan(sendbuf, recvbuf, cnt, type, function, comm, rc)
    real(*) sendbuf, recvbuf
    external function
    integer cnt, type, comm, rc

mpi_user_scana(sendbuf, recvbuf, cnt, type, function, comm, rc)
    real(*) sendbuf, recvbuf
    external function
    integer cnt, type, comm, rc
```

A.10 Fortran Bindings for Groups, Contexts, and Communicators

```
mpi_group_size(group, size, rc)
    integer group, size, rc

mpi_group_rank(group, rank, rc)
    integer group, rank, rc

mpi_translate_ranks(group_a, n, ranks_a, group_b, ranks_b, rc)
    integer group_a, n, ranks_a(*), group_b, ranks_b(*), rc

mpi_local_group_union(group1, group2, group_out, rc)
    integer group1, group2, group_out, rc

mpi_local_group_intersect(group1, group2, group_out, rc)
    integer group1, group2, group_out, rc

mpi_local_group_difference(group1, group2, group_out, rc)
    integer group1, group2, group_out, rc

mpi_local_subgroup(group, n, ranks, new_group, rc)
    integer group, n, ranks, new_group, rc

mpi_local_excl_subgroup(group, n, ranks, new_group, rc)
    integer group, n, ranks(*), new_group, rc

mpi_local_subgroup_ranges(group, n, ranges, new_group, rc)
    integer group, n, ranges, new_group, rc

mpi_local_subgroup_excl_ranges(group, n, ranges, new_group, rc)
    integer group, n, ranges, new_group, rc

mpi_group_dup(group, new_group, rc)
    integer group, new_group, rc

mpi_group_free(group, rc)
    integer group, rc

mpi_contexts_alloc(comm, n, rc)
    integer comm, n, rc

mpi_contexts_free(comm, n, rc)
```

```
    integer comm, n, rc

mpi_comm_size(comm, size, rc)
    integer comm, size, rc

mpi_comm_rank(comm, rank, rc)
    integer comm, rank, rc

mpi_comm_group(comm, group, rc)
    integer comm, group, rc

mpi_comm_contexts(comm, n, rc)
    integer comm, n, rc

mpi_comm_dup(comm, new_comm, rc)
    integer comm, new_comm, rc

mpi_comm_local_dup(comm, new_comm, rc)
    integer comm, new_comm, rc

mpi_comm_coll_dup(comm, new_comm, rc)
    integer comm, new_comm, rc

mpi_comm_make(sync_comm, comm_group, comm_new, rc)
    integer sync_comm, comm_group, comm_new, rc

mpi_comm_local_make(sync_comm, comm_group, comm_new, rc)
    integer sync_comm, comm_group, comm_new, rc

mpi_comm_coll_make(sync_comm, comm_group, comm_new, rc)
    integer sync_comm, comm_group, comm_new, rc

mpi_comm_coll_subset(comm, colour, key, comm_new, rc)
    integer comm, colour, key, comm_new, rc

mpi_comm_free(comm, rc)
    integer comm, rc

mpi_comm_stat(comm, status, rc)
    integer comm, status, rc

mpi_comm_peer_make(my_comm, my_leader, peer_comm, peer_rank, tag, num_comms,
             new_comms, rc)
    integer my_comm, my_leader, peer_comm, peer_rank, tag, num_comms,
    new_comms, rc

mpi_comm_peer_make_start(my_comm, my_leader, peer_comm, peer_rank, tag,
             num_comms, MPI_Comm_name make_id, rc)
    integer my_comm, my_leader, peer_comm, peer_rank, tag, num_comms,
    MPI_Comm_name make_id, rc

mpi_comm_peer_make_finish(MPI_Comm_name make_id, new_comms, rc)
    integer MPI_Comm_name make_id, new_comms, rc

mpi_comm_splitl(comm, key, nkeys, leaders, sub_comm, rc)
    integer comm, key, nkeys, leaders, sub_comm, rc
```

```
mpi_comm_name_make(my_comm, name, num_comms, new_comms, rc)
    integer my_comm, name, num_comms, new_comms, rc

mpi_comm_name_make_start(my_comm, name, num_comms, MPI_Comm_name make_id, rc)
    integer my_comm, name, num_comms, MPI_Comm_name make_id, rc

mpi_comm_name_make_finish(MPI_Comm_name make_id, new_comms, rc)
    integer MPI_Comm_name make_id, new_comms, rc

mpi_attribute_alloc(n,handle_array ,len, rc)
    integer n,handle_array,len, rc

mpi_attribute_free(handle_array, n, rc)
    integer handle_array,n, rc

mpi_get_attribute_key(keyval, rc)
    integer keyval, rc

mpi_set_attribute(handle, keyval, attribute_val, attribute_len,
             attribute_destructor_routine, rc)
    external attribute_destructor_routine
    integer handle, keyval, attribute_val, attribute_len, rc

mpi_test_attribute(handle,keyval,attribute_ptr,len, rc)
    integer handle,keyval,attribute_ptr,len, rc

mpi_delete_attribute(handle, keyval, rc)
    integer handle, keyval, rc
```

A.11 Fortran Bindings for Process Topologies

```
mpi_map_cart(comm_old, ndims, dims, period, newrank, rc)
    integer comm_old, ndims, dims(*), period, newrank, rc

mpi_map_graph(comm_old, nodes, index, nglist, newrank, rc)
    integer comm_old, nodes, index, nglist, newrank, rc

mpi_cart(comm_old, comm_cart, ndims, dims(*), period, reorder, rc)
    integer comm_old, comm_cart, ndims, dims(*), period, reorder, rc

mpi_graph(comm_old, comm_graph, nodes, index, nglist, reorder, rc)
    integer comm_old, comm_graph, nodes, index, nglist, reorder, rc

mpi_inqmap(comm, map, rc)
    integer comm, map, rc

mpi_inqgraphdims(comm, indexlen, nglistlen, rc)
    integer comm, indexlen, nglistlen, rc

mpi_inqgraph(comm, indexdim, nglistdim, index, nglist, rc)
    integer comm, indexdim, nglistdim, index, nglist, rc

mpi_inqcartdim(comm, dim, rc)
    integer comm, dim, rc
```

```
mpi_inqcart(comm, maxdim, size, period, pos, rc)
    integer comm, maxdim, size, period, pos, rc

mpi_inqrank(comm, location, rank, rc)
    integer comm, location, rank, rc

mpi_inqloc(comm, rank, location, maxdim, rc)
    integer comm, rank, location, maxdim, rc

mpi_partc(comm, lostdim, comm_new, rc)
    integer comm, lostdim, comm_new, rc

mpi_shift_id(comm, direction, steps, rank_source, rank_dest, rc)
    integer comm, direction, steps, rank_source, rank_dest, rc

mpi_makdim(nprocs, ndims, dims(*), rc)
    integer nprocs, ndims, dims(*), rc
```

A.12 Fortran Bindings for Environmental Inquiry

```
mpi_validtags(low, high, rc)
    integer low, high, rc

mpi_num_comm(n, rc)
    integer n, rc

mpi_bufferparams(b, n, bt, nt, rc)
    integer b, n, bt, nt, rc

mpi_exclusivebuffer(size, partner, comm, rc)
    integer size, partner, comm, rc

mpi_user_provides_buffer(comm, len, buffer, rc)
    integer comm, len, buffer, rc

mpi_errormode(newmode, comm, rc)
    integer newmode, comm, rc

mpi_errhandler mpi_set_handler( comm, handler, rc)
    integer comm, handler, rc

mpi_errorstring(errorcode, string, rc)
    character*(*) string
    integer errorcode, rc

mpi_dumpreceivedqueue(len, lenout, info, rc)
    integer len, lenout, info, rc

mpi_getmachinename(len, name, rc)
    chaaracter*(*) name
    integer len, rc

mpi_init(rc)
    integer rc
```

```
mpi_finalize(rc)
    integer rc

mpi_initialized(flag, rc)
    integer flag, rc

mpi_abort(errorcode)
    integer errorcode
```

MPI Function Index