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

Message Passing Interface Forum

August 14, 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 August 14, 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
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 [3], and PARMACS [10, 11]. Other important contributions have come from Zipcode [16, 17], Chimp [6, 7], PVM [8, 18], and PICL [9].

One of the objectives of this paper is to promote a discussion within the concurrent computing research community of the issues that must be addressed in establishing a practical, portable, and flexible standard for message passing. This cooperative process began with a workshop on standards for message passing held in April 1992 [19].

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 no mandate) extensions for use in heterogeneous environment.
- Allow convenient C, Fortran 77, Fortran 90, and C++ bindings for interface.
- Assume a reliable communication interface: User need not cope with communication failures. Such failures are dealt by the underlying communication subsystem.
- Focus on a proposal that can be agreed upon in 6 months.
- 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, C, Fortran 90, or 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 (this is temporarily as inclusive as possible):

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- A simple way to create processes for the SPMD model
- Bindings for Fortran 77, Fortran 90, C and C++
- A model implementation
- A formal specification.
- Process topology
- A 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

[1]

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.

[1]Version of July 30, 1993

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.

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 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.

2.6 Error Handling

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

Groups, Contexts, and Communicators

1

3.1 Introduction

We define the concepts of group, context, and communicator here. We discuss operations for how these should be used to provide safe (safer) communication in the MPI system. 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.

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

[1]Version of August 13, 1993

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.

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.

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.

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).

3.4.2 Group Management

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

3.4.3 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.

3.4.4 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.

3.4.5 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.

3.5 Context Management

3.5.1 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 3.10.6).

3.5.2 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.

3.6 Intra-Communication Communicator Management

3.6.1 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`

3.6.2 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.

3.6.3 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? .

3.7 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.

3.7.1 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().

3.7.2 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?

3.7.3 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().

3.7.4 Communicator Destructors

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

3.7.5 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.

3.7.6 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.

3.7.7 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.

3.8 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[2]. 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.

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

3.8.1 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.

3.9 Formalizing the Loosely Synchronous Model (Usage, Safety)

3.9.1 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.

3.9.2 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.

3.10 Motivating Examples

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

3.10.1 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.

3.10.2 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.

3.10.3 (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.

3.10.4 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);
```

3.10.5 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);
}
```

3.10.6 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.

3.10.7 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 4

Point to Point Communication

1

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 August 9, 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)

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	

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

MPI datatype	C datatype
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_UNSIGNED	unsigned
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	char
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:
Need to decide whether we want (for C)
`unsigned char`, `unsigned short`, `unsigned long`, `long double`
`DOUBLE COMPLEX` may not be standard Fortran 77
MPI_DOUBLE is now used both for Fortran and C. Should we use different names?

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. 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. Thus, the range of valid values for the source parameter is { 0, ... , n-1 } ∪{ 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

Discussion: This is a change from previous draft

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).

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.

information returned in it by the last receive operation that updated

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 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. 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.

Discussion:
Need to be discussed by MPIF
Alternative positions:

1. Sends are handled fairly; e.g. a posted send cannot be repeatedly overtaken by other sends. An example of an implementation that violates this: Assume that arriving messages are kept in a hash table organized by source, and that a receive with source=dontcare is handled by searching this table sequentially. Then a message posted at the end of this table may be repeatedly overtaken by messages that get posted at the head of the table. (Easy fix: rotate starting point of search.)

2. Receives are handled fairly; e.g. a posted receive cannot be repeatedly overtaken by other receives. An example of an implementation that violates this: assume that there are two different posting tables for receives with a specific source and for receives with source=dontcare; assume further that these two structures are searched in a fixed order when a message arrives. Then a receive with source=dontcare may be repeatedly overtaken by receives with source specified. (Fix: alternate search order; but this has performance problems, if we assume that one type of receive is much more frequent than the other type.)

3. Both send and receives are handled fairly.

Opinions?

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 processors 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, ierr)
  ELSE    ! rank.EQ.1
```

```
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, context)
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, 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, 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, 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, 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 communication system has sufficient buffer space to buffer **count** words of data, and the MPI implementation buffers messages.

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.

Examples:

1st 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, 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)`. 2nd 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, ierr)
END IF
```

This code is erroneous, since sender and receiver do not provide matching datatype parameters. 3rd 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, 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_CHARACTER. A buffer descriptor entry with datatype of MPI_CHARACTER can only match a Fortran variable of type `CHARACTER`; and representation conversion may occur when values of type MPI_CHARACTER 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 (non)solution means that inter-language communication relies on the vendor specific rules for inter-language calls at each site.

It may be desirable to provide information on the correspondence between MPI C datatypes (such as `MPI_FLOAT`) and MPI Fortran datatypes (such as `MPI_REAL`) with suitable environment query functions.

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, a blocking send or receive operation may only block the executing thread, not the entire process. Thus, while a thread is blocked waiting for a send or receive to complete, other threads may execute within 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.

Discussion:

Do we want to say that a blocking operation MAY or MUST only block the executing thread?

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.

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. A nonblocking communication 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 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	handle to communication object
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	handle to communication object
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	handle to communication object
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	handle to communication object
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)

The call allocates 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 sender should not update any part of a send buffer after a nonblocking send operation returns, until the send completes. A receiver should not access or update any part of a receive buffer after a nonblocking receive operation returns, until the receive completes.

4.8.3 Communication Completion

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.

Discussion:
Do we agree on the last sentence? (this implies some restriction on the implementation)

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.

Discussion:
We could dispose of the flag parameter and add an additional query function for status objects.

MPI_TEST is a local, nonblocking operation. If repeatedly called for an operation that is enabled, it must eventually succeed.

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.

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). On the other hand, the code may deadlock if the nonblocking operations are replaced by blocking ones.

4.8.4 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.

Discussion:
MPI_TESTALL **is new.**

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 successful execution of MPI_WAITANY(count, array_of_handles, index, status) has the same effect as the successful execution of MPI_WAIT(array_of_handles[i], status), where i is the value returned by index. In particular, the associated communication object 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.

Discussion:

If statement of fairness is changed in Section 4.4 then will need to change here, too.

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 -1 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 the index of that handle in the array, and returns the status of that operation in the object associated with the status. The communication object is deallocated and the handle to it in array_of_handles is set to null. In the latter case, it returns a value of -1 in index and status is undefined. Like MPI_TEST, this is a nonblocking operation, that returns immediately; and if one busy-waits with MPI_TESTANY, and some operation on the list is enabled, then MPI_TESTANY will eventually return flag = true.

Discussion:

Could have a flag parameter, rather than using index = -1 for unsuccessful return. This is more consistent and more elegant, but requires an additional parameter.

If we stick with current version, may replace -1 by an MPI constant.

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.

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 and the values of the status entries are undefined. This is a nonblocking operation that returns immediately; if one busy waits with MPI_TESTALL, and all operations in the list become enabled, then MPI_TESTALL will eventually return flag = true.

Example:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank < 2)
  THEN
    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.EQ.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.

Discussion:

Is the last paragraph agreed?

We may want to allow null pointers in an array_of_handles This will allow to loop with the same array_of_handles, until all communications have completed. On the other hand, this will encourage potentially inefficient code.

I added the requirement that the handed to a communication that completes is set to null. This prevents the occurrence of dangling references, but add some overhead.

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.

Discussion:

Made one change from approved draft on probe: removed the datatype argument from probe. If generalized datatype are adopted, then it seems more natural to provide the argument in the status decode function.

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	handle to communicator
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.

The call to MPI_PROBE does not carry a datatype argument: the message type may not be known when the probe is executed. Accordingly, the probe call does not return in the status object the length of the probed message, and MPI_GET_COUNT cannot be used after a call to MPI_PROBE to query on the length of the probed message. (Remember that the length of a message, in elements, depends on the types of these elements; some implementations may not carry in the transmitted message type information.) 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 length of 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.

Discussion:

Message passing in MPI can be implemented without appending type information to messages. A message is merely a string of bytes and the interpretation of these bytes into a sequence of typed elements is done using the datatype information provided by the local communication call. The ability to use such an implementation strategy is deemed to be an important goal. In such an implementation, when a message arrives, it is not known how many elements it contains, or even how much storage is needed to receive that message (because of possible representation conversion in a heterogeneous environment). The length of the incoming message can be computed by MPI_PROBE_LEN from the byte length of the message and from the datatype parameter.

The current solution saves us the need for one additional word per message that would otherwise be needed to transfer the message length (in elements) with the message.

Need be discussed by MPIF

We may want a call to MPI_PROBE_COUNT to be significant not just after probe, but also after receive, for consistency reasons.

It's tempting to merge MPI_GET_COUNT and MPI_PROBE_COUNT into one call. E.g., always provide a datatype argument, even though it is redundant after a receive. But then we impose an additional burden on regular receives.

We may want to relax type restrictions so that datatype = MPI_BYTE could be used as an argument to MPI_PROBE_COUNT, event if the message was sent with another datatype; this will allow to count the number of bytes received, and use this count in a malloc.

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, datatype, status)

IN	source	source rank, or MPI_ANY_SOURCE (integer)
IN	tag	tag value, or MPI_ANY_TAG (integer)
IN	comm	handle to communicator
IN	datatype	assumed type of data in message (status)
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.

Discussion:
CANCEL has been changed since last draft

MPI_CANCEL(handle)

IN	handle	handle to communication object

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 status 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	return status object
IN	flag	(logical)

Returns flag = true if the communication associated with the return status object was cancelled successfully. In such a case, all other fields of status (such as length 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 values of the return status.

4.10 Persistent communication objects

Discussion:

there are some changes here not yet approved by MPIF

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 handle to initiate and complete messages. The communication handle 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	handle to communication object
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	handle to communicator object

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

Discussion:

Two changes:

1. buffer_handle is replaced with start, count, datatype, to be in sync with the new datatype proposal.

2. Took out persistence. Does not seem very useful here. Instead, I introduce in the last section, a "universal" communication function, which would not usually used directly by programmers, but can be used to define the other communication functions.

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	handle to communication object
IN	start	initial address of send buffer
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	handle to communicator object

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	handle to communication object
IN	start	initial address of send buffer
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	handle to communicator object

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	handle to communication object
OUT	start	initial address of receive buffer
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	handle to communicator object

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

MPI_START(handle)

INOUT	handle	handle to communication object

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 4.8.4. 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_COMM_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_COMM_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_COMM_FREE(handle)

INOUT	handle	handle to communication object

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, and the object may not be deallocated when the call returns. It is permissible to call MPI_COMM_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 try to reuse the handle for a subsequent communication.

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

Discussion:

This section has not yet been approved by MPIF.

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 sends, next receives, odd receives first, next sends) 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, recv_start, recv_count, recv_type, source, 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)
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	tag	message tag (integer)
IN	comm	handle to communicator
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.

Discussion:
For a shift it's more natural to have same type and same length both for send and receive; for a remote procedure call, the additional freedom makes sense.

MPI_EXCHANGE(start, count, datatype, dest, source, 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	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	handle to communicator
OUT	status	status object

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

Discussion:

I omitted the ready and synchronous modes for send/receive and exchange. Also, I omitted nonblocking send/receive and exchange. Do we want any of those?

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, count, MPI_REAL, 0, recv_buff, count, MPI_REAL,
          1, tag, comm, status, ierr)
  ELSE
    CALL MPI_SENDRECV(send_buff, count, MPI_REAL, 1, recv_buff, 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

Discussion: This section has not been reviewed by MPIF.

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.

Discussion:
It would be nice to have MPI_PROCNULL "equivalenced" to -1 and to group_size, so that, when a shift is executed in a group the first and last processors automatically communicate with the null process.

A choice is to say that a source/destination with rank -1 or group_size is always the null process. We then lose some error catching capability.

Another choice is to say that only for send-receive and exchange.

A third choice is to live without this nicety. It means that one still have to execute conditional code: (if myrank=0 then myneighbor=mpi_procnull). But this code need be executed only when the communication pattern is set, and the conditional boundary code is not needed in the inner loop.

4.13 Derived datatypes

Discussion:
This is a new section, and the material has not yet been discussed by MPIF.

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 count 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 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 been 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 types
- 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.

Let

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

be such a general datatype, where $type_i$ are basic types, and $disp_i$ are displacements. This general datatype, 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 this communication buffer will contain a sequence of n values, where value i has type $type_i$. The general datatype determines the **type signature** of the message (i.e. the number of values carried by the message and the basic type of each value).

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 4.13.3 the case where the second argument count has value > 1.

The predefined basic datatypes presented in Section 4.2.1 are particular cases of a general datatype. Thus, MPI_INT is a predefined handle to the datatype $\{(\texttt{int}, 0)\}$, 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. I.e., if

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

$disp_r = \min_j disp_j$ and $disp_s = \max_j disp_j$ then

$$extent(Datatype) = disp_s + sizeof(type_s) - disp_r; \tag{4.1}$$

if furthermore, $type_i$ requires alignment to a byte address that is is a multiple of k_i, then $extent(Datatype)$ is rounded up to the next multiple of $\max_i k_i$.

Example: Assume that

$$Type1 = \{(\texttt{double}, 0), (\texttt{char}, 8)\}$$

(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.

4.13.1 Datatype constructors

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

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count (integer)
IN	oldtype	handle to input datatype object
OUT	newtype	handle to output datatype object

newtype is the datatype obtained by concatenating count copies of oldtype. Padding may be added to satisfy alignment requirements of the underlying architecture.

Example: let oldtype be a handle to the datatype

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

with extent 16, and let count = 3. The resulting 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 oldtype is a handle to the datatype

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

with extent *extent*. Then newtype is a handle to the datatype with $\texttt{count} \cdot \texttt{n}$ entries defined by:

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

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

MPI_TYPE_VECTOR is a more general constructor that allows replication of a datatype into locations that consist of equally spaced contiguous blocks of equal size – block sizes and block displacements are all multiples of the old type extent.

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

IN	count	number of blocks (integer)
IN	oldtype	handle to input datatype object
IN	stride	number of elements between start of each block (integer)
IN	blocklen	number of elements in each block (positive integer)
OUT	newtype	handle to output datatype object

Example: Assume, again, that oldtype points to the type

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

with extent 16. A call to MPI_TYPE_VECTOR(2, oldtype, 4, 3, newtype) will create the datatype

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

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

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

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

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

In general, assume that oldtype is a handle to the datatype

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

with extent *extent*. The newly created datatype is a sequence of length $\texttt{count} \cdot \texttt{blocklen} \cdot n$ with entries:

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

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

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

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

$$(type_0, disp_0 + (\texttt{stride} + \texttt{blocklen} - 1) \cdot extent), ...,$$

$$(type_{n-1}, disp_{n-1} + (\texttt{stride} + \texttt{blocklen} - 1) \cdot extent), ...,$$

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

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

$$(type_0, disp_0 + (\texttt{stride} \cdot (\texttt{count} - 1) + \texttt{blocklen} - 1) \cdot extent), ...,$$

$$(type_{n-1}, disp_{n-1} + (\texttt{stride} \cdot (\texttt{count} - 1) + \texttt{blocklen} - 1) \cdot extent)\}$$

Discussion:

I changed count from being the number of entries (or replicates), as in the old draft, to being the number of blocks. The reason is that in the next calls I allow block lengths to vary; it is easy to say what it means to have n blocks of sizes $b_0, ..., b_{n-1}$; it is harder to explain what it means to have n elements filling up successive positions in blocks of sizes $b_0, b_1, ...$

We loose the ability to use MPI_TYPE_VECTOR to create a partially filled last block (the more general MPI_TYPE_INDEXED function is needed); we gain consistency with the next calls, and convenience in these calls.

Opinions?

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

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 4.13.4. (`H` is used for "heterogeneous").

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

IN	count	number of blocks (integer)
IN	oldtype	handle to input datatype object
IN	stride	number of bytes between start of each block (integer)
IN	blocklen	number of elements in each block (positive integer)
OUT	newtype	handle to output datatype object

MPI_TYPE_INDEXED is a more general function that allows replication of an old datatype into a sequence of contiguous blocks, where block lengths and block displacements can all differ – but are all multiples of the old type extent.

MPI_TYPE_INDEXED(count, oldtype, array_of_indices, array_of_blocklen, newtype)

IN	count	number of blocks (integer)
IN	oldtype	handle to input datatype object
IN	array_of_indices	displacement for each block, in multiples of oldtype extent (array of integer)
IN	array_of_blocklen	number of elements per block (array of integer)
OUT	newtype	handle to output datatype object

Example: let oldtype be a handle to the type

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

with extent 16. Let `I = (64, 0)` and let `B = (3, 1)`. A call to MPI_TYPE_INDEXED(2, oldtype, I, B, newtype) returns in `newtype` a handle to the type

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

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

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

In general, assume that oldtype is a handle to the datatype

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

with extent *extent*. Let B be the array_of_blocklen parameter and I be the array_of_indices parameter. The newly created type returned by newtype is the sequence of $n \cdot \sum_{i=0}^{\mathtt{count}-1} \mathsf{B[i]}$ entries

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

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

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

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

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

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

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

and

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

The function MPI_TYPE_HINDEXED is identical to MPI_TYPE_INDEXED, except that block displacements in array_of_indices are specified in bytes, rather than in multiples of the old type extent.

MPI_TYPE_HINDEXED(count, oldtype, array_of_indices, array_of_blocklen, newtype)

IN	count	number of blocks (integer)
IN	oldtype	handle to input datatype object
IN	array_of_indices	byte displacement of each block (array of integer)
IN	array_of_blocklen	number of elements in each block (array of integer)
OUT	newtype	handle to output datatype object

`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_indices, array_of_blocklen, newtype)

IN	count	number of blocks (integer)
IN	array_of_types	type of elements in each block (array of datatype objects)
IN	array_of_indices	byte displacement of each block (array of integer)
IN	array_of_blocklen	number of elements in each block (array of integer)
OUT	newtype	handle to output datatype object

Example: Let type1 be a handle to the type

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

with extent 16. Let `I = (0, 16, 26)`, `B = (2, 1, 3)` and `T = (MPI_FLOAT, type1, MPI_BYTE)`. Then a call to MPI_TYPE_STRUC(3, T, I, B, newtype) returns in newtype a handle to the datatype

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

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_blocklen parameter and I be the array_of_indices parameter. Then the newly created datatype is the sequence of length $\sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]} \cdot extent_i$ with entries

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

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

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

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

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

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

Discussion:
I am open to suggestions for better names to these functions.

4.13.2 Additional 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.

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

MPI_TYPE_EXTENT(datatype, extent)

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

As defined now, the extent of a datatype goes from its first occupied byte to its last occupied byte. It is often convenient to define a datatype that has a "hole" at its beginning and/or its end; or a datatype with entries that extend beyond the datatype "extent". This allows to use the functions MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, or MPI_TYPE_INDEX to replicate a more general pattern; examples of such a usage are provided in Section 4.13.4. 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 signature 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 `I = (-3, 0, 6)`; `T = (MPI_LB, MPI_INT, MPI_UB)`, and `B = (1, 1, 1)`. Then a call to MPI_TYPE_STRUC(3, T, I, B, type1) creates a new datatype that has an extent of 9 (from -3 to 6), 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

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

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

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

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

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

Then

$$extent(Datatype) = ub(Datatype) - lb(Datatype);$$

Furthermore, if $type_i$ requires alignment to an address that is a multiple of k_i, then $extent(Datatype)$ is further rounded up to the nearest multiple of $\max_i k_i$. The formal definitions given for the various datatype constructors apply now, with the amended definition of **extent**.

A datatype object has to be **committed** before it can be used in a communication. A committed datatype object cannot be used to build new derived datatypes. The system may use a different internal representation for committed datatype object so as to facilitate communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism.

MPI_TYPE_COMMIT(intype, outtype)

IN	intype	handle to datatype object to be committed
OUT	outtype	handle to committed datatype object

There is no need to commit basic datatypes; they are "pre-committed". They are the exception to the rule that committed datatypes cannot be used to form derived datatypes.

Discussion:
1. Do we really need a commit here?
2. Do we want one INOUTargument, or two distinct arguments?
3. Are committed and uncommitted objects of the same type? (The answer must be yes, if it is an INOUTargument.)
4. If we have only one INOUTargument, does a commit change the datatype object, or does it create a new object and return a pointer to it in datatype? I.e., if we have another pointer to the same uncommitted object, can we continue to use it?

We prohibit the use of committed datatype in the construction of new datatypes in order to allow the commit function to change a datatype representation into one suitable for communication, and perhaps unsuitable for recursive datatype construction. This restriction could be relaxed by assuming that the "uncommitted" datatype representation is kept alongside the "committed" one. With such an implementation MPI_TYPE_COMMIT could have only one argument, and only one type of datatype object is needed. Such representation could also simplify the implementation of a datatype "flatten" function (e.g., for debugging).

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.

Discussion:
If we have different datatypes for committed and uncommitted, then we have 2 free functions (or never free uncommitted datatype)

4.13.3 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 defined by the sequence $\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}$, (with empty 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 datatype used in a receive operation may not have overlapping entries; otherwise the outcome of the receive operation is undefined.

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 defined by each datatype should be identical. Furthermore, each entry in the send buffer should be of type that matches the type of the corresponding entry in type1 and each entry in the receive buffer should of a type that matches the type of the corresponding entry in type2 (with type matching defined as in Section 4.5). Thus, type matching is defined by the "flat" structure of a datatype, and does not depend on its defining expression.

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.

Type matching requirements imply that (nonempty) entries of different types in a datatype cannot overlap, unless they coincide and have the same basic type, or one of them have basic type byte.

The length of a message received (i.e., the value returned when the receive status object is decoded by the function MPI_GET_LEN) is equal to the number of basic entries received. 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 length of the message received will be four. In the general case the basic entries need not be all of the same type.

Discussion:
An alternative definition that finds favor with many members of the subcommittee is that the length of the message received is measured in terms of subunits of the receiving datatype. Thus, if four reals are received using type4, the length of the received message is four; if they are received using type22, the length of the received message is two. With such a definition, it is required that the arriving message fills an integer number of subunits.

The rationale is that the these are the units that are meaningful to the programmer. Consider the case of an array of structures that is passed in a message (example 4.1 below). Then one might want to enforce the restriction that only complete structures are passed; and is natural to return in the receive status the number of structures received, rather than this number multiplied by 14, the number of entries per structure.

Consider, on the other hand, the case of a sequence of reals that is received in a 3D array section (defined as in the first example below). We may want to receive an arbitrary number of reals, and the 1D and 2D subsections may not carry any significance for the purpose of this communication.

Opinions?

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, such as would be produced by a call to MPI_CONTIGUOUS(count, datatype, newtype), and a count argument of one. The same holds true for the other communication functions that have a datatype argument.

4.13.4 Examples

The following examples illustrate the use of derived datatypes.

First example: extract a section of a 3D matrix.

```
      REAL a(100,100,100), e(9,9,9)
      INTEGER oneslice, twoslice, threeslice, finaltype, sizeofreal

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

      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, finaltype, ierr)
```

```
      CALL MPI_SENDRECV(a(1,3,2), 1, finaltype, MPI_ME, e, 9*9*9,
                MPI_REAL, MPI_ME, 0, MPI_ALL, status, ierr)
```

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

```
      REAL a(100,100), b(100,100)
      INTEGER  index(100), blocklen(100), ltype, finaltype

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

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

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

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

Third example: Transpose a matrix

```
      REAL a(100,100), b(100,100)
      INTEGER row, xpose, finaltype, sizeofreal

C     transpose matrix a onto b


      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, finaltype, ierr)

C     send matrix in row-major order and receive in column major order
      CALL MPI_SENDRECV( a, 1, finaltype, MPI_ME, b, 100*100,
               MPI_REAL, MPI_ME, 0, MPI_ALL, status, ierr)
```

Another approach to the transpose problem:

```
      REAL a(100,100), b(100,100)
      INTEGER  index(2), blen(2), type(2), row, row1, finaltype, sizeofreal
```

```
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
      index(1) = 0
      index(2) = sizeofreal
      type(1)  = row
      type(2)  = MPI_UB
      blen(1)  = 1
      blen(2)  = 1
      CALL MPI_TYPE_STRUC( 2, type, index, blen, row1)

      CALL MPI_TYPE_COMMIT( row1, finaltype, ierr)

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

Fourth example: manipulate an array of structures.

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

struct Partstruct particle[1000];

           /* build datatype describing first array entry */

MPI_datatype Particletype;
MPI_datatype stype[3] = {MPI_int, MPI_double, MPI_char};
int          sblock[3] = {1, 6, 7};
int          sindex[3];

MPI_address( (void *)particle, sindex);
MPI_address( (void *)particle[0].d, sindex+1);
MPI_address( (void *)particle[0].b, sindex+2);
MPI_type_struct( 3, stype, sindex, sblock, Particletype);
/* Particletype describes first array entry -- using absolute
   addresses */
```

```
        /* 4.1: send the entire array */

MPI_datatype Type1;
MPI_commit( Particletype, Type1);
MPI_send( MPI_bottom, 1000, Type1, dest, tag, comm);


        /* 4.2: send the entries with index value zero,
          preceded by the number of such entries */

MPI_datatype Zparticles;   /* datatype describing all particles
                                    with index zero (needs to be recomputed
                                    if indices change) */
MPI_datatype Ztype, Type2;

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


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

/* create datatype for particles with index zero */
MPI_type_index( j, Particletype,  zindex, zblock, Zparticles);

/* prepend particle count */
MPI_address((void *)&j, zzindex);
zzindex[1] = MPI_bottom;
zztype[0] = MPI_int;
zztype[1] = Zparticles;
MPI_type_struc(2, zztype, zzindex, zzblock, Ztype);

MPI_type_commit( Ztype, Type2);
MPI_send( MPI_bottom, 1, Type2, dest, tag, comm);


        /* A possibly 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) then
    {
    for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
    zindex[j] = i;
    zblock[j] = k-i;
    j++;
    i = k;
    }
MPI_type_index( j, Particletype, zindex, zblock, Zparticles);


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

MPI_datatype Allpairs;      /* datatype for all pairs of coordinates */
MPI_datatype Type3;

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, Type3);
MPI_send( particle.d, 1, Type3, dest, tag, comm);

      /* an alternative solution to 4.3 */

MPI_datatype Onepair;   /* datatype for one pair of coordinates, with
                          the extent of one particle entry */
MPI_datatype Type4;

int indexp[3];
int typep[3] = {MPI_lb, MPI_double, MPI_ub};
int blockp[3] = {1, 2, 1};

MPI_address( (void *)particle, indexp);
MPI_address( (void *)particle[0].d, indexp+1);
MPI_address( (void *)(particle+1), indexp+2);

MPI_type_struc( 3, typep, indexp, blockp, Onepair);
MPI_commit( Onepair, Type4);
MPI_send( MPI_bottom, 1000, Type4, dest, tag, comm);
```

Discussion:
The examples carry two implicit assumptions on the way padding is done in an array of structures.

First, that padding is done in the same way in the successive array entries, so that all entries have an identical memory layout. E.g., we assume that displacement from `particle[i].b` to `particle[j].b` is the same as the displacement from `particle[i].d` to `particle[j].d`.

Second, that the amount of padding added is, in some sense, minimal. I.e., if the most stringent alignment requirement for a basic component of a structure is alignment to multiples of 2^k, then no contiguous padding space has size greater or equal to 2^k. This assumption is implicit in the definition of **extent**.

I don't think either assumptions are mandated by C of Fortran standards, although I would be very surprised to see an implementation that does not fulfill these conditions. If I am wrong (e.g., if some compilers always align structures to double word boundaries, even if they have no double word components), then the definition of extent should be revisited.

4.13.5 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 host program variable.
2. The start parameter of a communication function evaluates to a valid address, when passed as argument a host program variable.
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 array or record 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_STRUC.

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 now the extent of the arrays and records in the host program.

4.14 Universal communication functions

Although the number of communication functions defined in this chapter is fairly large, it is possible to construct all of them from a much smaller number of primitive functions. The construction is outlined in this section. The construction can be used to build a portable implementation of MPI, although it is expected that more efficient implementation will implement directly most or all of the functions in this section. Also, this construction can be used in order to derive the behavior of MPI point to point communication from the behavior of a small subset of functions. 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, but merely creates a new copy of it.

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, Index[1]);
Block[1] = count;
MPI_TYPE_STRUCT( 1, Type, Index, Block, Newtype)
MPI_TYPE_COMMIT( Newtype, datatype1)
MPI_SEND( MPI_BOTTOM, 1, datatype1, 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).

(If the operation MPI_TYPE_COMMIT) is not assumed to be trivial, then the above construction cannot be used; this adds one more primitive function, namely MPI_TYPE_COMMIT, to our list, and introduces small changes in the construction.)

It is convenient to to make explicit the **persistence** attribute of communication objects. A persistent communication object need be explicitly deallocated by a MPI_COMM_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 universal function for the creation of communication objects:

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

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

The communication involves only one element, with an absolute address.

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_COMM_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 array-ofhandles of length one.

The function MPI_TESTALL 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_IS_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_STRUC; 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_STRUC
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

[1]

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

[1]Version of August 9, 1993

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 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.

Discussion: The last proposal contained two layers of functions one (contiguous) buffer for the "contiguous" functions, and an array of buffers for the noncontiguous ones.

The latest pt2pt chapter no longer contains a buffer descriptor handle, instead there is a general datatype parameter which can describe arbitrary structures. I have incorporated this feature into the collective functions, since we have agreed that the collective routines must be build-able from the pt2pt routines.

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.

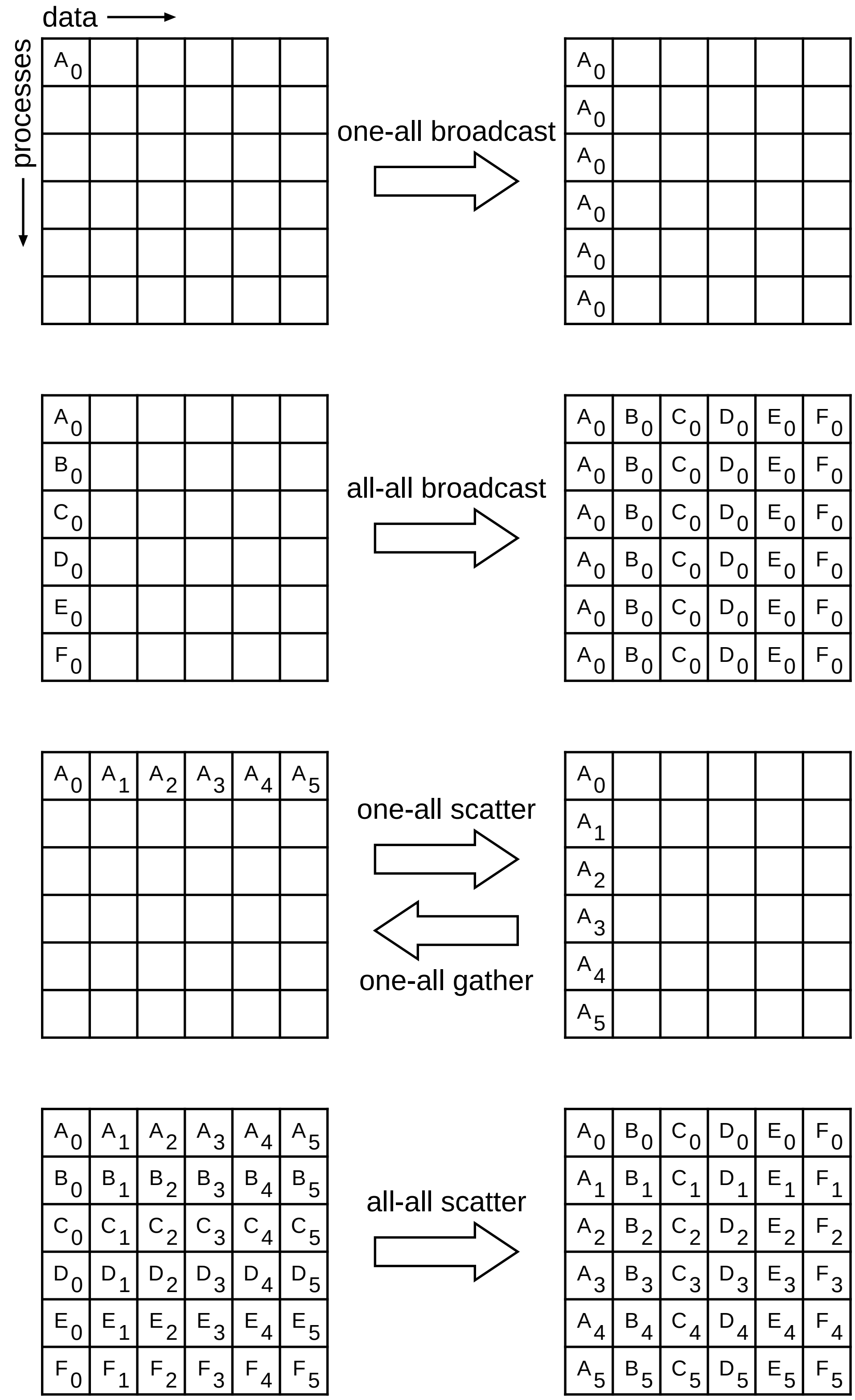

Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the one-all broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

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, recvcnt, 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	recvcnt	number of elements in receive buffer (integer)
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, each consisting of sendcnt elements. The data sent from process with rank i is stored in the i-th block. The function is called with the same values for sendcnt, sendtype, root, and comm at all participating processes.

5.4.3 Scatter

MPI_SCATTER(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm)

IN	sendbuf	address of send buffer – significant only at root
IN	sendcnt	number of elements in send buffer (integer)
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer.
IN	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, each consisting of recvcnt 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 recvcnt, 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, recvcnt, 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	recvcnt	number of elements in receive buffer (integer)
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); All send buffers have the same number of elements. Each process concatenates the incoming messages, in the order of the senders' ranks, and stores them in its receive buffer. The routine is called by all members of the group using the same arguments for sendcnt, sendtype, 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, sendcnt, sendtype, recvbuf, recvcnt, 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	recvcnt	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements
IN	comm	communicator handle

Each process in the group sends the `i`-th portion of its send buffer to the process with rank `i` (itself included). All messages sent from one process to another have the same length. The send buffer of each process is partitioned into MPI_GSIZE consecutive blocks, each consisting of sendcnt elements. The `i`-th block is sent to the `i`-th process in the group. Each process concatenates the incoming messages, in the order of the senders' ranks, and store them in its receive buffer. The routine is called by all members of the group using the same arguments for sendcnt, 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.

Discussion: Do we want to have completely general versions of the above data move routines? Without buffer descriptors this is nearly impossible in fortran because of variable length argument list or array_of_addresses requirements. In C it is merely ugly. For example the general Alltoall would be: MPI_ALLTOALL(array_of_sendadr, array_of_sendcnt, array_of_sendtype, array_of_recvadr, array_of_recvcnt, array_of_recvtype, comm) It is not clear if we would want to use an array_of_types or a more complex datatype with end_of_buffer markers inserted.

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.

Missing:

The last approved draft has an additional unitsize parameter in MPI_USER_REDUCE: Each element of invec or inoutvec corresponds to unitsize elements of the input (output) buffer. This allows, for example, to pass to MPI_USER_REDUCE a vector of real numbers, and have function treat each pair of real numbers as one complex number. I am not sure we need this if we approve the proposal to use buffer types: It we do so it will be easy to define the buffer to consists of pairs of real numbers, say.

MPI also 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.

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

Process Topologies

6.1 Introduction

A process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In 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.

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 grid. Usually a process does not have to know the identifiers of *all* processes in the 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.

Additionally, the user–defined process structure can be exploited by the system in the assignment of processes to hardware processors, if this helps to improve the communication performance on a given machine. How this mapping is done in detail is outside the scope of this standardization proposal, since it is too much dependent on the particular hardware. The description of a "virtual process topology", however, 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 [5, 4].

6.2 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 in this paper 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 [10, 11] 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.

6.3 Embedding in MPI

The support for virtual topologies has to be fully integrated in the framework of MPI. In particular, the virtual topology functions do not interfere with the functions for process group creation and with those 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 paper 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 process group creation function MPI_CREATE_GROUP plus a few topology specific mapping functions. In principle this is sufficient for defining topologies and use 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.

6.4 Overview of the proposed MPI functions

This chapter assumes the availability of the following MPI features which are defined in other chapters:

1. An existing process group (e.g., the ALL group) can be partitioned and/or renumbered using

   MPI_CREATE_GROUP(gid_old, gid_new, group_key, order_key)

IN	gid_old	group id of parent group
OUT	gid_new	group id of newly created subgroup of which the calling process is a member
IN	group_key	integer value used for the partitioning
IN	order_key	integer value used for determining the ranks in the new group

   This function is called by all processes in group *gid_old* loosely synchronously. It creates as many different subgroups as there are different values of *group_key*. The ranking in the newly created groups is determined by the *order_key* values, with the process ranks in the parent group for tie breaking. If there is a standardized internal information caching facility, it can be used by MPI routines to store and retrieve information related to a process group. This facility was proposed in the Context Subcommittee. High–level topology functions store structure information for a process group, which is then later used by topology inquiry functions.

   In case the caching mechanism does not become part of MPI, the topology information can be retrieved by group id in an implementation–dependent way, which each vendor can do as is most convenient.

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.

6.4.1 Low–level topology functions

On the lowest level groups are created by the general function MPI_CREATE_GROUP. Using the input argument *newrank* 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(gid_old, ndims, dims, period, newrank)

IN	gid_old	group id of parent group
IN	ndims	number of dimensions of Cartesian structure
IN	dims(*)	number of processes in each coordinate direction
IN	period(*)	periodicity specification in each coordinate direction
OUT	newrank	reordered rank of the calling process

The corresponding function for general graph structures is

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

IN	gid_old	group id of parent group
IN	nodes	number of graph nodes
IN	index(*)	define graph structure, see text
IN	nglist(*)	define graph structure, see text
OUT	newrank	reordered rank of the calling process

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.

Discussion: It cannot be ruled out completely that topology definition functions internally require communication. This could become a necessity if a parallel algorithm is used for doing the mapping. If so, then it might be necessary to pass them a communicator instead of/in addition to the group id.

In this case we also have to say that a low–level topology definition function is a collective operation on the group. If it is not called by all group processes loosely synchronously, deadlock might occur.

6.4.2 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 group creation routine. 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 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_RANK returns MPI_PROCNULL if called with the identifier of the new group.

MPI_CART is defined as

MPI_CART(gid_old, gid_cart, ndims, dims, period, reorder)

IN	gid_old	group id of parent group
OUT	gid_cart	group id of Cartesian group
IN	ndims, dims(*), period(*)	definition of Cartesian structure, (for details see the definition of MPI_MAP_CART)
IN	reorder	ranking may be reordered or not

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

For graph structures the corresponding routine is

MPI_GRAPH(gid_old, gid_graph, nodes, index, nglist, reorder)

IN	gid_old	group id of parent group
OUT	gid_graph	group id of graph group
IN	nodes, index(*), nglist(*)	definition of graph structure, (for details see the definition of MPI_MAP_GRAPH)
IN	reorder	ranking may be reordered or not

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(*))

For a graph structure no information has to be stored, because the graph node number is the rank in the graph group, and the size of the graph is the number of processes in that group.

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.

6.4.3 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(gid, map)

IN	gid	group id
OUT	map	topology type of group with identifier `gid`

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_GRAPHDIMS and MPI_INQGRAPH look up topology information:

MPI_GRAPHDIMS(gid, indexlen, nglistlen)

IN	gid	group id of group with graph structure
OUT	indexlen	number of entries in vector `index`
OUT	nglistlen	number of entries in vector `nglist`

The information provided by MPI_GRAPHDIMS 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(gid, indexdim, nglistdim, index, nglist)

IN	gid	group id of group with graph structure
IN	indexdim	length of vector index(*) in the calling program
IN	nglistdim	length of vector nglist(*) in the calling program
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_INQDIM and MPI_INQCART look up topology information:

MPI_INQDIM(gid, dim)

IN	gid	group id of group with Cartesian structure
OUT	dim	dimension of the Cartesian structure

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

IN	gid	group id of group with Cartesian structure
IN	maxdim	length of vectors size(*), period(*), and pos(*) in the calling program
OUT	size(*)	number of processes for each coordinate direction
OUT	period(*)	periodicity yes/no for each coordinate direction
OUT	pos(*)	position of calling process in Cartesian structure

The inquiry for Cartesian topology information is thus split up into two calls. The reason is that otherwise the user might not know a save 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(gid, location, rank)

IN	gid	group id of group with Cartesian structure
IN	location	coordinate vector of process
OUT	rank	process rank

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

MPI_INQLOC(gid, rank, location, maxdim)

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

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

6.4.4 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(gid, lostdim, gid_new)

IN	gid	group id of group with Cartesian structure
IN	lostdim	direction perpendicular to resulting hyper–planes
OUT	gid_new	group identifier of the hyper–plane group in which the calling process is contained

MPI_PARTC uses the function MPI_CREATE_GROUP 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 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(gid, direction, steps, rank_source, rank_dest)

IN	gid	group id of group with Cartesian structure
IN	direction	coordinate direction of shift
IN	steps	step length ($>$ 0: upwards shift, $<$ 0: downwards shift)
OUT	rank_source	rank of source process
OUT	rank_dest	rank of destination process

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
IN	ndims	number of Cartesian dimensions
INOUT	dims	number of processes in each coordinate direction

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.

6.5 A simple example

The example in figure 6.1 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for example the Poisson equation, is to be solved on a rectangular domain. First the processes organize themselves in a two–dimensional structure. Each process then inquires for the ranks of its neighbors in the four directions. The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine `relax`.

In each relaxation step each process computes new values for the solution grid function at all points owned by the process. Then the values at interprocess boundaries have to be exchanged with neighboring processes. For example, the exchange subroutine might contain a call like MPI_SENDC(...,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 gid, gid_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 /
C     Create a grid structure in ALL group and inquire about own position
      gid = MPI_ALL
C     Set process grid size and periodicity
      call MPI_MAKDIM(gid, ndims, dims)
      period(1) = MPI_PER
      period(2) = MPI_PER
      call MPI_CART (gid, gid_cart, ndims, dims, period, reorder, rank)
      call MPI_INQCART (gid_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 (gid, neigh_def, neigh_rank(1))
      neigh_def(1) = i+1
      neigh_def(2) = j
      call MPI_INQRANK (gid, neigh_def, neigh_rank(2))
      neigh_def(1) = i
      neigh_def(2) = j-1
      call MPI_INQRANK (gid, neigh_def, neigh_rank(3))
      neigh_def(1) = i
      neigh_def(2) = j+1
      call MPI_INQRANK (gid, 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, neigh_rank, num_neigh)
10    continue
      call output (u)
      end
```

Figure 6.1: Set–up of process structure for two–dimensional parallel Poisson solver.

Section 7

Language Binding

7.1 Introduction

This chapter defines the rules for MPI language binding in general and for Fortran 77, Fortran 90, ANSI C, 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.

Discussion: There was some informal discussion of this issue at the last meeting. Is it possible to say something stronger?

7.2 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 **strings**, **double precision**, **real**, and **integer** data in Fortran. Null-terminated **strings**, **double**, **real**, **short**, **char**, and **long** are supported in C. In addition, a special **byte** type is supported in both languages. This type allows communication to be performed with no type conversion at all. In C, the definitions of named constants, function prototypes, and aggregate type information must be supplied in

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

Figure 7.1: This is an example of calling a routine with mismatched formal and actual parameters.

an include file `mpi.h`.

7.3 Fortran 77 Binding Issues

There are several points were 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 31, not six, significant characters.
- Identifiers may contain underscores after the first character.
- The types of the formal parameter list may not match the calling parameter list. (An example is given in figure 7.3.)
- Although not required, it is strongly suggested that named constants be provided in an include file.

7.4 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.

7.5 Specific Bindings

Discussion: What would be the best way to present the following information? One big alphabetical list? In order of presentation in the individual chapters? By chapter but alphabetical within chapter? C and Fortran mixed or separate?

7.5.1 C bindings for Point-to-Point Communication

These are in alphabetical order.

```
int MPI_append_contiguous( MPI_Bd bd, void* start, int count,
             MPI_Datatype datatype )

int MPI_append_hindexed( MPI_Bd bd, void* start, int count,
             int *index_array, MPI_Datatype datatype )

int MPI_append_hvec( MPI_Bd bd, void* start, int count, int stride,
             int lenblk, MPI_Datatype datatype )

int MPI_append_indexed( MPI_Bd bd, void* start, int count, int *index_array,
             MPI_Datatype datatype )

int MPI_append_vec( MPI_Bd bd, int count, int stride, int lenblk,
             MPI_Datatype datatype )

int MPI_associated(MPI_Handle handle, MPI_Handle handle_type )

int MPI_cancel( MPI_Handle handle, int flag )

int MPI_commit_buffer( MPI_Bd bd )

int MPI_create_status( MPI_Handle handle )

int MPI_create_buffer( MPI_Bd *bd, MPI_LIFETIME persistence )

int MPI_free( MPI_Handle handle )

int MPI_get_len( int count, MPI_Status *status, MPI_Bd bd )

int MPI_init_recv ( MPI_Handle handle, MPI_Bd bd, int source, MPI_Tag tag,
             MPI_Communicator context, MPI_LIFETIME persistence )

int MPI_init_rsend ( MPI_Handle handle, MPI_Bd bd, int dest, MPI_Tag tag,
             MPI_Communicator context, MPI_LIFETIME persistence )

int MPI_init_send ( MPI_Handle handle, MPI_Bd bd, int dest, MPI_Tag tag,
             MPI_COMunicator context, MPI_LIFETIME persistence )

int MPI_init_ssend ( MPI_Handle handle, MPI_Bd bd, int dest, MPI_Tag tag,
             MPI_Communicator context, MPI_LIFETIME persistence )

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

int MPI_irecv( MPI_Handle handle, MPI_Bd bd, int source, MPI_Tag tag,
             MPI_Communicator context )

int MPI_irecvc( MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int source, MPI_Tag tag,
             MPI_Communicator context, MPI_Status *status )

int MPI_irsend ( MPI_Handle handle, MPI_Bd bd, int dest, MPI_Tag tag,
             MPI_Communicator context )
```

```
int MPI_irsendc ( MPI_Handle handle, void* start, int count,
             MPI_Datatype datatype, int dest, MPI_Tag tag,
             MPI_Communicator context )

int MPI_isend ( MPI_Handle handle, MPI_Bd bd, int dest, MPI_Tag tag,
             MPI_Communicator context )

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

int MPI_issend ( MPI_Handle handle, MPI_Bd bd, int dest, MPI_Tag tag,
             MPI_Communicator context )

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

int MPI_probe( int source, MPI_Tag tag, MPI_Communicator context,
             MPI_Datatype datatype, MPI_Status *status )

int MPI_query( MPI_Handle handle, int count, int source, MPI_Tag tag)

int MPI_recv( MPI_Bd bd, int source, MPI_Tag tag, MPI_Communicator context,
             MPI_Status *status )

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

int MPI_rsend ( MPI_Bd bd, int dest, MPI_Tag tag, MPI_Communicator context )

int MPI_rsendc ( void* start, int count, MPI_Datatype datatype, int dest,
             MPI_Tag tag, MPI_Communicator context )

int MPI_send ( MPI_Bd bd, int dest, MPI_Tag tag, MPI_Communicator context )

int MPI_sendc ( void* start, int count, MPI_Datatype datatype, int dest,
             MPI_Tag tag, MPI_Communicator context )

int MPI_ssend ( MPI_Bd bd, int dest, MPI_Tag tag, MPI_Communicator context )

int MPI_ssendc ( void* start, int count, MPI_Datatype datatype, int dest,
             MPI_Tag tag, MPI_Communicator context )

int MPI_start( MPI_Handle handle )

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

int MPI_Statusany ( MPI_Handle *handle_list, int count, int index,
             MPI_Status *status )

int MPI_wait ( MPI_Handle handle, MPI_Status *status )

int MPI_waitall( MPI_Handle *handle_list, int count, list_of_return_handles)

int MPI_waitany ( MPI_Handle *handle_list, int count, int index,
             MPI_Status *status )
```

7.5.2 C Bindings for Collective Communication Chapter

These are in order of presentation in the Collective Communication chapter.

```
int MPI_comm_init( MPI_Group group )

int MPI_rank( MPI_Group group, int rank )

int MPI_gsize( MPI_Group group, int size )

int MPI_barrier( MPI_Group group )

int MPI_synch( MPI_Group group, MPI_MAP map )

int MPI_bcast( MPI_Bd bd, MPI_Group group, int root )

int MPI_bcastc( buf, int len, MPI_Datatype datatype, MPI_Group group,
             int root )

int MPI_gather( MPI_Bd inbd, list_of_MPI_Bd outbds, MPI_Group group, int root
             )

int MPI_gatherc( void* inbuf, void* outbuf, int len, MPI_Datatype datatype,
             MPI_Group group, int root )

int MPI_scatter( list_of_MPI_Bd inbds, MPI_Bd outbd, MPI_Group group,
             int root )

int MPI_scatterc( void* inbuf, void* outbuf, int len,
             MPI_Datatype datatype, MPI_Group group, int root )

int MPI_allscatter( MPI_Bd inbds[], MPI_Bd outbds[], MPI_Group group )

int MPI_allscatterc( void* inbuf, void* outbuf, int len,
             MPI_Datatype datatype, MPI_Group group )

int MPI_allcast( MPI_Bd inbds[], MPI_Bd outbds[], MPI_Group group )

int MPI_allcastc( void* inbuf, void* outbuf, int len,
             MPI_Datatype datatype, MPI_Group group )

int MPI_reduce( MPI_Bd inbd, MPI_Bd outbd, MPI_Group group, int root,
             MPI_Op op )

int MPI_reducec( void* inbuf, void* outbuf, int len, MPI_Datatype datatype,
             MPI_Group group, int root, MPI_Op op )

int MPI_user_reduce( MPI_Bd inbd, MPI_Bd outbd, MPI_Group group, int root,
             void *userfunc(), int unitsize )

int MPI_user_reducec( void* inbuf, void* outbuf, int len,
             MPI_Datatype datatype, MPI_Group group, int root,
             void *userfunc(), unitsize )

int MPI_allreduce( MPI_Bd inbd, MPI_Bd outbd, MPI_Group group, MPI_Op op )

int MPI_allreducec( void* inbuf, void* outbuf, int len,
             MPI_Datatype datatype, MPI_Group group, MPI_Op op )
```

```
int MPI_user_allreduce( MPI_Bd inbd, MPI_Bd outbd, MPI_Group group,
             void *userfunc())

int MPI_user_allreducec( void* inbuf, void* outbuf, int len,
             MPI_Datatype datatype, MPI_Group group, void *userfunc())

int MPI_scan( MPI_Bd inbd, MPI_Bd outbd, MPI_Group group, MPI_Op op )

int MPI_scanc( void* inbuf, void* outbuf, int len, MPI_Datatype datatype,
             MPI_Group group, MPI_Op op )

int MPI_user_scan( MPI_Bd inbd, MPI_Bd outbd, MPI_Group group,
             void *userfunc() )

int MPI_user_scanc( void* inbuf, void* outbuf, int len,
             MPI_Datatype datatype, MPI_Group group, void *userfunc())
```

7.5.3 Fortran Bindings for Point-to-Point Routines

```
mpi_append_contiguous( bd, start, count, datatype, rc )
    real start(*)
    integer bd, count, datatype, rc

mpi_append_hindexed( bd, start, count, index_array, datatype, rc )
    real start(*)
    integer index_array(*)
    integer bd, count, datatype, rc

mpi_append_hvec( bd, start, count, stride, lenblk, datatype, rc )
    real start(*)
    integer bd, count, stride, lenblk, datatype, rc

mpi_append_indexed( bd, start, count, index_array, datatype, rc )
    real start(*)
    integer index_array(*)
    integer bd, count, datatype, rc

mpi_append_vec( bd, count, stride, lenblk, datatype, rc )
    integer bd, count, stride, lenblk, datatype, rc

mpi_associated( handle, handle_type, rc )
    integer handle, handle_type, rc

mpi_cancel( handle, flag, rc )
    integer handle, flag, rc

mpi_commit_buffer( bd, rc )
    integer bd, rc

mpi_create_buffer( bd, persistence, rc )
    integer bd, persistence, rc

mpi_create_status( handle, rc )
    integer handl, rce
```

```
mpi_free( handle, rc )
    integer handl, rce

mpi_get_len( count, status, bd, rc )
    integer count, status, bd, rc

mpi_init_recv ( handle, bd, source, tag, context, persistence, rc )
    integer handle, bd, source, tag, context, persistence, rc

mpi_init_rsend ( handle, bd, dest, tag, context, persistence, rc )
    integer handle, bd, dest, tag, context, persistence, rc

mpi_init_send ( handle, bd, dest, tag, context, persistence, rc )
    integer handle, bd, dest, tag, context, persistence, rc

mpi_init_ssend ( handle, bd, dest, tag, context, persistence, rc )
    integer handle, bd, dest, tag, context, persistence, rc

mpi_iprobe( source, tag, context, flag, type, status, rc )
    integer source, tag, context, flag, type, status, rc

mpi_irecv( handle, bd, source, tag, context, rc )
    integer handle, bd, source, tag, context, rc

mpi_irecvc( handle, start, count, datatype, source, tag, context, status,
             rc )
    real start(*)
    integer handle, count, datatype, source, tag, context, status, rc

mpi_irsend ( handle, bd, dest, tag, context, rc )
    integer handle, bd, dest, tag, context, rc

mpi_irsendc ( handle, start, count, datatype, dest, tag, context, rc )
    real start(*)
    integer handle, count, datatype, dest, tag, context, rc

mpi_isend ( handle, bd, dest, tag, context, rc )
    integer handle, bd, dest, tag, context, rc

mpi_isendc ( handle, start, count, datatype, dest, tag, context, rc )
    real start(*)
    integer handle, count, datatype, dest, tag, context, rc

mpi_issend ( handle, bd, dest, tag, context, rc )
    integer handle, bd, dest, tag, contex, rct

mpi_issendc ( handle, start, count, datatype, dest, tag, context, rc )
    real start(*)
    integer handle, count, datatype, dest, tag, context, rc

mpi_probe( source, tag, context, datatype, status, rc )
    integer source, tag, context, datatype, status, rc

mpi_query( handle, count, source, tag, rc )
    integer handle, count, source, tag, rc
```

```
mpi_recv( bd, source, tag, context, status, rc )
    integer bd, source, tag, context, status, rc

mpi_recvc (start, count, datatype, source, tag, context, statu, rcs)
    real start(*)
    integer count, datatype, source, tag, context, status, rc

mpi_rsend ( bd, dest, tag, context, rc )
    integer bd, dest, tag, context, rc

mpi_rsendc (start, count, datatype, dest, tag, context, rc )
    real start(*)
    integer count, datatype, dest, tag, context, rc

mpi_send ( bd, dest, tag, context, rc )
    integer bd, dest, tag, context, rc

mpi_sendc (start, count, datatype, dest, tag, context, rc )
    real start(*)
    integer count, datatype, dest, tag, context, rc

mpi_ssend ( bd, dest, tag, context, rc )
    integer bd, dest, tag, context, rc

mpi_ssendc (start, count, datatype, dest, tag, context, rc )
    real start(*)
    integer count, datatype, dest, tag, context, rc

mpi_start( handle, rc )
    integer handle, rc

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

mpi_statusany ( handle_list, count, index, status, rc )
    integer handle_list(*)
    integer count, index, status, rc

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

mpi_waitall( handle_list, count, list_of_return_handles, rc )
    integer handle_list(*)
    integer count
    integer list_of_return_handles(*, rc)

mpi_waitany ( handle_list, count, index, status, rc )
    integer handle_list(*)
    integer count, index, status, rc
```

7.5.4 Fortran Bindings for Collective Communciation Chapter

These are not done yet.

```
mpi_comm_init( group, rc )
```

```
mpi_rank( group, rank, rc )

mpi_gsize( group, size, rc )

mpi_barrier( group, rc )

mpi_synch( group, map, rc )

mpi_bcast( buffer_handle, group, root, rc )

mpi_bcastc( buf, len, type, group, root, rc )

mpi_gather( inhandle, list_of_outhandles, group, root, rc )

mpi_gatherc( inbuf, outbuf, inlen, type, group, root, rc )

mpi_scatter( list_of_inhandles, outhandle, group, root, rc )

mpi_scatterc( inbuf, outbuf, len, type, group, root, rc )

mpi_allscatter( list_of_inhandles, list_of_outhandles, group, rc )

mpi_allscatterc( inbuf, outbuf, len, type, grou, rcp)

mpi_allcast( inhandle, list_of_outhandles, group, rc )

mpi_allcastc( inbuf, outbuf, len, type, grou, rcp)

mpi_reduce( inhandle, outhandle, group, root, op, rc )

mpi_reducec( inbuf, outbuf, len, type, group, root, op, rc )

mpi_user_reduce( inhandle, outhandle, group, root, function, unitsize, rc )

mpi_user_reducec( inbuf, outbuf, len, type, group, root, function,
             unitsize, rc )

mpi_allreduce( inhandle, outhandle, group, op, rc )

mpi_allreducec( inbuf, outbuf, len, type, group, op, rc )

mpi_user_allreduce( inhandle, outhandle, group, function, rc )

mpi_user_allreducec( inbuf, outbuf, len, type, group, function, rc )

mpi_scan( inhandle, outhandle, group, op, rc )

mpi_scanc( inbuf, outbuf, len, type, group, op, rc )

mpi_user_scan( inhandle, outhandle, group, function, rc )

mpi_user_scanc( inbuf, outbuf, len, type, group, function, rc )
```

Section 8

Environmental Management and Inquiry

8.1 Introduction

This chapter contains MPI-specific routines for interacting with the environment in which MPI programs run. The chapter opens with a discussion of MPI routines for initialization of the MPI environment. The chapter includes a large number of enquiry functions, by which an MPI program can determine features of the environment in which it is running in order to tune itself. There is also a discussion of functions that are not specific to MPI, such as timing functions, that may be provided by other standards (e.g. POSIX). For completeness, we discuss signals, although perhaps this belongs in the chapter on correctness.

8.1.1 Program Startup Issues

The MPI standard should provide enough information to specify an entire, source code portable program. To do this, it is necessary to specify how parallel programs start and end. This is the analogue of `PROGRAM MAIN ... END` in Fortran and `main(argc,argv){...exit(rc);}`. This proposal does not deal with methods for spawning tasks during the execution of an MPI program, or for handling explicitly MIMD models (beyond the usual way of starting a Turing machine on all nodes and sending the appropriate tape). This is not meant to suggest that MPI not have such interfaces, just that we are only proposing an SPMD model.

It should be possible to write a program that is source-language portable to any system, particularly if it requires only a simple interface to the operating system. An example is an environment that starts up a program with a fixed number of processors. Without this, it is not possible to write *any* portable programs.

However, no standard should restrict programs to a particular interface. For example, some applications may want to dynamically add and remove tasks during the execution of an application. We leave it to others on the committee to propse a mechanism for this case.

Below, we discuss three different approaches; we note that all of these have been tried in some existing system or systems.

MPI_INIT

The seemingly simplest approach is to use an explicit initialization routine, `MPI_Init`. There are several problems with this approach. The major one is that the parallel part of the program is the code that "follows" the MPI_Init. This is potentially error prone; further, it can be difficult to implement on some systems.

MPI_MAIN

Another approach is to insist that the entire program be run in parallel; a simple way to do this is to replace the "main" program with a routine. In C, this could be MPI_Main(argc, argv). The linking step then provides a main that handles all appropriate startup (including the processing of command line arguments where appropriate for information on the number of tasks etc.) and then calls the user's MPI_main routine.

Note that the MPI_main interface can be implemented with MPI_call through

```
int main( argc, argv )
int argc;
char **argv;
{
return MPI_call( argc, argv, MPI_main );
}
```

MPI_CALL

Another approach is to provide a routine that runs a user-defined routine in parallel. For example, MPI_Call(argc, argv, routine) can be used to run routine(argc, argv) in parallel. A generalization of this would allow the running of a particular routine on a defined group: MPI_Call(groupid, argc, argv, routine). This is more flexible than the MPI_Main approach but retains the simplicity of running a well-defined program unit (the routine) in parallel.

Note that in this case, program termination is handled by having (a) the routine exit and (b) the calling routine exit. For example

```
    irc = MPI_Call( argc, argv, worker );
    exit( irc );
...
    int worker( argc, argv )
    int argc; char **argv;
    {
...
    if (error condition) return 1;
    ...
    return 0;
    }
```

All three of these approaches are currently in use.

These are not the most general methods. For example, these provide no mechanism for dynamically creating (remote spawn) tasks on other processors. Also, there is no direct support for MIMD programming. Finally, there is no support for adding an already-running process to the MPI collection of processes.

In the case of heterogeneous environments, it is necessary to specify the set of workstations to be used. Our only requirement is that this be done, in the C/POSIX case, through command-line arguments (thus maintaining source-code portability). We could specify an extensible, common format for specifying the machines, executables, and resource limits, but, to avoid additional debate, we suggest only that the command line option `-mpi` be reserved for future extensions.

8.2 MPI Program Startup Proposal–MPI_INIT

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`.

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:

```
start_ctx = MPI_init( ... args ... );
```

The `start_ctx` can then be used in place of the `all_ctx` (context of all processes). Multiple calls to `MPI_init` (say first by a user-code and then by a software package called by the user's code) have meaning; they set up the context that will be used as the context of all processes.

Discussion: Several other approaches were discussed, including a replacement main program and a subroutine-call interface. I've included the `MPI_init` version because it seemed to be the least objectionable and since workable versions of the other two may be written using `MPI_init`.

I'm not sure what arguments `MPI_init` should take. One argument should be the initial context; if null, then the "all context" is assumed. There is some reason to pass the command line arguments `argc,argv` to `MPI_init` in order to allow it to look for MPI-specific arguments (such as the number and name of processors to use in a heterogeneous environment), but this may be too controversial.

We should also discuss the other possibilities. One, `MPI_main`, is a main program replacement (users use `MPI_main` instead of `main` in C and `PROGRAM name` in Fortran). Another approach is `MPI_call` that calls a specified routine.

8.3 MPI-Specific Enquiry Functions

These routines are specific to the operation of MPI.

The first two routines are absolutely essential; they are:

MPI_rank My process id relative to a group

MPI_size The number of process id's in a group

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

MPI_ValidTags The range (or other description) of valid tags

MPI_NumGroups Maximum number of groups

MPI_NumCntxs Maximum number of contexts

MPI_BufferSize Buffer space for unsent/unreceived messages

MPI_BufferManagement Buffer management type

Note that all of these could return "not applicable."

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.

More detailed routines for buffer management may also be provided. These could include:

MPI_BufferSizePairs Between pairs of processors

MPI_BufferSizeAll Between "me" and all others

MPI_ExclusiveBuffer Dedicate a buffer of a specified size between a selected pair of processors

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.

There is also some desire for

MPI_IOmode Whether IO is per node or per group/context

MPI_ErrorMode Various choices of error handling

Both of these, since they have global effect, are somewhat undesirable. They should return the previous mode; multithreaded applications will have to be very careful with these.

It is important to be able to determine, both at run time and compile time, which of the "optional" capabilities the MPI implementation supports. These include

MPI_Has_Ready_Receiver Whether ready-receiver communication is simulated with regular sends and receives.

MPI_Has_Nonblocking Whether nonblocking operations are simulated with blocking sends and receives

MPI_Has_Heterogeneous Whether heterogeneous processing is supported (that is, this MPI works with heterogeneous systems)

In addition, 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. Operations should include

MPI_UnpackBufferDescriptor Takes a buffer and returns a context and (possibly) a number of members. Individual routines can

- **BufferDescType** Given a context, return the kind of buffer descriptor element (contiguous etc).
- **MPI_UnpackContig** Given a context, return the data. Advance the context.
- **MPI_UnpackIndex** Ditto of indexed
- **MPI_UnpackBlock** Ditto for blocks.

MPI_DumpMessageQueue Takes a length and a user-allocated array, it return number of elements available (may be greater than length) and information on the messages in the format includes (tag, context, source, size)

MPI_UnpackContext Takes a context and returns members of group/context

MPI_StatusOfHandles I've forgotten what this is for.

Note that some of these are tough to implement directly or efficiently in Fortran.

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.

8.3.1 Parallel programming

This section includes routines that relate to parallel programming but are not MPI-specific. These may include such things as:

MPI_GetNbrs Return the "neighbors" in the underlying physical topology

MPI_GetPhysNode Return a unique specifier for the actual (as opposed to virtual) node. This should be 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 "not applicable" is allowed but strongly discouraged.

GetTopology Return physical network topology. Predefined values include `MPI_mesh2d`, `MPI_hypercube`, and `MPI_omeganetwork`.

MPI_GetDistance Return the "distance" from one processor to another

MPI_GetDiameter Return the maximum "distance" from one processor to another.

Each of these routines may return "not applicable;" it is hoped that quality implementations will try and provide as much information as possible to the user. In addition, some of the "get" routines must be interpreted as providing approximate answers. For example, if groups and contexts are allocated from the user's address space, then any return value will be approximate.

The exact meaning of "distance" is left to the implementation. In a system with only local connections, such as a mesh or hypercube, the distance should be the number of edges in the connection graph that a message would travel along. In a system with a flat interconnect, such as an omega or butterfly network, the distance could be zero from a node to itself and either one or the number of stages in the network for all other nodes.

8.3.2 non-MPI

This section contains routines that are *not* part of MPI but are included for completeness. These will include various routines for resource usage, time-of-day, etc. The POSIX interfaces will be summarized here.

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.

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 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 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 group that can be created with the subset is an allowed argument in any subset function where a group is an appropriate argument. However, some of the more complex mechanisms for group creation have been removed from the subset.

Users may use 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 implemented on multiple platforms.

Discussion:
At the June meeting we decided to try and make a single subset. It was felt that having a multilevel subset would be more complex and potentially confusing. Thus, an item is either in or out of the subset. This will only be reconsidered if this binary choice becomes a serious problem.

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 standard 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 feature(s) 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.

Discussion:
Should it be a criteria that the rest of MPI be able to be layered on top so users might get early (though lower performance) access to the complete standard? This is aimed at allowing easy and efficient layering of the rest so that a "machine independent" package could be created without having the developer jump through lots of hops. This might happen if a few functions were left out that didn't meet the other criteria. However, some felt it was not a strong enough reason to leave it in and may lead to poor MPI implementations of these layered routines. At the June meeting it was decided that unless this turns out to be important, it will not be added to the text.

10.3 Subset Functionality

Discussion:
Most of the material in this section is either substantially reworked or new. It hopefully reflects the changes made in many of the chapters since the June meeting. As such, it has not been discussed by the MPI group and still needs polishing. I would like to agree to the concepts at this point and then polish up the presentation. Appropriate text will be added after we agree to what is in/out of the subset.

10.3.1 Point to Point Functionality

The table below contains a summary by section of which features of the point to point communications 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	——
4.6	Data Conversion	——	“All”
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_ADDRESS
		MPI_FREE	MPI_TYPE_EXTEND

In addition to these specific functions, all the features and criteria of point to point communication hold for the subset. Thus, 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.

Discussion:
At the June meeting we voted to remove the multiple completion tests from the subset

Discussion:
At the June meeting we voted to remove INIT, START, etc. This is carried over in the removal of the persistent communication objects.

Discussion:
send-recv is new. It is used in several systems now and is somewhat common practice. Users coding this often make a mistake. The procnull seems easy and of great use if we have send-recv. I left these in the subset. comments?

Discussion:
Derived datatypes are new. We previously voted to allow the contiguous buffer versions and limit buffer operations to only one append. The current draft does not have these two versions but uses the generalized datatype which includes the basic datatypes as special cases. The above table allows for the non-"h" versions so that arrays and sparse matrix people can get what they need (this also is closest to what we had before). The heterogeneous cases (hvector, hindexed) seem very easy to code but are another set of functions. They allow elements of a C type structure to be pulled out in the subset. What we decide will also affect the additional functions that are included/excluded. We need to discuss the right way to limit this functionality in the subset.

10.3.2 Collective Communication Functionality

The table below contains a summary by section of which features of 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
		(All ops)	MPI_USER_REDUCEA
		MPI_ALLREDUCE	MPI_USER_ALLREDUCE
			MPI_USER_ALLREDUCEA
5.5.2	Scan	——	MPI_SCAN
5.5.2			MPI_USER_SCAN
5.5.2			MPI_USER_SCANA

A collective communication operates on a communicator. This defines the scope of the operation. A collective operation can accept any communicator that can be created within the subset. As stated previously, the limitations will be made through the mechanisms available to create and manipulate communicators.

Discussion:
There are a lot of data move functions. I could not think of a way to draw a line to exclude some. Do we want all of these in the subset. How often are ALLCAST, ALLSCATTER used?

Discussion:
For the subset do we want both the REDUCE and ALLREDUCE versions?

Discussion:
We voted at the June meeting to remove scan from the subset

10.3.3 Topology Functionality

Discussion:
One of the new features of MPI is process topologies. As such it would be nice to have them in the subset. Since a legitimate remapping is to do nothing it can be argued that it is not hard to solve the mapping problem on a first pass. However, to implement something that gains speed for the user will take effort. We have, so far, avoid putting things in the subset if it was believed that the initial implementation would be poor and thus potentially reflect badly on that part of MPI. Given all of this, it is my inclination to exclude the entire topology chapter from the subset. Comments?

If the topology section were to be included in the subset, it would be possible to only include the grid/tori functions and remove the more general graph functions. This would simplify the initial implementation and probably cover a large percentage of applications.

10.3.4 Language Binding

Discussion:
I assume we want to support both the F77 and C binding in the subset. The Fortran 90 and C++ bindings are done through F77 and C and thus are not an issue for the subset.

10.3.5 Environmental Management and Inquiry

Discussion:
Clearly the startup mechanism must be in the subset. We also need MPI_rank and MPI_size. After this it is up for grabs. I don't think any of the others are essential to the subset. For simplification reasons maybe we should remove all the other calls. Comments?

10.3.6 Profiling Functionality

Discussion:
The profiling chapter is easy to implement but doubles the number of functions. As such, it increase the work that must be done by an implementer though this could easily be automated. It does not seem crucial to other parts of the subset nor a significant new feature. Not sure what to do here but from a simplification stand point it would be best to also drop this chapter.

10.3.7 Maps, Groups and Contexts

Discussion:
We have not discussed a single unified proposal yet for this chapter. Until we pin down to a greater extent what will be in the full standard I am delaying doing anything for the subset. Several people have expressed that contexts are a significant new feature in MPI and we should try and get them into the subset.

10.3.8 Subset Testing

Discussion:
There has been discussion of creating a test suite for MPI. Would it be possible/make sense to create a subset test suite so that these early versions could be tested and not simply have people say that the full test suite will obviously fail in some ways?

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

[4] 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.

[5] 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.

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

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

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

[9] 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.

[10] R. Hempel. The ANL/GMD macros (PARMACS) in fortran for portable parallel programming using the message passing programming model – users' guide and reference manual. Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, November 1991.

[11] R. Hempel, H.-C. Hoppe, and A. Supalov. Parmacs–6.0 library interface specification. Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, December 1992.

[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.

MPI Function Index