DRAFT DOCUMENT FOR A STANDARD MESSAGE-PASSING INTERFACE

Scott Berryman, *Yale Univ*
James Cownie, *Meiko Ltd*
Jack Dongarra, *Univ. of Tennessee and ORNL*
Al Geist, *ORNL*
Bill Gropp, *ANL*
Rolf Hempel, *GMD*
Bob Knighten, *Intel*
Rusty Lusk, *ANL*
Steve Otto, *Oregon Graduate Inst*
Tony Skjellum, *Missisippi State Univ*
Marc Snir, *IBM T. J. Watson*
David Walker, *ORNL*
Steve Zenith, *Kuck & Associates*

Date Published: March 1993

Contents

1 Introduction to MPI **1**
- 1.1 Overview and Goals . . . 1
- 1.2 Who Should Use This Standard? . . . 2
- 1.3 What Platforms Are Targets For Implementation? . . . 2
- 1.4 What Is Included In The Standard? . . . 2
- 1.5 What Is Not Included In The Standard? . . . 3

2 Point to Point Communication **4**
- 2.1 Introduction . . . 4
- 2.2 Data Types . . . 4
  - 2.2.1 Handle . . . 4
  - 2.2.2 State . . . 5
  - 2.2.3 Named constants . . . 5
  - 2.2.4 Choice . . . 6
- 2.3 Processes . . . 6
- 2.4 Contexts . . . 7
  - 2.4.1 Context Operations . . . 8
  - 2.4.2 Error Handling . . . 10
- 2.5 Messages . . . 11
  - 2.5.1 Data . . . 11
  - 2.5.2 Envelope . . . 11
- 2.6 Data Buffers . . . 12
  - 2.6.1 Data Conversion . . . 14
- 2.7 Receive Criteria . . . 15
- 2.8 Communication Mode . . . 16
- 2.9 Communication Objects . . . 16
  - 2.9.1 Communication Object Creation . . . 17
  - 2.9.2 Communication Start . . . 18
  - 2.9.3 Communication Completion . . . 18
  - 2.9.4 Multiple Completions . . . 19
- 2.10 Blocking Communication . . . 20
- 2.11 Nonblocking Communication . . . 21
- 2.12 Block Sending Operations . . . 21
- 2.13 Correctness . . . 23
  - 2.13.1 Order . . . 23
  - 2.13.2 Progress and Fairness . . . 23

3 Collective Communication **26**
- 3.1 Introduction . . . 26
- 3.2 Group Functions . . . 27
- 3.3 Communication Functions . . . 29
- 3.4 Synchronization . . . 29
- 3.5 Data move functions . . . 29
- 3.6 Global Compute Operations . . . 36
- 3.7 Correctness . . . 40

4 Process Topologies **43**
- 4.1 Introduction . . . 43

4.2 Virtual Topologies 43
4.3 Embedding in MPI 44
4.4 Overview of the proposed MPI functions 44
4.5 A simple example 45

5 Language Binding **47**

6 Formal Semantics **48**

7 Environmental Inquiry **49**

8 Profiling **50**
8.1 Objectives 50
8.2 Logic of the design 50
8.3 Examples 50
8.3.1 Profiler implementation 50
8.3.2 MPI library implementation 51
8.4 Requirements 52
8.5 Outstanding issues 52

9 Communication Contexts **53**
9.1 References 54

A Process Topology Routines **56**

DRAFT DOCUMENT FOR A STANDARD MESSAGE-PASSING INTERFACE

Scott Berryman, *Yale Univ*

James Cownie, *Meiko Ltd*

Jack Dongarra, *Univ. of Tennessee and ORNL*

Al Geist, *ORNL*

Bill Gropp, *ANL*

Rolf Hempel, *GMD*

Bob Knighten, *Intel*

Rusty Lusk, *ANL*

Steve Otto, *Oregon Graduate Inst*

Tony Skjellum, *Missisippi State Univ*

Marc Snir, *IBM T. J. Watson*

David Walker, *ORNL*

Steve Zenith, *Kuck & Associates*

Abstract

We don't have an abstract yet.

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 by Bala, Kipnis, Snir and colleagues [1, 2], Intel's NX/2 [16], Express [15], nCUBE's Vertex [13], P4 [4], and PARMACS [11, 12]. Other important contributions have come from Zipcode [17, 18], Chimp [8, 9], PVM [3, 19], and PICL [10].

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 [20].

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.

NEED to add some words about MIMD environments and threads.

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

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.

2. Point to Point Communication

Marc Snir
William Gropp and Ewing Lusk

2.1. Introduction

This section is a draft of the current proposal for point-to-point communication. It does not yet include a description of the Fortran 77 and C bindings.

I have tried to indicate, wherever appropriate, gaps and unresolved issues, using small type.

Discussion: The following subsections of the introduction contain general information on the design of MPI procedures. The material should be moved to a general introduction for the entire document.

2.2. Data Types

2.2.1. Handle

MPI procedures use at various places *handles*. Handles are used to access opaque objects. Such object can be created, updated and destroyed only by by calling suitable MPI procedures, and providing the handle as parameter. Opaque objects hide from the user the internal representation used for various MPI objects, thus allowing to have similar calls in C and Fortran, allowing to overcome problems with the typing rules in these languages, and allowing for future extension of their functionality. Handles are of type `void *` in C and of type `integer` in Fortran.

An opaque object can be *persistent* or *ephemeral*. A persistent object persists until destroyed by an explicit operation. An ephemeral object is good for a single use; thus an ephemeral object associated with a communication operation disappears once this operation is completed (or once this object is not needed anymore for the completion of the operation).

An opaque object is created by a call to `MPI_CREATE`, and destroyed by a call to `MPI_FREE`. Additional MPI functions are available to create, access and update specific opaque objects.

MPI_CREATE(handle, type, persistence)

OUT handle handle to object

IN type state value that identifies the type of object to be created (e.g., `MPI_COMMUNICATION`, `MPI_BUFFER`, `MPI_CONTEXT`, etc.).

IN persistence state value; either `MPI_PERSISTENT` or `MPI_EPHEMERAL`.

MPI_FREE(handle)

IN handle handle to object

An object can be destroyed only if there is no pending operation that is using this object; after successful return of the routine, the handle is undefined.

MPI_ASSOCIATED(handle, type)

IN handle handle to object

OUT type state

Returns the type of the object the handle is currently associated with, if such exists. Returns the special type `MPI_NULL` if the handle is not currently associated with any object.

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

List of handles

An MPI call may need a parameter that is a *list of handles*. In C, such list will be a record with one component being the length of the list, the other components being an array of pointers. In Fortran, the list will be an array of integers, the first one of which is the length of the list.

Discussion: The mechanism for opaque objects used here follows the POSIX Fortran binding standard. An alternative choice is to have different type declarations for each type of opaque object. Then, opaque objects are created/destroyed like regular variables, rather than by MPI calls; they are still accessed and updated only via MPI functions.

2.2.2. 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_CREATE` routine has a state type parameter with values `MPI_PERSISTENT` and `MPI_EPHEMERAL`.

An `enumeration` declared in an included MPI.h file will be used in C for state datatypes. The Fortran 77 mechanism needs to be decided.

Discussion: Named integer constants can be used in Fortran 90, using the `PARAMETER` mechanism. The constant declarations can be made available via an `INCLUDE` file. Fortran 77 does not seem to offer any convenient mechanism. One possibility is to specify explicit integer values, and allow the use of named constants with those Fortran 77 compilers that support them conveniently. Another possibility is to use character strings, rather than integers.

2.2.3. 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 `DONTCARE` value. 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 DONTCARE) will be outside the regular range. The range of regular values can be queried, and sometimes set, using environment inquiry or environment setting functions (Section 7). The special values are provided by named constant, that are made available via an MPI.h include file in a C binding.

Discussion: Need to agree on a Fortran mechanism for named constants (see the discussion above).

Implementers should try to detect illegal uses of "special values". Thus, the use of the `DONTCARE` value to tag a message sent should be flagged as an error.

2.2.4. Choice

MPI functions sometimes use parameters with a *choice* (or union) data type. I.e., 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. In C, a formal parameter of type `void *` will be used, with an actual pointer parameter. in Fortran, we shall cheat.

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 compiler either don't check type consistency of calls to external routines, or support a special mechanism to link foreign (e.g., C) routines. I suggest that we accept this nonconformity with 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.

2.3. Processes

An MPI program is executed by several autonomous processes that execute each 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. In particular, it is assumed that message buffers at distinct processors are disjoint.

MPI does not specify the execution model for each process. A process can be sequential, or can be multithreaded, with threads possibly executing concurrently. Care has been taken to make MPI "thread-safe", by avoiding the use of implicit global states.

The initial allocation of processes to an MPI computation and their binding to physical processors is not specified by the program itself. It is expected that vendors will provide mechanisms to do so either at load time or at run time. Such mechanisms will allow to specify 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, although it is intended to be consistent with such extension. 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 context (group), so that, effectively, processes are identified by consecutive integers. Absolute, system-wide unique process id's are (will be) needed only if dynamic process creation is to be supported (in such eventuality we propose to use handles to opaque *process structures* for that purpose).

2.4. Contexts

Discussion: This section contains a proposal for use of contexts that will subsume groups. It borrows heavily on the current group proposal. This proposal has not yet been discussed in MPI meetings.

A **context** consists of:

- A set of processes that currently belong to the context (possibly all processes, or a proper subset).
- A **ranking** of the processes within that context, i.e., a numbering of the processes in that context from 0 to $n-1$, where n is the number of processes in that context.

A process may belong to several contexts at the same time.

Any interprocess communication occurs within a context, and messages sent within one context can be received only within the same context. A context is specified using a *context handle* (i.e., a handle to an opaque object that identifies a context). Context handles cannot be transferred for one process to another; they can be used only on the process where they where created.

Follows examples of possible uses for contexts.

Loosely synchronous library call interface

Consider the case where a parallel application executes a "parallel call" to a library routine, i.e., where all processes transfer control to the library routine. If the library was developed separately, then one should beware of the possibility that the library code may receive by mistake messages send by the caller code, and vice-versa. To prevent such occurrence one might use a barrier synchronization before and after the parallel library call. Instead, one can allocate a different context to the library, thus preventing unwanted interference. Now, the transfer of control to the library need not be synchronized.

Functional decomposition and modular code development

Often, a parallel application is developed by integrating several distinct functional modules, that is each developed separately. Each module is a parallel program that runs on a dedicated set of processes, and the computation consists of phases where modules compute separately, intermixed with global phases where all processes communicate. It is convenient to allow each module to use its own private process numbering scheme, for the intramodule computation. This is achieved by using a private module context for intramodule computation, and a global context for intermodule communication.

Collective communication

MPI supports collective communication within dynamically created groups of processes. Each such group can be represented by a distinct context. This provides a simple mechanism to ensure that communication that pertains to collective communication within one group is not confused with collective communication within another group.

Lightweight gang scheduling

Consider an environment where processes are multithtreaded. Contexts can be used to provide a mechanism whereby all processes are time-shared between several parallel executions, and can context switch from one parallel execution to another, in a loosely synchronous manner. A

thread is allocated on each process to each parallel execution, and a different context is used to identify each parallel execution. Thus, traffic from one execution cannot be confused with traffic from another execution. The blocking and unblocking of threads due to communication events provide a "lazy" context switching mechanism. This can be extended to the case where the parallel executions are spanning distinct process subsets. (MPI does not require multithreaded processes.)

Discussion: A context handle might be implemented as a pointer to a structure that consists of context label (that is carried by messages sent within this context) and a context member table, that translates process ranks within a context to absolute addresses or to routing information. Of course, other implementations are possible, including implementations that do not require each context member to store a full list of the context members.

Contexts can be used only on the process where they were created. Since the context carries information on the group of processes that belong to this context, a process can send a message within a context only to other processes that belong to that context. Thus, each process needs to keep track only of the contexts that where created at that process; the total number of contexts per process is likely to be small.

The only difference I see between this current definition of context, which subsumes the group concept, and a pared down definition, if that I assume here that process numbering is relative to the context, rather then being global, thus requiring a context member table. I argue that this is not much added overhead, and gives much additional needed functionality.

- If a new context is created by copying a previous context, then one does not need a new member table; rather, one needs just a new context label and a new pointer to the same old context member table. This holds true, in particular, for contexts that include all processes.
- A context member table makes sure that a message is sent only to a process that can execute in the context of the message. The alternative mechanism, which is checking at reception, is less efficient, and requires that each context label be system-wide unique. This requires that, to the least, all processes in a context execute a collective agreement algorithm at the creation of this context.
- The use of relative addressing within each context is needed to support true modular development of subcomputations that execute on a subset of the processes. There is also a big advantage in using the same context construct for collective communications as well.

2.4.1. Context Operations

A global context **ALL** is predefined. All processes belong to this context when computation starts. MPI does not specify how processes are initially ranked within the context ALL. It is expected that the start-up procedure used to initiate an MPI program (at load-time or run-time) will provide information or control on this initial ranking (e.g., by specifying that processes are ranked according to their pid's, or according to the physical addresses of the executing processors, or according to a numbering scheme specified at load time).

Discussion: If we think of adding new processes at run-time, then `ALL` conveys the wrong impression, since it is just the initial set of processes.

The following operations are available for creating new contexts.

MPI_COPY_CONTEXT(newcontext, context)

Create a new context that includes all processes in the old context. The rank of the processes in the previous context is preserved. The call must be executed by all processes in the old context. It is a blocking call: No call returns until all processes have called the function. The parameters are

OUT newcontext handle to newly created context. The handle should not be associated with an object before the call.

IN context handle to old context

Discussion: I considered adding a string parameter, to provide a unique identifier to the next context. But, in an environment where processes are single threaded, this is not much help: Either all processes agree on the order they create new contexts, or the application deadlocks. A key may help in an environment where processes are multithreaded, to distinguish call from distinct threads of the same process; but it might be simpler to use a mutex algorithm at each process.

Implementation note: No communication is needed to create a new context, beyond a barrier synchronization; all processes can agree to use the same naming scheme for successive copies of the same context. Also, no new rank table is needed, just a new context label and a new pointer to the same old table.

MPI_NEW_CONTEXT(newcontext, context, key, index)

OUT newcontext handle to newly created context at calling process. This handle should not be associated with an object before the call.

IN context handle to old context

IN key integer

IN index integer

A new context is created for each distinct value of `key`; this context is shared by all processes that made the call with this key value. Within each new context the processes are ranked according to the order of the `index` values they provided; in case of ties, processes are ranked according to their rank in the old context.

This call is blocking: No call returns until all processes in the old context executed the call.

Particular uses of this function are:

(i) Reordering processes: All processes provide the same `key` value, and provide their index in the new order.

(ii) Splitting a context into subcontexts, while preserving the old relative order among processes: All processes provide the same `index` value, and provide a key identifying their new subcontext.

MPI_RANK(rank, context)

OUT rank integer

IN context context handle

Return the rank of the calling process within the specified context.

MPI_SIZE(size, context)

OUT size integer

IN context context handle

Return the number of processes that belong to the specified context.

Usage note

Use of contexts for libraries: Each library may provide an initialization routine that is to be called by all processes, and that generate a context for the use of that library.

Use of contexts for functional decomposition: A harness program, running in the context `ALL` generates a subcontext for each module and then starts the submodule within the corresponding context.

Use of contexts for collective communication: A context is created for each group of processes where collective communication is to occur.

Use of contexts for context-switching among several parallel executions: A preamble code is used to generate a different context for each execution; this preamble code needs to use a mutual exclusion protocol to make sure each thread claims the right context.

Discussion: If process handles are made explicit in MPI, then an additional function needed is **MPI_PROCESS(process, context, rank)**, which returns a handle to the process identified by the `rank` and `context` parameters.

A possible addition is a function of the form **MPI_CREATE_CONTEXT(newcontext, list_of_process_handles)** which creates a new context out of an explicit list of members (and rank them in their order of occurrence in the list). This, coupled with a mechanism for requiring the spawning of new processes to the computation, will allow to create a new all inclusive context that includes the additional processes. However, I oppose the idea of requiring dynamic process creation as part of MPI. Many implementers want to run MPI in an environment where processes are statically allocated at load-time.

2.4.2. Error Handling

It is assumed that MPI is implemented on top of an error-free communication subsystem: A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, and the likes. In other words, MPI does not provide mechanisms to deal with failures in the underlying communication subsystem – it is the responsibility of the MPI implementer to insulate the user from such errors (or to reflect them as global program failures). The same holds true for 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 profile provides several mechanisms to alleviate the portability problem this represents. One can also write **safe** programs, that are not subject to resource errors.

All MPI procedure calls return an error parameter that indicates successful completion of the operation, or the error condition that occurred, otherwise.

The recommended implementation profile in a POSIX environment is for any MPI routine that encounters a recoverable error to store an error number in a global variable (*errno* in a C environment) and generate an *MPI error signal*, using a special signal value. The default handler for this signal terminates the execution of all involved processes, with a suitable error

message being returned to the user. However, the user can provide his or her own signal handling routine. In particular, the user can specify a "noop" signal handler, thus relegating all error handling to the user code, using the error parameters returned by the MPI calls.

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). In such cases, an error will be undetected, if the user disabled the MPI error signal.

Discussion: The alternative choice is to have fatal and non-fatal signals.

2.5. Messages

A message consists of an *envelope* and *data*.

2.5.1. Data

The data part of a message consists of a sequence of values, each of a basic datatype in the host language. Thus, in Fortran, a message consists of a sequence of values that are each of type `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `LOGICAL`, or (length 1) `CHARACTER`. A message may mix values of different types.

Discussion: May also need `DOUBLE COMPLEX` in Fortran.

2.5.2. Envelope

The following information is associated with each message:

source The rank the sending process

destination The rank of the receiving process

tag User defined

context handle

The range of valid values for the **source** and **destination** fields is `0 ... n-1`, where `n` is the number of processes in the current context. The ranges of valid values for `tag` is implementation dependent, and can be found by calling a suitable query function, as described in Section 7. `Context` should be a context shared by both source and destination.

The `tag` field can be arbitrarily set by the application, and can be used to distinguish different messages.

The actual mechanism used to associate an envelope with a message is implementation dependent; some of the information (e.g., **sender** or **receiver**) may be implicit, and need not be explicitly carried by a message.

2.6. Data Buffers

The basic point to point communication operations are **send** and **receive**. A **send** operation creates a message; the message data is assembled from the **send buffer**. A **receive** operation consumes a message; the message data is moved into the **receive buffer**. The specification of send or receive buffers uses the same syntax.

A buffer consists of a sequence **buffer components**. Each component consists of a sequence variables of the same basic type. There are three basic types of buffer components:

block A sequence of contiguous values of the same basic type, specified by

start Initial element

len Number of elements (`len` $\geq$ `0`)

datatype Type of elements

vector A sequence of equally spaced and equally sized blocks of elements of the same basic type, specified by

start Initial element

len Number of elements (`len` $\geq$ `0`)

stride Number of elements between the start of each block

lenblk Number of elements in each block (`lenblk` $\leq$ `stride`)

datatype Type of elements

Note that a constant stride becomes contiguous when `stride = lenblk`. A vector buffer component can be an arbitrary submatrix of a two-dimensional matrix.

indexed A sequence of elements of the same basic type, specified by

start initial element

list_of_indices List of displacements of the elements in the buffer components, relative to the initial element.

datatype Type of elements

For example, if a Fortran array is declared as `double precision a(10)`, then the tuple `<(a(3), (0,3,6,2), DOUBLE>` specifies a buffer component with entries `a(3) a(6), a(9), a(5)`.

Discussion: Do we allow entries to be repeated in an indexed buffer component?
Do we allow different buffer components to overlap?
Do we require in an vector buffer component that `len` be a multiple of `lenblk`?

A buffer is described by an opaque object accessed via a **buffer handle**. Such object is created and destroyed via calls to `MPI_CREATE` and `MPI_FREE`. It is associated with successive buffer components by calling in succession one of the functions `MPI_ADD_BLOCK`, `MPI_ADD_VECTOR` or `MPI_ADD_INDEX`, in order to append a component to the buffer associated with a buffer handle. A buffer object can be destroyed only if there is no pending communication operation using it. After a buffer object is destroyed the associated buffer handle is undefined.

MPI_ADD_BLOCK(buffer, start, len, datatype)

Append a block component to buffer. The parameters are:

INOUT buffer buffer handle

IN start buffer component initial element (choice)

IN len Number of elements (integer)

IN datatype datatype identifier (status)

MPI_ADD_VEC(buffer, len, stride, lenblk, datatype)

Append a vector buffer component to buffer. The parameters are:

INOUT buffer buffer handle

IN start buffer component initial element (choice)

IN len Number of elements (integer)

IN stride Number of elements between the start of each block (integer)

IN lenblk Number of elements in each block (integer)

IN datatype datatype identifier (status)

MPI_ADD_INDEX(buffer, start, list_of_indices, datatype)

Append an indexed buffer component to buffer. The parameters are:

INOUT buffer buffer handle

start initial position for indexing (choice)

list_of_indices list of relative indices of entries (array of integers)

IN datatype datatype identifier (status)

Consider, for example, the following fragment of Fortran code

```
DOUBLE PRECISION A(10,20)
INTEGER B, C(5,10)
INTEGER BH
...
CALL MPI_CREATE(BH, MPI_BUFFER, MPI_PERSISTENT)
CALL MPI_ADD_BLOCK (BH, B, 1, MPI_INT)
CALL MPI_ADD_VEC (BH, A(1,3), 11, 4, 2, MPI_DOUBLE)
CALL MPI_ADD_INDEX(BH, C(3,7), (4,2,1), MPI_INT)
```

Then the buffer associated with the handle `BH` consists of the sequence of variables
`B, A(1,3), A(2,3), A(5,3), A(6,3), A(9,3), A(10,3), A(3,4), A(4,4), A(7,4), A(8,4), A(1,5), C(2,8), C(5,7), C(4,7)`.

A message created from this buffer will consist of a sequence of one integer, followed by eleven double precision reals, followed by three integers.

A buffer handle can be used for communication, even if it is not associated with any variables (i.e., even if it was not set by any `MPI_ADD_xxx` call). Such handle is associated with an empty buffer, and a message created from it contains no data.

Discussion: The main modifications w.r.t. the proposal of Gropp and Lusk is measuring length in elements, rather than bytes. Seems more natural, since type is known. Also, object creation uses a generic function, rather than a function specific to buffer descriptors. As result, I gave up on the size parameter. This may not be such a loss: it is not clear that specifying a maximum length at buffer object creation is useful, since indices of arbitrary size may need to be stored in the object.

2.6.1. Data Conversion

The types and the locations of the entries used to create a message is solely determined from the information in the buffer descriptor, using the storage association rules specified by the host language and its implementation; the type and the locations of these entries do not depend on the declaration for the corresponding variables in the calling program. It is not required that the data types specified in a buffer descriptor match the data types of the corresponding elements in the host program. However, in case of mismatches, the correspondence between entries in the host program and entries in a message created with the buffer descriptor may be implementation dependent. No data conversion occurs when data is moved from a sender buffer into a message.

Consider the following fragment of Fortran code

```
REAL A(100)
INTEGER BH
...
CALL MPI_CREATE(BH, MPI_BUFFER, MPI_PERSISTENT)
CALL MPI_ADD_BLOCK (BH, A(1), 1, MPI_REAL)
CALL MPI_ADD_BLOCK (BH, A(2), 1, MPI_INT)
CALL MPI_ADD_BLOCK (BH, A(3), 1, MPI_LOGICAL)
CALL MPI_ADD_BLOCK (BH, A(4), 1, MPI_COMPLEX)
CALL MPI_ADD_BLOCK (BH, A(6), 1, MPI_DOUBLE)
CALL MPI_ADD_BLOCK(BH, A(8), 4, MPI_CHAR)
```

A message created from this buffer will consist of a sequence containing one real, one integer, one logical, one complex, one double, and four characters. No data conversion occurs when values are copied from the sender buffer to the message. Thus, the first entry in the message is a real number equal to `A(1)`; the second entry is an integer number that happens to have the same binary representation as `A(2)`; the third entry is a logical value that happens to have the same binary representation as `A(3)`; the fourth entry in a complex number with a binary representation identical to `A(4), A(5)`; the fifth entry is a double precision value with a binary representation identical to `A(6), A(7)`; and if words have four bytes then the last four entries are bytes that make the binary representation of `A(8)`, in the byte order of the executing machine.

The correspondence between the first seven entries of the array `A` and the first five entries of the message created from this buffer is determined by the rules of Fortran 77 on storage association: Each variable of type `INTEGER`, `REAL`, or `LOGICAL` occupy one *numeric storage unit*; a variable of type `DOUBLE` or `COMPLEX` occupy two numeric storage units. Thus, the same correspondence will hold for any implementation. However, different implementations may have different binary encodings of integer, real and logical values, so that the actual values transferred by the message may differ.

The correspondence between the entry `A(8)` of the array, and the last four character entries in the message is implementation dependent, since the Fortran language does not specify a correspondence between character storage units and numeric storage units (an array of characters cannot be equivalenced with an array of integers). Different results may occur in big-endians or small-endians machines, or in 32 bit or 64 bit machines.

The same holds, symmetrically, when a message is received. Entries are moved from the message into the receiver memory according to the information provided by the buffer descriptor, with no regard to the way the corresponding variables are declared in the receiving program.

When data is moved in a homogeneous environment between nodes having the same architecture, then no data conversion occur at any point during data transfer. Assume, in the previous example, that an identically declared buffer descriptor is used to receive data into an identically declared array at the receiving process. Then, when these two nodes communicate, the values in the first eight entries of array `A` of the sender will be copied into the first eight entries of array `A` at the receiver (assuming that reals occupy the same storage as four bytes). In particular, in a homogeneous environment, it is possible to communicate using only character typed messages.

When data is moved into a heterogeneous environment between nodes having distinct architectures, data conversion may occur during the transfer: Each entry is converted from the data representation used on the sending node to the data representation used in the receiving node.

Consider the following fragment of Fortran code.

```
REAL X, Y
CHARACTER (4) Z
INTEGER BH
...
CALL MPI_CREATE(BH, MPI_BUFFER, MPI_PERSISTENT)
CALL MPI_ADD_BLOCK (BH, X, 1, MPI_REAL)
CALL MPI_ADD_BLOCK (BH, Y, 4, MPI_CHAR)
CALL MPI_ADD_BLOCK (BH, Z, 4, MPI_CHAR)
```

Assume that the same arrays `X, Y, Z` and buffer handle `BH` are created at two processes A and B; this handle is used by A to create and send a message for B, by B to receive this message. Further assume that both A and B run on distinct nodes, with possibly different 32 bit architectures. Then `X` at process B is assigned the value of `X` at process A (up to rounding errors that may occur during conversion); `Z` on process B is assigned the character string value of `Z` on process A; if both nodes use ASCII encoding, then no conversion is required for the characters. On the other hand, variable `Y` on process B may be allocated a value that differs from the value of variable `Y` on process A. This may occur if the two nodes use a different byte sequence (little-endian vs big-endian), or use a different binary representation for reals (IEEE vs HEX). Thus, in order to ensure correct execution in a heterogeneous environment, it is important that the types of values in a message match the types of the corresponding values in the sending and in the receiving program.

2.7. Receive Criteria

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. In addition, the value for the `dest` field is set, implicitly, to be equal to the receiving process id. The receiver may specify a **DONTCARE** value for `source`, or `tag`, indicating that any source and/or tag are acceptable. It cannot specify a DONTCARE value for `context` or `dest`. Thus, a message can be received by a receive operation only if it is addressed to the receiving task, has a matching context, has matching source unless source=DONTCARE in the pattern, and has a matching tag unless tag=DONTCARE in the pattern.

The length of the received message must be less or equal the length of the receive buffer. I.e., all incoming data must fit, without truncation, into the receive buffer. It is erroneous to

receive a message which length exceed the receive buffer, and the outcome of program where this occurs is undetermined.

2.8. Communication Mode

A sending operation can occur in one of two modes:

REGULAR The send may start whether or not a matching receive has been posted.

READY The send may start only if a matching receive has been posted.

A **ready send** can start only if a matching receive is already posted; otherwise the operation is erroneous and its outcome is undefined. In some systems, this will allow to optimize communication and avoid a hand-shaking operation that is otherwise required.

Discussion: I deleted the symmetric ready receive. Will revive it if there is a requirement for it.

2.9. Communication Objects

An opaque communication object identifies various properties of a communication operation, such as the buffer descriptor 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 last communication operation that was performed with this object. This object is accessed using a communication handle.

One can consider communication operations to consist of the following suboperations:

INIT(operation, params, handle) Process provides all relevant parameters for its participation in the communication operation (type of operation, data buffer, tag, participants, etc.). An object is created that identifies the operation.

START(handle) The communication operation is started

COMPLETE(handle) The communication operation is completed.

FREE(handle) The communication object, and associated resources are freed.

Correct invocation of these suboperations is a sequence of the form

INIT (**START COMPLETE**)* **FREE**.

I.e., an object needs be created before communication occurs; it can be reused only after the previous use has completed; and it needs to be freed eventually (of course, one can assume that all objects are freed at program termination, by default).

The above scenario pertains to *persistent* objects. One can also create *ephemeral* objects. Such object persists only until the communication operation is completed, at which point it is destroyed. Thus, correct invocation of suboperations with an ephemeral object is **INIT START COMPLETE**.

A user may directly invokes these suboperations. This would allow to amortize the overhead of setting up a communication over many successive uses of the same handle, and allows to overlap communication and computation. Simpler communication operations combine several of these suboperations into one operation, thus simplifying the use of communication primitives. Thus, one only needs to specify precisely the semantics of these suboperations in order to specify the semantics of MPI point to point communication operations.

We say that a communication operation (send or receive) is **posted** once a **start** suboperation was invoked; the operation is **completed** once the **complete** suboperation completes. A send and a receive operation **match** if the receive pattern specified by the receive matches the message envelope created by the send.

2.9.1. Communication Object Creation

An object for a send operation is created by a call to **MPI_INIT_SEND**. A call to **MPI_INIT_RECV** is similarly used for creating an object for a receive operation. The creation of a communication object is a local operation that need not involve communication with a remote process.

MPI_INIT_SEND (handle, buffer_handle, dest, tag, context, mode, persistence)

Creates a send communication object. Parameters are

OUT handle message handle. The handle should not be associated with any object before the call.

IN buffer_handle handle to send buffer descriptor

IN dest rank in context of destination (integer)

IN tag user tag for messages sent with this handle (integer)

IN context handle to context of messages sent with this handle

IN mode send mode (state type, with two values: `MPI_REGULAR` and `MPI_READY`)

IN persistence handle persistence (state type, with two values: `MPI_PERSISTENT` and `MPI_EPHEMERAL`)

MPI_INIT_RECV (handle, buffer_handle, source, tag, context, persistence)

Create a receive handle. Parameters are

OUT handle message handle. The handle should not be associated with any object before the call.

IN buffer_handle handle to receive buffer descriptor.

IN source rank in context of source, or DONTCARE (integer).

IN tag user tag for messages received with this handle, or DONTCARE (integer).

IN context handle to context of messages received with this handle.

IN persistence handle persistence (state type, with two values: `MPI_PERSISTENT` and `MPI_EPHEMERAL`)

See Section 2.5.2 for a discussion of source, tag and context.

Discussion: I have not included proposals for partially specified message handles, that some peoples seem to desire.

I have merged all handle setup into one call.

2.9.2. Communication Start

MPI_START(handle)

IN handle communication handle

The `MPI_START` function starts the execution of a communication operation (send or receive). A sender should not update the send buffer after a send operation has started and until it is completed. A receiver should not access the receive buffer after a receive operation was started and until it is completed. A program that does not satisfy this condition is erroneous and its outcome is undetermined.

2.9.3. Communication Completion

MPI_WAIT (handle, return_status_handle)

IN handle communication handle

OUT return_handle handle that is associated with return status object.

A call to **MPI_WAIT** returns when the send operation identified by `handle` is complete. 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. However, it does not indicate that the message has been received; rather it may have been buffered by the communication subsystem.

The completion of a receive operation indicates that the receiver is now free to access the locations in the receive buffer, which contain the received message, or any other location that can be referenced by the receive operation. It does not indicate that the matching send operation has completed.

The call returns a handle to an opaque object that contains information on the completed operation – the **return status** object.

MPI_STATUS (handle, flag, return_handle)

IN handle communication handle

OUT flag logical

OUT return_handle handle that is associated with return status object.

A call to **MPI_STATUS** returns `flag=true` if the operation identified by `handle` is complete, In such case, the return handle points to an opaque object that contains information on the completed information. It returns `flag=false`, otherwise. In such case, the value of the return handle is undefined.

Implementation notes:

A call to `MPI_WAIT` blocks only the executing thread. If the executing process is multi-threaded, then other threads within the process can be scheduled for execution.

The use of a blocking receive operation (`MPI_WAIT`) allows the operating system to deschedule the blocked thread and schedule another thread for execution, if such is available. The use of a nonblocking receive operation (`MPI_STATUS`) allows the user to schedule alternative activities within a single thread of execution.

The intended implementation of `MPI_STATUS` is for that operation to return as soon as possible. However, if repeatedly called for an operation that is enabled, it must eventually succeed.

The return status object for a send operation carries no information. The return status object for a receive operation carries information on the source, tag and length of the received message. These fields are required because the receive operation may have specified `DONTCARE` in either source or tag field, and the message may have been shorter than the receive buffer.

MPI_RETURN_STAT(handle, len, source, tag)

IN handle handle to return status object

OUT len difference between length of receive buffer and length of received message, in bytes. Thus, the value returned is zero if the received message matches the the receive buffer, positive if it is shorter (integer).

OUT source rank of message sender in message context (integer).

OUT tag tag of received message (integer).

Discussion: I put the difference between message buffer and message length as the value returned, rather than length of received message, so that it might be easy to test for exact match.

The use of a return status object, rather than a list of parameters may simplify the use of MPI routines, if the values stored in the object are seldom checked. A predefined return status object should be provided, to ease programming.

2.9.4. 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 specific message. A call to `MPI_WAITANY` or `MPI_STATUSANY` can be used to wait for the completion of one out of several operations; a call to `MPI_WAITALL` can be used to wait for all pending operations in a list.

MPI_WAITANY (list_of_handles, index, return_handle)

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 return_handle. The parameters are:

IN list_of_handles list of handles to communication objects.

OUT index index of handle for operation that completed (integer).

OUT return_handle handle that is associated with return status object. Set to return status of operation that completed.

The successful execution of `MPI_WAITANY(list_of_handles, index, return_handle)` is equivalent to the successful execution of `MPI_WAIT(handle[i], return_handle)`, where `i` is the value returned by `index` and `handle[i]` is the `i`-th handle in the list.

If more then one operation is enabled and can terminate, one is arbitrarily chosen (subject to the restrictions on operation termination order, see Section 2.13).

`MPI_WAITANY ( list_of_handles, index, return_status_handle)` is

```
{MPI_WAIT_RECV (handle[1], return_handle); index = 0} || ...
||
{MPI_WAIT_RECV (handle[n], return_handle); index = n-1}
```

("||" indicates choice; one of the alternatives is chosen, nondeterministically.)

MPI_STATUSANY (list_of_handles, index, return_handle)

Causes either one or none of the operations associated with the communication handles to return. In the former case, it has the same return semantics as a call to `MPI_WAIT_ANY`. In the later case, it returns a value of -1 in `index` and `return_handle` is undefined. The parameters are:

IN list_of_handles list of handles to communication objects.

OUT index index of handle for operation that completed, or -1 if none completed (integer).

OUT return_handle handle that is associated with return status object. Set to return status of operation that completed, if any; undefined when `index = -1`.

MPI_WAITALL(list_of_handles, list_of_return_handles)

Blocks until all communication operations associated with handles in the list complete, and return the status of all these operations. The parameters are:

IN list_of_handles list of handles to communication objects.

OUT list_of_return_handles Must have the same length as the first list. Each return status object is set to the return status of the corresponding operation in the first list.

2.10. Blocking Communication

Blocking send and receive operations combine all communication suboperations into one call. The operation returns only when the communication completes and no communication object persists after the call completed. However, the buffer descriptor object needs be created ahead of the call.

We use the following naming convention for such operations:

$$\begin{bmatrix} - \\ \mathbf{R} \end{bmatrix} \begin{bmatrix} \mathbf{SEND} \\ \mathbf{RECV} \end{bmatrix}$$

The first letter (void or **R**) indicates the start mode (regular or ready).

MPI_SEND (buffer_handle, dest, tag, context) is

```
MPI_INIT_SEND(handle, buffer_handle, dest, tag, context, REGULAR, EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle, null)
```

MPI_RSEND (buffer_handle, dest, tag, context) is

```
MPI_INIT_SEND(handle, buffer_handle, dest, tag, context, READY, EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle, null)
```

MPI_RECV(buffer_handle, source, tag, context, return_handle) is

```
MPI_INIT_RECV(handle, buffer_handle, source, tag, context, EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle,return_handle)
```

Implementation note:
While these functions can be implemented via calls to functions that implement suboperations, as described in this subsection, an efficient implementation may optimize away these multiple calls, provided it does not change the behavior of correct programs.

2.11. Nonblocking Communication

Nonblocking send and receive operations combine the first two suboperations (`INIT` and `START`) into one call. They use ephemeral communication objects, so that the operation is completed, and the associated resources are freed, by using one of the functions `MPI_WAIT, MPI_STATUS, MPI_WAITANY, MPI_STATUSANY`, or `MPI_WAITALL`. Here, too, a buffer object has to be created ahead of the communication initiation operation.

We use the same naming convention as for blocking operations: a prefix of **R** indicates the `READY` mode. In addition, a prefix of **I** is used to indicate *immediate* (i.e., nonblocking) execution.

MPI_ISEND (handle, buffer_handle, dest, tag, context) is

```
MPI_INIT_SEND(handle, buffer_handle, dest, tag, context, REGULAR, EPHEMERAL)
MPI_START(handle)
```

MPI_IRSEND (handle, buffer_handle, dest, tag, context) is

```
MPI_INIT_SEND(handle, buffer_handle, dest, tag, context, READY, EPHEMERAL)
MPI_START(handle)
```

MPI_IRECV(handle, buffer_handle, source, tag, context, return_status_handle) is

```
MPI_INIT_RECV(handle, buffer_handle, source, tag, context, EPHEMERAL)
MPI_START(handle)
```

2.12. Block Sending Operations

The most frequent type of buffer used is a contiguous buffer of numeric storage units, i.e., a contiguous buffer of words that may contain either INTEGER, REAL or LOGICAL values (in FORTRAN). In a homogeneous environment such messages can be used to send arbitrary sequences of contiguous items, where each item occupies an integer number of words.

We specialize the functions in the two previous subsections to this case, thus avoiding the need for the creation of a buffer descriptor object. We use the same naming scheme used in the previous subsections, and append a **B** in the function name, for `BLOCK`.

MPI_SENDB (start, len, dest, tag, context) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, MPI_REAL)
MPI_SEND (buffer_handle, len, dest, tag, context)
```

MPI_RSENDB (buffer_handle, dest, tag, context) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, MPI_REAL)
MPI_RSEND (buffer_handle, len, dest, tag, context)
```

MPI_RECVB(buffer_handle, source, tag, context, return_status_handle) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, MPI_REAL)
MPI_RECV(buffer_handle, source, tag, context, return_status_handle)
```

MPI_ISENDB (handle, buffer_handle, dest, tag, context) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, MPI_REAL)
MPI_ISEND(handle, buffer_handle, dest, tag, context)
```

MPI_IRSENDB (handle, buffer_handle, dest, tag, context) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, MPI_REAL)
MPI_IRSEND(handle, buffer_handle, dest, tag, context)
```

MPI_IRECVB(handle, buffer_handle, source, tag, context, return_status_handle) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, MPI_REAL)
MPI_IRECV(handle, buffer_handle, source, tag, context)
```

Discussion: I use word count, rather than byte count. I believe word messages are much more prevalent than byte messages, and it's a blessing not to have to multiply by 4 for each message. Byte

messages are still available the hard way. Also, word messages fit well with the Fortran numeric storage unit concept.

If we are to add more functions, my next addition would be a vector send operation.

2.13. Correctness

Discussion: The material in this section has not yet been discussed by MPIF. Some or all of it is likely to move to Section 6. It is incorporated here for completeness.

2.13.1. Order

MPI preserves the order of messages between any fixed pair of processes. In other words, if process A executes two successive send `start` suboperations, process B executes two successive receive `start` operations, and both receives match either sends, then the first receive will receive the message sent by the first send, and the second receive will receive the message sent by the second send. Thus, if a two messages from the same source can satisfy a pending receive, the first message sent is accepted; if a message can satisfy two pending receives, the first receive posted is satisfied.

The last paragraph assumes that the send `start` operations are ordered by the program order at process A, and the receive `start` operations are ordered by the program order at process B. If a process is multithreaded and the operations are executed by distinct threads, then the semantics of the threaded system may not define an order between the two operations, in which case the condition is void.

2.13.2. Progress and Fairness

We can model the execution of MPI programs as an interaction between executing processes that execute each their own program, and the **communication subsystem**. The communication subsystem may have various constraints on the amount of resources it can use. E.g.:

Bounds on the number and total sizes of active communication objects. Such bound can be global, per node, or per pair of communicating nodes.

Bounds on the number and total sizes of messages buffered in the system. Such bound can, again, be global, per node, or per pair of communicating node. In addition, a message may be buffered at the sender, at the receiver, at both, or perhaps at another place altogether.

Thus, it will be difficult to set rules on resource management of the communication subsystem. However, it is generally expected that implementers will provide information on the mechanism used for resource allocation, and that query and set functions will allow to query and possibly control the amount of available resources.

We provide in this section a set of minimal requirements on the communication subsystem. Programs that execute on any subsystem that fulfils these minimal requirements are **safe** and will port to any MPI implementation. **Unsafe** programs may execute on some MPI implementations, depending on the amount of available resources and the implementation used for the MPI communication subsystem. Finally **erroneous** programs never execute correctly. (While it is desirable to detect erroneous programs, it is not possible to do so at compile time, and often prohibitive to do so a run time. Thus, the document does not specify a behavior for erroneous programs, although the desired behavior is to return a useful error message.)

1. If a process executes an `INIT` operation, then the operation eventually succeeds, or a

resource exception occurs. The standard does not specify when a resource exception is allowed to occur. It is expected that an operational definition will be made available, in the form of test programs that have to execute with no resource exceptions. It is highly desirable to have generous bounds on the number of concurrently active communication objects each process may have, so that, in practice, `INIT` operations will always be guaranteed to succeed.

2. Each process can initiate a communication operation for each active communication object. I.e. correct `START` operations always succeed (eventually).

3. A send operation is **enabled** if the sending process has issued a `COMPLETE` operation and the receiving process has issued a `START` operation for a matching receive. Symmetrically, a receive operation is **enabled** if the receiving process has issued a `COMPLETE` operation and the sending process has issued a `START` operation for a matching send. An enabled operation may become **disabled** either because it completes successfully or, in the case of a receive, because the matching message is successfully received by another receive operation.

   An enabled operation either completes successfully or becomes permanently disabled.

4. A `FREE` operation always succeeds (eventually).

The four conditions guarantee progress in the communication subsystem. The third condition guarantee (weak) fairness among competing communication requests.

Examples (involving two processors with ranks 0 and 1)

The following program is safe, and should always succeed.

```
CALL MPI_RANK(rank, context)
IF (rank.EQ.0) THEN
   CALL MPI_SENDB(sendbuf, len, 1, tag, context)
   CALL MPI_RECVB(recvbuf, len, 1, tag, context)
ELSE    ! rank.EQ.1
   CALL MPI_RECVB(recvbuf, len, 0, tag, context)
   CALL MPI_SENDB(sendbuf, len, 0, tag, context)
END IF
```

The following program is erroneous, and should always fail.

```
CALL MPI_RANK(rank, context)
IF (rank.EQ.0) THEN
   CALL MPI_RECVB(recvbuf, len, 1, tag, context)
   CALL MPI_SENDB(sendbuf, len, 1, tag, context)
ELSE    ! rank.EQ.1
   CALL MPI_RECVB(recvbuf, len, 0, tag, context)
   CALL MPI_SENDB(sendbuf, len, 0, tag, context)
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_RANK(rank, context)
IF (rank.EQ.0) THEN
   CALL MPI_SENDB(sendbuf, len, 1, tag, context)
   CALL MPI_RECVB(recvbuf, len, 1, tag, context)
ELSE    ! rank.EQ.1
   CALL MPI_SENDB(sendbuf, len, 0, tag, context)
   CALL MPI_RECVB(recvbuf, len, 0, tag, context)
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 out of either processes' address space. Thus, this program can succeed only if the communication system has sufficient buffer space to buffer `len` words of data.

If additional requirements will become part of the standard (e.g., bounds on the minimal number of concurrently active handles that need be supported, then further programs become safe.

3. Collective Communication

Al Geist
Marc Snir

3.1. Introduction

This section is a draft of the current proposal for collective communication. Collective communication is defined to be communication that involves a group of processes. Examples are broadcast and global sum. A collective operation is executed by having all processes in the group call the communication routine, with matching parameters. 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. However, it does not indicate that other processes in the group have started the operation (unless otherwise indicated in the description of the operation). However, the successful completion of a collective communication call may depend on the execution of a matching call at all processes in the group.

The syntax and semantics of the collective operations is defined so as to be consistent with the syntax and semantics of the point to point operations.

The reader is referred to the point-to-point communication section of the current MPI draft for information concerning groups (aka contexts) and group formation operations, and for general information on types of objects used by the MPI library.

The collective communication routines are built above the point-to-point routines. 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. We are using naive implementations of the collective calls in terms of point to point operations in order to provide an operational definition of their semantics.

The following communication functions are proposed.

- 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.
- Simultaneous shift of data around the group, the simplest example being all members sending their data to (rank+1) with wrap around.
- Scan across all members of a group (also called parallel prefix).
- Broadcast from all members to all members of a group.
- Scatter data from all members to all members of a group (also called complete exchange or index).

To simplify the collective communication interface it is designed with two layers. The low level routines have all the generality of, and make use of, the buffer descriptor routines of the point-to-point section which allows arbitrarily complex messages to be constructed. The second level routines are similar to the upper level point-to-point routines in that they send only a contiguous buffer.

Missing:
The current draft does not include the nonblocking collective communication calls that where discussed at the last meeting.

3.2. Group Functions

The point to point document discusses the use of groups (aka contexts), and describe the operations available for the creation and manipulation of groups and group objects. For sake of completeness, we list them anew here.

MPI_CREATE(handle, type, persistence) Create new opaque object

OUT handle handle to object

IN type state value that identifies the type of object to be created

IN persistence state value; either `MPI_PERSISTENT` or `MPI_EPHEMERAL`.

MPI_FREE(handle) Destroy object associated with handle.

IN handle handle to object

MPI_ASSOCIATED(handle, type) Returns the type of the object the handle is currently associated with, if such exists. Returns the special type `MPI_NULL` if the handle is not currently associated with any object.

IN handle handle to object

OUT type state

MPI_COPY_CONTEXT(newcontext, context)

Create a new context that includes all processes in the old context. The rank of the processes in the previous context is preserved. The call must be executed by all processes in the old context. It is a blocking call: No call returns until all processes have called the function.

OUT newcontext handle to newly created context. The handle should not be associated with an object before the call.

IN context handle to old context

MPI_NEW_CONTEXT(newcontext, context, key, index) A new context is created

for each distinct value of `key`; this context is shared by all processes that made the call with this key value. Within each new context the processes are ranked according to the order of the `index` values they provided; in case of ties, processes are ranked according to their rank in the old context. This call is blocking: No call returns until all processes in the old context executed the call.

OUT newcontext handle to newly created context at calling process. This handle should not be associated with an object before the call.

IN context handle to old context

IN key integer

IN index integer

MPI_RANK(rank, context) Return the rank of the calling process within the specified

context.

OUT rank integer

IN context context handle

MPI_SIZE(size, context) Return the number of processes that belong to the specified

context.

OUT size integer

IN context context handle

Extensions

Possible extensions for dynamic process spawning (MPI2):

MPI_PROCESS(process, context, rank) Returns a handle to the process identified by

the `rank` and `context` parameters.

OUT process handle to process object

IN context handle to context object

IN rank integer

MPI_CREATE_CONTEXT(newcontext, list_of_process_handles) creates a new con-

text out of an explicit list of members and rank them in their order of occurrence in the list.

OUT newcontext handle to newly created context. Handle should not be associated with an object before the call.

IN list_of_process_handles List of handles to processes to be included in new group.

This, coupled with a mechanism for requiring the spawning of new processes to the computation, will allow to create a new all inclusive context that includes the additional processes.

3.3. Communication Functions

The proposed communication functions are divided into two layers. The lowest level uses the same buffer descriptor objects available in point-to-point to create noncontiguous, multiple data type messages. The second level is similar to the block send/receive point-to-point operations in that it supports only contiguous buffers of arithmetic storage units. For each communication operation, we list these two level of calls together.

3.4. Synchronization

Barrier synchronization

MPI_BARRIER(group, tag)

MPI_BARRIER blocks the calling process until all group members have called it; the call returns at any process only after all group members have entered the call.

IN group group handle

tag communication tag (integer)

`MPI_BARRIER( group, tag )` is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_PERSISTENT);
MPI_SIZE( &size, group);
MPI_RANK( &rank, group);
if (rank==0)
{
   for (i=1; i < size; i++)
      MPI_RECV(buffer_handle, i, tag, group);
   for (i=1; i < size; i++)
      MPI_SEND(buffer_handle, i, tag, group);
}
else
{
   MPI_SEND(buffer_handle, 0, tag, group);
   MPI_RECV(buffer_handle, 0, tag, group);
}
MPI_FREE(buffer_handle);
```

3.5. Data move functions

Circular shift

MPI_CSHIFT(inbuf, outbuf, tag, group, shift)

Process with rank `i` sends the data in its input buffer to process with rank (`i` + `shift`) mod `group_size`, who receives the data in its output buffer. All processes make the call with the same values for `tag, group`, and `shift`. The `shift` value can be positive, zero, or negative.

IN inbuf handle to input buffer descriptor

OUT outbuf handle to output buffer descriptor

IN tag operation tag (integer)

IN group handle to group

IN shift integer

MPI_CSHIFTB(inbuf, outbuf, len, tag, group, shift)

Behaves like `MPI_CSHIFT`, with buffers restricted to be blocks of numeric units. All processes make the call with the same values for `len, tag, group`, and `shift`.

IN inbuf initial location of input buffer

OUT outbuf initial location of output buffer

IN len number of entries in input (and output) buffers

IN tag operation tag (integer)

IN group handle to group

IN shift integer

`MPI_CSHIFT( inbuf, outbuf, tag, group, shift)` is

```
MPI_SIZE( &size, group);
MPI_RANK( &rank, group);
MPI_ISEND( handle, inbuf, mod(rank+shift, size), tag, group);
MPI_RECV( outbuf, mod(rank-shift,size), tag, group)
MPI_WAIT(handle);
```

Discussion: Do we want to support the case `inbuf = outbuf` somehow?

End-off shift

MPI_EOSHIFT(inbuf, outbuf, tag, group, shift)

Process with rank `i`, $\max(0, -\texttt{shift}) \leq \texttt{i} < \min(\texttt{size}, \texttt{size} - \texttt{shift})$, sends the data in its input buffer to process with rank `i+ shift`, who receives the data in its output buffer. The output buffer of processes which do not receive data is left unchanged. All processes make the call with the same values for `tag, group`, and `shift`.

IN inbuf handle to input buffer descriptor

OUT outbuf handle to output buffer descriptor

IN tag operation tag (integer)

IN group handle to group

IN shift integer

MPI_EOSHIFTB(inbuf, outbuf, len, tag, group, shift)

Behaves like `MPI_EOSHIFT`, with buffers restricted to be blocks of numeric units. All processes make the call with the same values for `len, tag, group`, and `shift`.

IN inbuf initial location of input buffer

OUT outbuf initial location of output buffer

IN len number of entries in input (and output) buffers

IN tag operation tag (integer)

IN group handle to group

IN shift integer

Discussion:
Two other possible definitions for end-off shift: (i) zero filling for processes that don't receive messages, or (ii) boundary values explicitly provided as an additional parameter. Any preferences? (Fortran 90 allows to optionally provide boundary values, and does zero filling, if none were provided)

Broadcast

MPI_BCAST(buffer_handle, tag, group, root)

`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 `tag, group, and root`. On return the contents of the buffer of the process with rank `root` is contained in buffer of all group members.

INOUT buffer_handle Handle for buffer where from message is sent or received.

IN tag tag of communication operation (integer)

IN group context of communication (handle)

IN root rank of broadcast root (integer)

MPI_BCASTB(buf, len, tag, group, root)

`MPI_BCASTB` behaves like broadcast, restricted to a block buffer. It is called by all processes with the same arguments for `len, tag, group` and `root`.

INOUT buffer Starting address of buffer (choice type)

IN len Number of words in buffer (integer)

IN tag tag of communication operation (integer)

IN group context of communication (handle)

in root rank of broadcast root (integer)

`MPI_BCAST( buffer_handle, tag, group, root )` is

```
MPI_SIZE( &size, context);
MPI_RANK( &rank, context);
MPI_IRECV(handle, buffer_handle, root, tag, group);
if (rank==root)
   for (i=0; i < size; i++)
      MPI_SEND(buffer_handle, i, tag, group);
MPI_WAIT(handle)
```

Gather

MPI_GATHER(inbuf, outbuf, tag, group, root, len)

Each process (including the root process) sends the content of its input buffer to the root process. The root process concatenates all the incoming messages in the order of the senders' rank and places the results in its output buffer. It is called by all members of group using the same arguments for `tag, group`, and `root`. The input buffer of each process may have different length.

IN inbuf handle to input buffer descriptor

OUT outbuf handle to output buffer descriptor – significant only at root (choice)

IN tag operation tag (integer)

IN group group handle

IN root rank of receiving process (integer)

OUT len difference between output buffer size (in bytes) and number of bytes received.

Discussion:
It would be more elegant (but no more convenient) to have a return status object.

MPI_GATHERB(inbuf, inlen, outbuf, tag, group, root)

`MPI_GATHER` behaves like `MPI_GATHER` restricted to block buffers, and with the additional restriction that all input buffers should have the same length. All processes should provided the same values for `inlen, tag, group`, and `root` .

IN inbuf first variable of input buffer (choice)

IN inlen Number of (word) variables in input buffer (integer)

OUT outbuf first variable of output buffer – significant only at root (choice)

IN tag operation tag (integer)

IN group group handle

IN root rank of receiving process (integer)

`MPI_GATHERB( inbuf, inlen, outbuf, tag, group, root)` is

```
MPI_SIZE( &size, group);
MPI_RANK( &rank, group);
MPI_ISENDB(handle, inbuf, inlen, root, tag, group);
if (rank==root)
   for (i=0; i < size; i++)
   {
      MPI_RECVB(outbuf, inlen, i, tag, group, return_status);
      outbuf += inlen;
   }
MPI_WAIT(handle);
```

Scatter

MPI_SCATTER(list_of_inbufs, outbuf, tag, group, root, len)

The root process sends the content of its `i`-th input buffer to the process with rank `i`; each process (including the root process) stores the incoming message in its output buffer. The difference between the size of the output buffer (in bytes) and the number of bytes received is returned in `len`. The routine is called by all members of the group using the same arguments for `tag, group`, and `root`.

IN list_of_inbufs list of buffer descriptor handles

OUT outbuf buffer descriptor handle

IN tag operation tag (integer)

IN group handle

IN root rank of sending process (integer)

OUT len number of remaining bytes in the output buffer at each process (integer)

`MPI_SCATTER( list_of_inbufs, outbuf, tag, group, root, len)` is

```
MPI_SIZE( &size, group);
MPI_RANK( &rank, group);
MPI_IRECV(handle, outbuf, root, tag, group);
if (rank=root)
   for (i=0; i < size; i++)
      MPI_SEND(inbuf[i], i, tag, group);
MPI_WAIT(handle, return_status);
MPI_RETURN_STATUS(return_status, len, source, tag);
```

MPI_SCATTERB(inbuf, outbuf, len, tag, group, root)

`MPI_SCATTERB` behaves like `MPI_SCATTER` restricted to block buffers, and with the additional restriction that all output buffers have the same length. The input buffer block of the root process is partitioned into `n` consecutive blocks, each consisting of `len` words. The `i`-th block is sent to the `i`-th process in the group and stored in its output buffer. The routine is called by all members of the group using the same arguments for `tag, group, len`, and `root`.

IN inbuf first entry in input buffer – significant only at root (choice).

OUT outbuf first entry in output buffer (choice).

IN len number of entries to be stored in output buffer (integer)

IN group handle

IN root rank of sending process (integer)

`MPI_SCATTERB( inbuf, outbuf, outlen, tag, group, root)` is

```
MPI_SIZE( &size, group);
MPI_RANK( &rank, group);
MPI_IRECVB( handle, outbuf, outlen, root, tag, group);
if (rank=root)
   for (i=0; i < size; i++)
   {
      MPI_SENDB(inbuf, outlen, i, tag, group, return_status);
      inbuf += outlen;
   }
MPI_WAIT(handle);
```

All-to-all scatter

MPI_ALLSCATTER(list_of_inbufs, outbuf, tag, group, len)

Each process in the group sends its `i`-th buffer in its input buffer list to the process with rank `i` (itself included); each process concatenates the incoming messages in its output buffer, in the order of the senders' ranks. The number of bytes left in the output buffer is returned in `len`. The routine is called by all members of the group using the same arguments for `tag` and `group`.

IN list_of_inbufs list of buffer descriptor handles

OUT outbuf buffer descriptor handle

IN tag operation tag (integer)

IN group handle

OUT len number of remaining bytes in the output buffer (integer)

MPI_ALLSCATTERB(inbuf, outbuf, len, tag, group)

`MPI_ALLSCATTERB` behaves like `MPI_ALLSCATTER` restricted to block buffers, and with the additional restriction that all blocks sent from one process to another have the same length.

The input buffer block of each process is partitioned into `n` consecutive blocks, each consisting of `len` words. The `i`-th block is sent to the `it`-th process in the group. Each process concatenates the incoming messages, in the order of the senders' ranks, and store them in its output buffer. The routine is called by all members of the group using the same arguments for `tag, group`, and `len`.

IN inbuf first entry in input buffer (choice). root (integer)

OUT outbuf first entry in output buffer (choice).

IN len number of entries sent from each process to each other (integer).

IN tag operation tag (integer)

IN group handle

`MPI_ALLSCATTERB( inbuf, outbuf, len, tag, group)` is

```
MPI_SIZE( &size, group);
MPI_RANK( &rank, group);
for (i=0; i < rank; i++)
   {
    MPI_IRECVB(recv_handles[i], outbuf, len, tag, group);
    outbuf += len;
   }
for (i=0; i < size; i++)
   {
    MPI_ISENDB(send_handle[i], inbuf, len, i, tag, group);
    inbuf += len;
   }
MPI_WAITALL(send_handle);
MPI_WAITALL(recv_handle);
```

All-to-all broadcast

MPI_ALLCAST(inbuf, outbuf, tag, group, len)

Each process in the group broadcasts its input buffer to all processes (including itself); each process concatenates the incoming messages in its output buffer, in the order of the senders' ranks. The number of bytes left in the output buffer is returned in `len`. The routine is called by all members of the group using the same arguments for `tag` and `group`.

IN inbuf buffer descriptor handle for input buffer

OUT outbuf buffer descriptor handle for output buffer

IN tag operation tag (integer)

IN group handle

OUT len number of remaining untouched bytes in each output buffer (integer)

MPI_ALLCASTB(inbuf, outbuf, len, tag, group)

`MPI_ALLCASTB` behaves like `MPI_ALLCAST` restricted to block buffers, and with the additional restriction that all blocks sent from one process to another have the same length. The routine is called by all members of the group using the same arguments for `tag, group`, and `len`.

IN inbuf first entry in input buffer (choice). root (integer)

OUT outbuf first entry in output buffer (choice).

IN len number of entries sent from each process to each other (including itself).

IN group handle

`MPI_ALLCASTB( inbuf, outbuf, len, tag, group)` is

```
MPI_SIZE( &size, group);
MPI_RANK( &rank, group);
for (i=0; i < rank; i++)
   {
    MPI_IRECVB(recv_handles[i], outbuf, len, tag, group);
    outbuf += len;
   }
for (i=0; i < size; i++)
   {
    MPI_ISENDB(send_handle[i], inbuf, len, i, tag, group);
   }
MPI_WAITALL(send_handle);
MPI_WAITALL(recv_handle);
```

3.6. Global Compute Operations

Reduce

MPI_REDUCE(inbuf, outbuf, tag, group, root, op)

Combines the values provided in the input buffer of each process in the group, using the operation `op`, and returns the combined value in the output buffer of the process with rank `root`. Each process can provide one value, or a sequence of values, in which case the combine operation is executed pointwise on each entry of the sequence. For example, if the operation is `max` and input buffers contains two floating point numbers, then outbuf(1) = global max(inbuf(1)) and outbuf(2) = global max(inbuf(2)). All input buffers should define sequences of equal length of entries of types that match the type of the operands of `op`. The output buffer should define a sequence of the same length of entries of types that match the type of the result of `op`. (Note that, here as for all other communication operations, the type of entries inserted in a message depend on the information provided by the input buffer descriptor, and not on the declarations of these variables in the calling program. The types of the variables in the calling program need not match the types defined by the buffer descriptor, but in such case the outcome of a reduce operation may be implementation dependent.)

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 `tag, group, root` and `op`.

IN inbuf handle to input buffer

OUT outbuf handle to output buffer – significant only at root

IN tag operation tag (integer)

IN group handle to group

IN root rank of root process (integer)

IN op operation (status)

We list below the operations are supported for Fortran, each with the corresponding value of the `op` parameter.

MPI_IMAX integer maximum

MPI_RMAX real maximum

MPI_DMAX double precision real maximum

MPI_IMIN integer minimum

MPI_RMIN real minimum

MPI_DMIN double precision real minimum

MPI_ISUM integer sum

MPI_RSUM real sum

MPI_DSUM double precision real sum

MPI_CSUM complex sum

MPI_DCSUM double precision complex sum

MPI_IPROD integer product

MPI_RPROD real product

MPI_DPROD double precision real product

MPI_CPROD complex product

MPI_DCPROD double precision complex product

MPI_AND logical and

MPI_IAND integer (bit-wise) and

MPI_OR logical or

MPI_IOR integer (bit-wise) or

MPI_XOR logical xor

MPI_IXOR integer (bit-wise) xor

MPI_MAXLOC rank of process with maximum integer value

MPI_MAXRLOC rank of process with maximum real value

MPI_MAXDLOC rank of process with maximum double precision real value

MPI_MINLOC rank of process with minimum integer value

MPI_MINRLOC rank of process with minimum real value

MPI_MINDLOC rank of process with minimum double precision real value

MPI_REDUCEB(inbuf, outbuf, len, tag, group, root, op)

Is same as `MPI_REDUCE`, restricted to a block buffer.

IN inbuf first location in input buffer

OUT outbuf first location in output buffer – significant only at root

IN len number of entries in input and output buffer (integer)

IN tag operation tag (integer)

IN group handle to group

IN root rank of root process (integer)

IN op operation (status)

Discussion:
If we are to be compatible with the point to point block operations, the `len` parameter should indicate the number of words in buffer. But it might be more natural to have `len` indicate the number of entries in the buffer, so that if the entries are complex or double precision, `len` will be half the number of words in the buffer.

MPI_USER_REDUCE(inbuf, outbuf, tag, group, root, function)

Same as the reduce operation function above except that a user supplied function is used. `function` is an associative and commutative function with two arguments. The types of the two arguments and of the returned values all agree.

IN inbuf handle to input buffer

OUT outbuf handle to output buffer – significant only at root

IN tag operation tag (integer)

IN group handle to group

IN root rank of root process (integer)

IN function user provided function

MPI_USER_REDUCEB(inbuf, outbuf, len, tag, group, root, function) Is same

as `MPI_USER_REDUCE`, restricted to a block buffer.

IN inbuf first location in input buffer

OUT outbuf first location in output buffer – significant only at root

IN len number of entries in input and output buffer (integer)

IN tag operation tag (integer)

IN group handle to group

IN root rank of root process (integer)

IN op operation (status)

Discussion:
Do we also want a version of reduce that broadcasts the result to all processes in the group? (This can be achieved by a reduce followed by a broadcast, but a combined function may be somewhat more efficient.

Scan

MPI_SCAN(inbuf, outbuf, tag, group, op)

MPI_SCAN is used to perform a parallel prefix with respect to an associative reduction operation on data distributed across the group. The operation returns in the output buffer of the process with rank `i` the reduction of the values in the input buffers of processes with ranks `0,...,i`. The type of operations supported and their semantic, and the constraints on input and output buffers are as for `MPI_REDUCE`.

IN inbuf handle to input buffer

OUT outbuf handle to output buffer

IN tag operation tag (integer)

IN group handle to group

IN op operation (status)

MPI_SCANB(inbuf, outbuf, len, tag, group, op) Same as `MPI_SCAN`, restricted to block buffers.

IN inbuf first input buffer element (choice)

OUT outbuf first output buffer element (choice)

IN len number of entries in input and output buffer (integer)

IN tag operation tag (integer)

IN group handle to group

IN op operation (status)

MPI_USER_SCAN(inbuf, outbuf, tag, group, function)

Same as the scan operation function above except that a user supplied function is used. `function` is an associative and commutative function with two arguments. The types of the two arguments and of the returned values all agree.

IN inbuf handle to input buffer

OUT outbuf handle to output buffer

IN tag operation tag (integer)

IN group handle to group

IN function user provided function

MPI_USER_SCANB(inbuf, outbuf, len, tag, group, function) Is same as `MPI_USER_SCAN`, restricted to a block buffer.

IN inbuf first location in input buffer

OUT outbuf first location in output buffer

IN len number of entries in input and output buffer (integer)

IN tag operation tag (integer)

IN group handle to group

IN function user provided function

Discussion:

Do we want scan operations executed by segments? (The HPF definition of prefix and suffix operation might be handy – in addition to the scanned vector of values there is a mask that tells where segments start and end.)

Missing:

Nonblocking (immediate) collective operations. The syntax is obvious: for each collective operation `MPI_op(params)` one may have a new nonblocking collective operation of the form `MPI_Iop(handle, params)`, that initiates the execution of the corresponding operation. The execution of the operation is completed by executing `MPI_WAIT(handle,...`, `MPI_STATUS(handle,...)`, `MPI_WAITALL`, `MPI_WAITANY`, or `MPI_STATUSANY`. There are three issues to consider:

(i) The exact definition of the semantics of there operations (in particular constraints on order.

(ii) The complexity of implementation (including the complexity of having the same `WAIT` or `STATUS` functions apply both to point-to-point and to collective operations).

(iii) The accrued performance advantage.

3.7. Correctness

Discussion: This is still very preliminary

The semantics of the collective communication operations can be derived from their operational definition in terms of point-to-point communication. It is assumed that messages pertaining to one operation cannot be confused with messages pertaining to another operation. Also messages pertaining to two distinct occurrences of the same operation cannot be confused, if the two occurrences have distinct parameters. The relevant parameters for this purpose are `group`, `tag`, `root` and `op`. messages pertaining to another occurrence of the same operation, with different parameters. The implementer can, of course, use another, more efficient implementation, as long as it has the same effect.

Discussion:

This statement does not yet apply to the current, incomplete and somewhat careless definitions I provided in this draft.

The definition above means that messages pertaining to a collective communication carry information identifying the operation itself, and the values of the `tag`, `group` and, where relevant, `root` or `op` parameters. Is this acceptable?

A few examples:

```
MPI_BCAST(buf, len, tag, group, 0);
MPI_BCAST(buf, len, tag, group, 1);
```

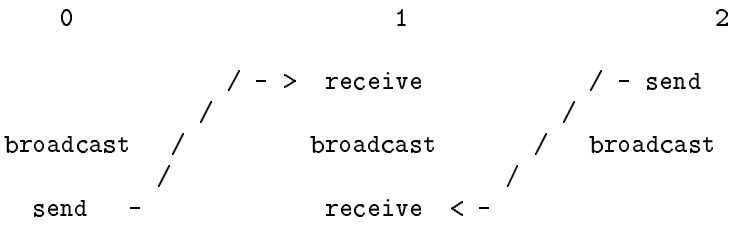
Two consecutive broadcasts, in the same group, with the same tag, but different roots. Since the operations are distinguishable, messages from one broadcast cannot be confused with messages from the other broadcast; the program is safe and will execute as expected.

```
MPI_BCAST(buf, len, tag, group, 0);
MPI_BCAST(buf, len, tag, group, 0);
```

Two consecutive broadcasts, in the same group, with the same tag and root. Since point-to-point communication preserves the order of messages here, too, messages from one broadcast will not be confused with messages from the other broadcast; the program is safe and will execute as intended.

```
MPI_RANK(&rank, group)
if (rank==0)
  {
   MPI_BCASTB(buf, len, tag, group, 0);
   MPI_SENDB(buf, len, 2, tag, group);
  }
elseif (rank==1)
  {
   MPI_RECVB(buf, len, MPI_DONTCARE, tag, group);
   MPI_BCASTB(buf, len, tag, group, 0);
   MPI_RECVB(buf, len, MPI_DONTCARE, tag, group);
  }
else
  {
   MPI_SENDB(buf, len, 2, tag, group);
   MPI_BCASTB(buf, len, tag, group, 0);
  }
```

Process zero executes a broadcast followed by a send to process one; process two executes a send to process one, followed by a broadcast; and process one executes a receive, a broadcast and a receive. A possible outcome is for the operations to be matched as illustrated by the diagram below.



The reason is that broadcast is not a synchronous operation; the call at a process may return before the other processes have entered the broadcast. Thus, the message sent by process zero can arrive to process one before the message sent by process two, and before the call to broadcast on process one.

4. Process Topologies

4.1. Introduction

The MPI1 draft standard for a user–level message–passing interface [7] explicitly states that it does not yet provide a complete interface for message–passing parallel programming. Instead the draft concentrates on the functions which are essential in any reasonable standard. More sophisticated features will have to be added later, some of which are listed in section 5 of the draft standard.

One of the outstanding issues is the handling of logical process topologies (or virtual topologies). Process groups in MPI1 are unstructured collections of processes. This contrasts with the majority of parallel applications, in which the processes are arranged in topological patterns which are often 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 PID numbers 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. 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 [14]. 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. Whereas this is not a problem for machines with fast interconnect networks and slow processors, in general resources can be used more efficiently by keeping the communication local. 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 [6, 5].

4.2. 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 [11, 12] 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 structures of arbitrary dimension are supported. The topology definition function MPI_CART can be used to describe both grids and tori by specifying per coordinate direction whether the process structure is periodic or not. Similar functions are contained in EXPRESS [15] 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 PID numbers of other processes by calling MPI_INQPID. The processes are specified by their logical node numbers (graph) or rectangular process coordinates (grid/torus). Thus, it is possible to inquire for the PID numbers of only those (few) processes with which messages have to be exchanged.

4.3. Embedding in MPI

The support for virtual topologies has to be fully integrated in the framework of MPI. In particular, the process group concept is being defined in close collaboration with the Collective Communication subcommittee. 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 remapping 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 logical topology.

The support for virtual topologies cannot be handled by a library on top of MPI. One reason is the interaction of the topologies with group creation and collective communication. Furthermore, a portable topology definition routine would need detailed information about the available hardware resources (processors, interconnect links, etc.) which for the most general case is difficult to standardize in MPI. Also, the implementation of such a portable library would tend to be inefficient because it could not take advantage of efficient mapping algorithms which depend strongly on the hardware topology.

4.4. Overview of the proposed MPI functions

The topology definition functions MPI_CART and MPI_GRAPH assign a topology to a process group. The group may either be the ALL group, or it may be the result of a copying or

partitioning operation. The default topology of a process group is: *no mapping*. The function MPI_INQMAP is provided for inquiring about the topology of a process group, whereas MPI_INQCART and MPI_INQGR return the size of the process structure and the position of the calling process in a cartesian or graph topology, respectively.

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 coodinate direction, depending on the number of processes in the group and optional constraints which can be specified by the user.

The number of processes in the group must be greater than or equal to the number of processes in the virtual topology. If there are more processes in the group than in the topology, the system decides which processes are used.

A mapping can be assigned to a process group at any time. It is a synchronization point for all processes of the group, and all participating processes must call the *same* mapping function with consistent parameters.

If a group has been assigned a cartesian topology, the function MPI_PARTC can be used to partition the group into subgroups which are assigned certain sections 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 inquiry functions MPI_INQCART and MPI_INQPID consistently refer to the grid sections if applied to a subgroup.

MPI_PARTC combines the assignment of a virtual topology with the creation of groups. The reason is that if in an application a group is partitioned into subgroups, this is most likely done along coordinate lines in a virtual topology. It therefore seems to be reasonable to provide an automatic function for this operation.

The virtual topology of a process group can be changed whenever necessary. The old topology is overwritten. This does not affect the topology of subgroups. As an example, assume the ALL group has been assigned a two–dimensional torus mapping, and MPI_PARTC has been called for partitioning the original topology into process rings. Then, if MPI_GRAPH assignes a graph topology to the ALL group, the previous group partitions still exist, and they still have the ring topology.

The definitions and detailed descriptions of the proposed MPI topology functions can be found in appendix A.

4.5. A simple example

The example in figure 4.1 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 PID numbers of its neighbors in the four directions. The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine `relax`.

In each relaxation step each process computes new values for the solution grid function at all points owned by the process. Then the values at interprocess boundaries have to be exchanged with neighboring processes. For example, the exchange subroutine might contain a call like MPI_SEND(...,neigh_pid(1),...) to send updated values to the left–hand neighbor (i-1,j). (We can be more precise when the definition of MPI_SEND is finished.) Note that there is only local communication in this exchange operation.

```
      integer ndim, mx_neigh
      parameter (ndim=2, mx_neigh=4)
      integer gid, dims(ndim), neigh_def(2,mx_neigh), neigh_pid(mx_neigh), num_neigh, ret
      integer own_position(ndim), period(ndim), i, j
      real*8 u(0:101,0:101), f(0:101,0:101)
      data dims / ndim * 0 /
C     Create a grid structure in ALL group and inquire about own position
      gid = ALL
C     Set process grid size and periodicity
      ret = MPI_MAKDIM(gid, ndim, dims)
      if (ret .lt. 0) call error_handler(ret)
      period(1) = MPI_PER
      period(2) = MPI_PER
      ret = MPI_CART (gid, ndim, dims, period)
      if (ret .lt. 0) call error_handler(ret)
C     If not a grid process: stop
      if (ret .eq. 0) stop
      ret = MPI_INQCART (gid, ndim, dims, period, own_position)
      if (ret .lt. 0) call error_handler(ret)
C     Look up the PID numbers 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(1,1) = i-1
      neigh_def(2,1) = j
      neigh_def(1,2) = i+1
      neigh_def(2,2) = j
      neigh_def(1,3) = i
      neigh_def(2,3) = j-1
      neigh_def(1,4) = i
      neigh_def(2,4) = j+1
      num_neigh = mx_neigh
      ret = MPI_INQPID (gid, num_neigh, neigh_def, neigh_pid)
      if (ret .lt. 0) call error_handler(ret)
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_pid)
10    continue
      call output (u)
      end
```

Figure 4.1: Set–up of process structure for two–dimensional parallel Poisson solver.

5. Language Binding

6. Formal Semantics

7. Environmental Inquiry

8. Profiling

8.1. Objectives

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, without requiring access to the source code of the MPI implementation itself.

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.

Since 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 requirement of the profiling interface is solely that a differently named set of MPI routines should be made available, anything beyond that is an implementation issue.

The examples below show one way in which this objective could be met on a Unix system.

8.2. Logic of the design

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.

The suggested solution to this problem is to ask implementors of MPI libraries to provide a mechanism by which it is possible to access the MPI functions, but with a systematically different set of function names.

This allows 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 to achieve the desired effects.

8.3. Examples

8.3.1. 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 :–

```
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 = P_MPI_SEND(tag,bytes,buffer,who);   /* Pass on all the arguments */

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

   return result;
}
```

8.3.2. MPI library implementation

On a Unix system, in which the MPI library is implemented in C, then there are various possible options, depending on whether the linker and compiler support weak symbols (see below for a description of weak symbols).

Systems without weak symbols

In the absence of weak symbols then a possible solution would 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)(int n, void *buffer, int target)
{
    /* 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.

When the linker does not support weak symbols, then it is recommended 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.) If this is the case then the author of the profiling library need only define those MPI functions which she wishes to intercept, and 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.

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 = P_MPI_EXAMPLE

int P_MPI_EXAMPLE(int n, void * buffer, int target)
{
    /* 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.

8.4. 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 (*PRECISE SHIFT TO BE DETERMINED :* `P_` *prefix ?*).
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.

[EDITOR this next section could do with being in a different style, to make clear that it is not a part of the proposal, rather annotation for the committee]

8.5. Outstanding issues

1. Should we define the prefix, or should we let each implementor choose for themselves ?
2. Can we come up with a reasonably portable scheme which can encompass nesting of profiler interfaces ? The scheme outlined above only gives one level, and I don't have a solution for multiple levels.
3. Probably lots of other things...

9. Communication Contexts

9.1. References

[1] V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint.

[2] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint.

[3] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. A users' guide to PVM parallel virtual machine. Technical Report TM-11826, Oak Ridge National Laboratory, July 1991.

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

[5] 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.

[6] 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.

[7] J. Dongarra, R. Hempel, A. Hey, and D. Walker. A proposal for a user–level, message–passing interface in a distributed memory environment. Technical Report TM-12231, ORNL, October 1992.

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

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

[10] G. 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.

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

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

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

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

A. Process Topology Routines

In this appendix Fortran specifications for the proposed topology functions are given.

- **MPI_CART** Map a cartesian structure (grid/torus)
- **MPI_MAKDIM** Divide a process group among n cartesian dimensions
- **MPI_PARTC** Partition a cartesian structure
- **MPI_GRAPH** Map a graph
- **MPI_INQMAP** Check mapping type
- **MPI_INQCART** Find size of cartesian structure and own position
- **MPI_INQGR** Find own node number in graph
- **MPI_INQPID** Look up PID numbers

NAME

MPI_CART Arrange the processes of a process group in a cartesian structure (torus/grid) with `n` dimensions.

SYNOPSIS

integer function MPI_CART (gid, ndim, dims, period)
integer gid
integer ndim
integer dims(*)
integer period(*)

INPUT ARGUMENTS

gid	the ID number of the process group to be structured.
ndim	the number of dimensions of the torus.
dims	number of processes in each coordinate direction.
period	periodicity specification in each coordinate direction.

DESCRIPTION

The processes of the group with ID number `gid` are arranged in a cartesian structure (grid/torus). By assigning neighboring positions in the logical process torus to neighboring processors in the hardware the system can optimize the locality of inter–process communications. The number of dimensions is specified by `ndim`. Array `period` specifies the periodicity in each coordinate direction. If `period(i)` is set to **MPI_PER** then dimension `i` is treated as periodic. If `period(i)` is set to **MPI_NPER** then dimension `i` is treated as non-periodic. If all `period(i)` are set to **MPI_PER**, then the topology is a torus.

Note that a n-dimension hypercube is a torus with 2 processes per coordinate direction and MPI_PER. Thus, ndim=n, dims(1:n)=2 and period(1:n)=MPI_PER.

RETURN VALUE

If the calling process is assigned a position in the grid, **MPI_CART** returns the total number of processes in the grid. If the calling process is not in the grid, the return value is set to 0. A value of -1 is returned if an error occurs.

NAME

MPI_MAKDIM Divide the processes of a process group among n dimensions of a Cartesian grid.

SYNOPSIS

integer function MPI_MAKDIM (gid, ndim, dims)
integer gid
integer ndim
integer dims(*)

INPUT ARGUMENTS

gid	the ID number of the process group to be structured.
ndim	the number of dimensions of the torus.
dims	number of processes in each coordinate direction.

OUTPUT ARGUMENTS

dims	number of processes in each coordinate direction.

DESCRIPTION

The processes of the group with ID number `gid` are divided as evenly as possible among `ndim` dimensions, subject to the constraint that all processes in the group 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 `dims(i)` is positive and does not evenly divide the number of processes in the group. 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 `dim(*)` is suitable for use as input to routine MPI_CART.

RETURN VALUE

If the routine completes without error, the return value is set to 0. A value of -1 is returned if an error occurs.

NAME

MPI_PARTC Partition a process group with cartesian structure in subgroups.

SYNOPSIS

integer function MPI_PARTC (gid, lostdim)
integer gid
integer lostdim

INPUT ARGUMENTS

gid	the ID number of the process group to be partitioned.
lostdim	the coordinate direction by which the process map will be reduced.

DESCRIPTION

It is assumed that the processes of the group with ID number `gid` are arranged in a cartesian structure of size (n1,n2,...) by a previous call to MPI_CART. Then, MPI_PARTC creates a number of subgroups with cartesian structures, the dimension of which is one lower than its parent's. The direction of splitting of the parent structure is specified by `lostdim`. If, for example, `lostdim = 2`, then `n2` subgroups with size (n1,n3,n4,...) are formed.

RETURN VALUE

Upon successful completion **MPI_PARTC** returns the group ID number of the subgroup to which the calling process belongs. A value of -1 is returned if an error occurs.

NAME

MPI_GRAPH Arrange the processes of a process group in a general graph structure.

SYNOPSIS

integer function MPI_GRAPH (gid, nodes, index, nglist)
integer gid
integer nodes
integer index(∗)
integer nglist(∗)

INPUT ARGUMENTS

gid	the ID number of the process group to be structured.
nodes	the number of nodes in the graph.
index	index vector into vector nglist (see below).
nglist	for each graph node: list of neighboring nodes (see below).

DESCRIPTION

The processes of the group with ID number `gid` are arranged in a graph structure. By assigning neighboring graph nodes to neighboring processors in the hardware the system can optimize the locality of inter–process communications. The number of nodes is specified by `nodes`.

Input to MPI_GRAPH is for each node its set of neighbors, 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 ($1 \leq i \leq nodes$) 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.

RETURN VALUE

If the calling process is assigned a position in the graph, **MPI_GRAPH** returns its node number. If it is not in the graph, a value of 0 is returned. A value of -1 is returned if an error occurs.

NAME

MPI_INQMAP Find out mapping type of a process group.

SYNOPSIS

integer function MPI_INQMAP (gid)

INPUT ARGUMENTS

gid the ID number of the process group.

DESCRIPTION

The function MPI_INQMAP tells the calling process the type of mapping which has been applied to the process group with the ID number `gid`, of which the calling process is a member.

RETURN VALUE

Upon successful completion **MPI_INQMAP** returns

MPI_NOMAP:	if no mapping has been assigned,
MPI_CARTMAP:	in case of a cartesian mapping, and
MPI_GRAPHMAP:	in case of a graph mapping.

A value of -1 is returned if an error occurs.

NAME

MPI_INQCART Look up cartesian structure and own position.

SYNOPSIS

integer function MPI_INQCART (gid, maxdim, size, period, pos)
integer gid
integer maxdim
integer size(*)
integer period(*)
integer pos(*)

INPUT ARGUMENTS

gid	the ID number of the process group with a cartesian structure.
maxdim	the dimension of vectors `size` and `pos` in the calling program.

OUTPUT ARGUMENTS

size	vector returning the size of the grid in each coordinate direction.
period	periodicity characteristics in each coordinate direction.
pos	vector returning the position of the calling process in the grid.

DESCRIPTION

After the processes of the group `gid` have been assigned a cartesian structure (function MPI_CART), the function MPI_INQCART tells the calling process the parameters of that structure and its own logical position.

If, for example, a three–dimensional grid mapping has been applied, then `size` returns the three values `nx,ny,nz`, and `period(i)`, i=1,2,3, is set to **MPI_NPER**. The vector `pos` contains the logical process coordinates `i,j,k` with $1 \leq i \leq nx$, $1 \leq j \leq ny$, and $1 \leq k \leq nz$. If the calling process is not assigned a position in the grid structure, the coordinates are set to zeroes. If the vectors `size`, `period` and `pos` are too short, MPI_INQCART reports an error.

RETURN VALUE

Upon successful completion **MPI_INQCART** returns the number of dimensions of the cartesian structure, `ndim`. A value of -1 is returned if an error occurs.

NAME

MPI_INQGR Look up size of graph mapping and position of calling process.

SYNOPSIS

integer function MPI_INQGR (gid, nodes)

INPUT ARGUMENTS

gid the ID number of the process group with a graph structure.

OUTPUT ARGUMENTS

nodes the total number of nodes in the graph structure.

DESCRIPTION

After the processes of the group `gid` have been structured as a graph (function MPI_GRAPH), the function MPI_INQGR tells the calling process the size of the graph and its own node number. Nodes are numbered from 1 to `nodes` (cf. function MPI_GRAPH).

RETURN VALUE

Upon successful completion **MPI_INQGR** returns the node number `i` of the calling process, $1 \leq i \leq nodes$. If the calling process is not assigned a node in the graph, zero is returned. A value of -1 is returned if an error occurs.

NAME

MPI_INQPID Look up PID numbers of selected processes.

SYNOPSIS

integer function MPI_INQPID (gid, nproc, coord, plist)
integer gid
integer nproc
integer coord(∗)
integer plist(∗)

INPUT ARGUMENTS

gid	the ID number of the process group with a cartesian or graph structure.
nproc	the number of processes for which the PID numbers are to be looked up (see below).
coord	the logical process coordinates of the specified processes (see below).

OUTPUT ARGUMENTS

plist	the list of PID numbers of the specified processes.

DESCRIPTION

Usually a process only communicates with a relatively small number of other processes in a large process structure (a cartesian structure or a graph). Given the logical coordinates of those other processes, MPI_INQPID looks up the corresponding PID numbers. Those PID numbers can then be used for the message–passing.

The processes of which the PID numbers are required are specified by the input arguments `nproc` and `coord`. Two mechanisms are available:

1. Separately specified processes: `nproc` is set to the number of processes $1 \leq nproc$. Vector `coord` contains the logical coordinates for each process. In periodic directions the coordinates are interpreted modulo the number of processes in the corresponding direction (torus topology). As an example, assume the processes are mapped as a two–dimensional torus, and the PID numbers of three processes are to be looked up. Then, `nproc` is set to 3, and the first 6 entries of `coord` are set to $i_1, j_1, i_2, j_2, i_3, j_3$.
   Similarly, if the mapping was done with MPI_GRAPH, the entries of vector `coord` are set to the logical node numbers of the specified graph processes.
2. No individual selection: in this case `nproc` is set to a negative number $nproc \leq -1$. Vector `plist` returns the PID numbers of the *first* `|nproc|` processes. The vector `coord` is not used in this case. Note that this feature does not scale to very high process numbers. In small applications, however, it can be used to get a global list of all PID numbers in the map.

RETURN VALUE

Upon successful completion **MPI_INQPID** returns 0. A value of -1 is returned if an error occurs.