D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

April 24, 1993
This work was supported by ARPA and NSF under contract number ###, by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and by the Commission of the European Community through Esprit project P6643.

Contents

Abstract

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

This is a draft of what will become the Final Report, Version 1.0, of the Message Passing Interface Forum. This document contains all the technical features proposed for the interface. This copy of the draft was processed by LaTeX on April 24, 1993.

MPIF invites comments on the technical content of MPI, as well as on the editorial presentation in the document. Comments received before July 1, 1993 will be considered in producing the final draft of Version 1.0 of the Message Passing Interface Specification.

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

Acknowledgments

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Message Passing Interface (MPI), many people served in positions of responsibility:

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point to Point Communications
- Al Geist, Marc Snir, Steve Otto, Collective Communications
- Rolf Hempel, Process Topologies
- Scott Berryman, Language Binding
- Steve Zenith, Formal Semantics
- William Gropp, Environmental Inquiry
- James Cownie, Profiling
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Marc Sears, Communication Contexts
- Steve Otto, Editor.

Do we want an organization/company list here?

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

Ed Anderson	Jim Feeney	Jon Flower
Daniel Frye	Ian Glendinning	Adam Greenberg
Robert Harrison	Leslie Hart	Tom Haupt
Tom Henderson	C. T. Howard Ho	Steven Huss-Lederman
John Kapenga	Bob Leary	Barney Maccabe
Phil McKinley	Chuck Mosher	Dan Nessett
Paul Pierce	Peter Rigsbee	Ambuj Singh
Robert G. Voigt	Dennis Weeks	Stephen Wheat

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

Talk about financial support here.

Chapter 1

Introduction to MPI

1.1 Overview and Goals

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

In designing MPI we have sought to make use of the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI has been strongly influenced by work at the IBM T. J. Watson Research Center [1, 2], Intel's NX/2 [16], Express [15], nCUBE's Vertex [13], P4 [3], and PARMACS [11, 12]. Other important contributions have come from Zipcode [17, 18], Chimp [7, 8], PVM [9, 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.

The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. Although no explicit support for threads is provided, the interface has been designed so as not to prejudice their use.

1.4 What Is Included In The Standard?

The standard includes (this is temporarily as inclusive as possible):

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

1.5 What Is Not Included In The Standard?

The standard does not specify:

- Explicit shared-memory operations
- Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages
- Program construction tools
- Debugging facilities
- Auxiliary functions such as timers
- Explicit support for threads

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

Chapter 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 sections 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. The actual binding of these objects in Fortran and C will be discussed in the language binding subcommittee.

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. The mechanism for opaque objects used here loosely follows the POSIX Fortran binding standard.

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).

Opaque object are created by calls that are specific to each object type. They are destroyede by a call to `MPI_FREE`. Additional MPI functions are available to access and update specific opaque objects.

MPI_FREE(handle)

IN handle handle to object

Mark the handle for destruction. The object can destroyed as soon as there there is no pending operation that is using this object. This is equivalent to changing the persistence of the handle from `persistent` to `ephemeral`. For example, if a communication object is freed, after the communication started but before it completed, then the pending operation is not affected; one can use the handle in calls to `MPI_WAIT` or `MPI_STATUS`. However, once the operation completes, it is erroneous to reuse the freed handle for a new communication.

Discussion:
An alternative design point was discussed, whereby it is the user responsibility to free a handle only if there are no pending operations on this handle. The current design was deemed more elegant. Note that it is not necessary to destroy the handle as soon as it is not needed anymore. A valid implementation may just mark freed handles and garbage collect them periodically.

On the other hand, some objects, such as buffer descriptor objects, can be used by several distinct concurrent communication operations. Thus, to find out whether such object can be freed, one would need a reference count of pending operations. This complication was not observed at our last meeting, and may be a good argument to reconsider the current definition of `MPI_FREE`.

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.

2.2.2 Array of handles

An MPI call may need a parameter that is an *array of handles*. The array-of-handles object is not self-delimiting; it does not carry information on the number of entries in the array. This information has to be provided by an additional `len` parameter, whenever a list-of-handles parameter is used.

Discussion:
The reason for not having self-delimiting arrays is that records are not first-class citizens in Fortran and that having an array where one entry encodes length, whereas the other entries encode handles was deemed to be too error-prone. This line of argumentation really implies that we do not view array-of-handles as an opaque object, and that we expect the user to build these objects explicitly, not via MPI calls.

2.2.3 State

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

2.2.4 Named constants

MPI procedures sometimes assign a special meaning to a special value of a basic type parameter; e.g. `tag` is an integer valued parameter of point-to-point communication operations, with a special `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).

Discussion:
Need to agree on a language mechanism for named constants.

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

2.2.5 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.

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

Discussion: This material belongs to an introduction or environment section

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 (total number of processes is fixed), 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.

2.3.1 Error Handling

Discussion: Material of this section should be moved to a general introduction section. This material has not yet been apporived by MPIF

MPI provides the user with reliable message transmission: A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. Where the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementer of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as global program failures. Similarly MPI itself provides no mechanisms for handling node failures.

Of course, MPI programs may still be erroneous. A **program error** can occur when an MPI call is called with an incorrect parameter (non-existing destination in a send operation, buffer too small in a receive operation, etc.) This type of error would occur in any implementation. In addition, a **resource error** may occur when a program exceeds the amount of available system resources (number of pending messages, system buffers, etc.). The occurrence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used; this may differ from system to system. The recommended implementation 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 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.
One might want different signals for different modules.
The details of such proposal need be elaborated in an appropriate "profile" subcommittee.

2.4 Messages

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

2.4.1 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 77, a message consists of a sequence of values that are each of type `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `DOUBLE PRECISION COMPLEX`, `LOGICAL`, or (length 1) `CHARACTER`. A message may also contain a value of type `BYTE`, which consists of 8 binary digits. A byte is different from a character. Diferent machines may have different representations for the same character, or may use more than one byte to reprensent a character. On the other hand, a byte has the same binary value on any machine. A message may mix values of different types.

Missing:
Need to agree on the C types (including handling of signed/unsigned), and on Fortran 90 types (including handling of kinds).

2.4.2 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 specified context. `source = destination` is allowed: a process can send a message to itself.

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.

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

`Context` should be a context shared by both source and destination.

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.

2.5 Communication Buffers

The basic point to point communication operations are **send** and **receive**. A **send** operation creates a message; the message data is taken from the **send buffer**. A **receive** operation consumes a message; the message data is put 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 of (not necessarily distinct) variables of the same basic datatype. The datatype of such variable is specified by a state parameter. The possible values for Fortran are `MPI_INT`, `MPI_REAL`, `MPI_DOUBLE`, `MPI_COMPLEX`, `MPI_DCOMPLEX`, `MPI_LOGICAL`, MPI_CHAR and `MPI_BYTE`, corrsponding to datatypes `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `DOUBLE PRECISION COMPLEX`, `LOGICAL`, `CHARACTER` and to untyped `BYTE`. The possible values for C are `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE` and `MPI_BYTE`.

Discussion:
May need to take care of kinds in Fortran 90 and of signed/unsigned in C. More elaborate definitions belong to the language binding.

There are five kinds of buffer components:

Contiguous 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

Thus, if `A` is an array of integers, then `A(1), 4, MPI_INT` is the buffer component with entries `A(1), A(2), A(3), A(4)`.

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` $>$ `0`)

datatype Type of elements

Assume that an element of type `datatype` occupies `m` bytes. Then, in a byte addressable machine, the buffer component described by these five parameters consists of the `len` variables at addresses `start, start+m, ... , start+m*(lenblk-1), start+m*stride, start+m*(stride+1), ..., start+m*(stride+lenblk-1), ...`. For example, if `A` is an array of integers, then

- `A(1), 5, 3, 2, MPI_INT` is the buffer component with entries `A(1), A(2), A(4), A(5), A(7)`.
- `A(1), 5, 0, 2, MPI_INT` is the buffer component with entries `A(1), A(2), A(1), A(2), A(1)`.
- `A(10), 5, -1, 2, MPI_INT` is the buffer component with entries `A(10), A(11), A(9), A(10), A(8)`.

`Contiguous` is a special case of `vector`. A vector buffer component can be used to extract an arbitrary submatrix of a two-dimensional matrix.

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

start initial element

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

array_of_indices Array of displacements of the elements in the buffer components, relative to the initial element.

datatype Type of elements

For example, if `A` is an array of integers, and `I` is an array with values `2, -1, 2, 1` then `A(3), 4, I, MPI_INT` is the buffer compontent with entries `A(5), A(2), A(5), A(4)`.

`Vector` is a special case of `indexed`.

Heterogeneous Vector (hvector) Same as `vector`, except that `stride` is given in multiple of bytes, rather than multiples of elements:

start Initial element

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

stride Number of bytes between the start of each block

lenblk Number of elements in each block (`lenblk` > 0)

datatype Type of elements

Assume that an element of type `datatype` occupies `m` bytes. Then, in a byte addressable machine, the buffer component described by these five parameters consists of the `len` variables at addresses `start, start+m, ... , start+m*(lenblk-1), start+stride, start+stride+m, ..., start+stride+m*(lenblk-1), ...`.

Consider a C array declared as

```
struct {
   char info[3];
   short number;
   double val;
       }   a[100]
```

Assume that all structure components are byte aligned, a `short` occupies two bytes, and a `double` occupies eight bytes. Then the buffer component `&a, 100, 13, 2, MPI_CHAR` extracts the first two characters of each structure. The buffer component `&(a.number), 100, 13, 1, MPI_SHORT` extracts the `short` field of each structure; and the buffer component `&(a.val), 100, 13, 1` extracts the `double` field of each structure. Thus, heterogeneous vectors can be used to extract vectors of homogeneous values from heterogenous data.

`Vector` is a special case of `heterogenous vector`.

Heterogeneous indexed (hindexed) Same as indexed, except that displacements are in bytes, rather than multiples of element size.

start initial element

len Number of elements (`len` ≥ 0)

array_of_indices Array of byte displacements of the elements in the buffer components, relative to the initial element.

datatype Type of elements

For example, if `a` is the C array of the previous example, and `I` is an array with values `0, 13, 13, 27, 26` then `&a, 5, I, MPI_CHAR` is the buffer component with entries `a[0].info[0], a[1].info[0], a[1].info[0], a[1].info[1], a[1].info[0]`.

Like `heterogenous vector`, `heterogenous indexed` extends the `indexed` facility so as to allow the extraction of homogeneous data from a heterogenous layout.

Discussion:
The use of displacements in indexed buffer components has the same effect as indirect addressing into an array with initial index zero. I.e., if `a` is an array with initial index zero, then a buffer component defined by parameters `a[0], N, i, type` extract the elements `a[i[0]], a[i[1]], ..., a[i[n]]`. This fits well C usage but ill-fits Fortran usage (initial index one). We took a decision of principle to force "start from zero" index arithmetic on Fortran users, rather than have diferent index arithmetic in the C and Fortran binding of MPI.

The use of both buffer components with byte and element size displacements is motivated by the observation that some applications require the more general byte displacement, but that most applications would use element size displacement, which is much more natural and convenient (and machine independent). Thus, rather than forcing all users to do byte arithemetic, we decided to provide separately the more advanced function to "expert programmers".

A buffer is described by an opaque object accessed via a **buffer handle**. Such object is created by `MPI_CREATE_BUFFER`. It is associated with successive buffer components by calling in succession one of the functions `MPI_APPEND_xxx`, where `xxx` identifies the type of buffer component appended to the buffer. It becomes available for use in communication operations once committed by the function `MPI_COMMIT`. A buffer descriptor is destroyed after the completion of the first communication operation that uses it, if it is ephemeral, or after it is freed by a call to `MPI_FREE` and any pending communication operation that uses it has completed, if it is persistent. After the buffer descriptor object has been destroyed, the handle is undefined.

MPI_CREATE_BUFFER(buffer, persistence)

Create a new buffer descriptor object. The parameters are:

OUT buffer buffer handle

IN persistence buffer persistence (state type with possible values `MPI_PERSISTENT` or `MPI_EPHEMERAL`).

Discussion:
May want to provide information on maximum object size, and perhaps user space for the object.

MPI_APPEND_CONTIGUOUS(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_APPEND_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_APPEND_INDEXED(buffer, start, len, array_of_indices, datatype)

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

INOUT buffer buffer handle

IN start initial position for indexing (choice)

IN len Number of elements (integer)

IN array_of_indices array of displacements of entries relative to start (array of integers)

IN datatype datatype identifier (status)

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

Append a heterogenous 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 bytes between the start of each block (integer)

IN lenblk Number of elements in each block (integer)

IN datatype datatype identifier (status)

MPI_APPEND_HINDEXED(buffer, start, len, array_of_indices, datatype)

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

INOUT buffer buffer handle

IN start initial position for indexing (choice)

IN len Number of elements (integer)

array_of_indices array of byte displacements of entries relative to start (array of integers)

IN datatype datatype identifier (status)

MPI_COMMIT_BUFFER(bufferin)

Commit a buffer descriptor object. The object cannot be changed anymore after it is committed. A buffer descriptor object can be used for communication only after it is committed. The parameters are:

INOUT buffer buffer descriptor handle

Discussion:
The commit function may do preprocessing of the buffer descriptor to simplify its use and commit resources in the communication subsystem for its use. The commit function may generate a new object and, hence, modify the object handle.

Consider, for example, the following fragment of Fortran code

```
DOUBLE PRECISION A(10,20)
INTEGER B, C(5,10)
INTEGER BH
...
CALL MPI_CREATE_BUFFER(BH, MPI_PERSISTENT)
CALL MPI_APPEND_BLOCK (BH, B, 1, MPI_INT)
CALL MPI_APPEND_VEC (BH, A(1,3), 11, 4, 2, MPI_DOUBLE)
CALL MPI_APPEND_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.

The same entry may appears more than once in a buffer component, and distinct buffer components may overlap. If a buffer descriptor with replicated entries is used to receive a message, then the final value of an entry that occurs more than once in the buffer descriptor is undefined.

Discussion:
It is not clear to me that overlaps have any useful purpose, and they may complicate implementation. But we had an overwhelming vote in favor of this feature.

2.5.1 Type Matching and Data Conversion

One can think of message transmission as consisting of three phases:

1. Data is pulled out of the send buffer and a message is assembled
2. A message is transfered from sender to receiver
3. Data is pulled from the incoming message and disassembled into the receive buffer

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type of the corresponding entry in the buffer descriptor; the type of an entry in the buffer descriptor used to send a message has to match the type of the corresponding entry in the buffer descriptor used to receive the message; and the type of each variable in the receive buffer has to match the corresponding entry in the buffer descriptor for this buffer. A program that fails to observe these three rules is erroneous.

To define more precisely type matching, we need to deal with two issues: matching of types of the host language with buffer descriptor types; and matching of buffer descriptor types at sender and receiver. We restrict ourselves to the case where sending and receiving program are written in the same language.

The type of an entry in the sender buffer descriptor matches the type of an entry in the receiver buffer descriptor is they have identical names: MPI_INT matches MPI_INT, MPI_REAL matches MPI_REAL, and so on.

The type of a variable in a host program matches the type of the entry in a buffer descriptor if the name of the buffer descriptor entry type corresponds to the basic type of the host program entry: an entry with type name MPI_INT matches a Fortran variable of type INTEGER, an entry with type name MPI_REAL matches a Fortran variable of type REAL, and so on. There is one exception to this last rule: An entry with type name MPI_BYTE can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte. The value of the message entry will be the binary value of the corresponding byte in memory.

These type matching rules imply that no type conversion occur during the MPI communication. If a value transfered in a message is typed (i.e. of a type different from BYTE), then the type of the variable the value is read from, the type of the value transferred in the message, and the type of the variable where the value is stored at the receiver all match, and match the types of the corresponding entries in the buffer descriptors at sender and receiver. The value stored at the receiver will be the value loaded from the sender. If sender and receiver run in an identical environment (same node architecture, same compiler, etc.), then the binary representation of the data stored at the receiver is identical to the binary representation of the data loaded from the sender. However, if sender and receiver have a different environment, then the binary representation of the value sent will be converted to the binary representation of that same value in the receiver environment. Thus, if a Fortran real value is transferred (from a variable of type REAL to a variable of type REAL, using buffer descriptor entries with type state MPI_REAL at sender and receiver) from a 64 bit machine to a 32 bit machine, then the binary representation of the floating point value sent will be converted from 64 bits to 32 bits. Such conversions may introduce rounding error and cause overflows or underflow conditions.

If a value sent in a message is untyped (i.e., of type BYTE), then it must be declared so in the buffer descriptor of both sender and receiver. A byte is moved from sender to receiver; the binary representation of the byte stored at the receiver is identical to the binary representation of the byte loaded at the sender. This holds true, whether sender and receiver run in the same or in distinct environments.

Consider the following examples.

```
!  FIRST EXAMPLE
INTEGER A(100)
...
CALL MPI_CREATE_BUFFER(BH, MPI_PERSISTENT)
CALL MPI_ADD_CONTIGUOUS (BH, A(1), 1, MPI_CHAR)
```

This program is erroneous, since the type of the buffer descriptor does not match the type of the variable in the buffer. The behavior or this program is undefined. A desireable behavior is for that error to be detected (at compile time or run time) and result in an error condition.

```
!  SECOND EXAMPLE
INTEGER A(100)
...
CALL MPI_CREATE_BUFFER(BH, MPI_PERSISTENT)
CALL MPI_ADD_CONTIGUOUS (BH, A(1), 1, MPI_BYTE)
```

This program is correct. The byte that will be transmitted in a message created with this buffer descriptor is the first byte in the storage occupied by `A(1)`. The value of this byte is implementation dependent, and will vary according to the number of bytes used to store Fortran variables of type `INTEGER` and the byte ordering on the host machine.

```
!  THIRD EXAMPLE
INTEGER A(100)
...
CALL MPI_CREATE_BUFFER(BH1, MPI_PERSISTENT)
CALL MPI_ADD_CONTIGUOUS (BH1, A(1), 1, MPI_INTEGER)
...
CALL MPI_CREATE_BUFFER(BH2, MPI_PERSISTENT)
CALL MPI_ADD_CONTIGUOUS (BH2, A(1), 4, MPI_BYTE)
```

Consider the following three cases:

1. A message is sent using buffer descriptor handle `BH1` and received using buffer descriptor handle `BH2`. The program is erroneous, since the sender and receiver buffer descriptors don't match. The behavior of the program is undefined, and a possible (desireable but unlikely) behavior is for an error condition to occur.

2. Both sender and receiver use buffer descriptor handle `BH1`. The program is correct, and an integer value is transmitted from sender to receiver. The value assigned to `A(1)` at the receiver is identical to the value of `A(1)` at the sender, but the binary representation of this value may be different, if sender and receiver run in different environments.

3. Both sender and receiver use buffer descriptor handle `BH2`. The program is correct. Assume that both sender and receiver use four bytes to store integers. Then the value stored in `A(1)` at the receiver has the same binary representation as the value of `A(1` at the sender. If the same byte ordering and the same encoding of integer values is used at both ends then `A(1)` at the receiver is set to the value of `A(1)` at the sender; otherwise the same 32 bits may encode different values.

Implementation note:
The current definition does not require messages to carry type information. A message can be composed and sent using only the information in the sender buffer descriptor, and can be received and stored using only the information in the receiver buffer descriptor. If messages are sent between

different machines then one can either use a "universal" data encoding for messages, use knowledge of the receiver environment in order to convert data at the sender, or use knowledge of the sender environment in order to convert data at the receiver. In either case the local buffer descriptor can be used to derive the types of the values transferred. However, additional type information carried by messages can provide better error detection.

There is no agreement yet on the need for a definition and, if needed, on the right definition of type matching and conversion between programs written in different languages. The natural interpretation of "send from C and receive in Fortran" is either "send from C, receive in C and convert to Fortran", or "convert to Fortran, send from Fortran and receive in Fortran". However, the correspondence between Fortran types and C types is implementation dependent. Thus, the diagram shown below is not commutative, and the two interpretations of C to Fortran communication may lead to different results.

$$\begin{array}{ccc} \text{machine 1} & & \text{machine 2} \\ \text{C program} & \xrightarrow{\text{send}} & \text{C program} \\ \downarrow & & \downarrow \\ \text{Fortran program} & \xrightarrow{\text{send}} & \text{Fortran program} \end{array}$$

A good solution would make this diagram commutative. Otherwise, one needs to pick one of the two paths as the canonical definition of interlanguage communication. If the upper path is chosen, the receiver of a message needs to know the language the message was sent from; if the lower path is chosen, the sender needs to know the language the message is received into. Rather than struggling with these issues it might be simpler to let the behavior of C to Fortran communication be undefined. The user can always match language at sending and receiving end if he or she is willing to struggle with the issue of calling MPI C functions from Fortran or vice-versa.

2.6 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. DONTCARE values for source or tag are specified via a named constant for source dontcare and a named constant for tag dontcare.

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.

2.7 Communication Mode

A send operation can occur in one of three modes:

STANDARD 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.

SECURE The send operation will not complete successfully until it is guaranteed that the message sent will be received.

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.

A **secure send** will complete if a matching receive is posted, and the receive operation is committed to receive the message sent by the secure send. (E.g., the send and receive operations have already been matched, the receive operation has already started and cannot be cancelled anymore.)

A receive operation can occur in one of two modes: **STANDARD** and **SECURE**. A secure send can be matched only by a **secure receive**; a program where a secure send can be matched by a standard receive is erroneous and its behavior is undefined. The secure receive can be posted either before or after the matching send.

A standard send and a ready send can both be matched only by a **standard receive**; a program where a secure receive can match a nonsecure send is erroneous and its behavior is undefined. The receive has to be posted ahead of the send for a ready send and can be posted either before or after the send is posted for a standard send.

Implementation note:

The current definition assumes it si the user responsibility to prevent matching of a regular send with a secure receive and vice-versa. Otherwise, messages need be tagged as standard or secure.

Assume an implementation of MPI communication that avoids retries. What protocols can be used? Since a receive may have a wildcard source field, the first step in the protocol must be executed by the sender.

For a ready send, this first step may involve sending the data, since the receiver is guaranteed to have space to store it. The protocol is send, possibly followed by an ack-send.

The protocol for secure send is likely to be the following: The sender sends a request-to-send message. The receiver stores this request (which contains a copy of the message envelope). When a matching receive is posted, the receiver sends back an acknowledge, and the sender now sends the message. The protocol is (req, ack-req, send), possibly followed by an ack-send.

What protocol is used for standard sends? Either the second protocol (req, ack-req, send) or a mix of the first and the second (e.g. first used for short messages, second for long messages).

When the receiver does not wildcard the source field then the protocol can be reversed: The receiver sends a clear-to-send message, and the sender can proceed with the short protocol if it has a clear-to-send when the send is executed.

What do we gain from the "secure receive"? Some optimizations for regular receives (e.g., in a regular receive of a short message the receiver knows the short protocol will be used, and need not check for a request-to-send). Q: is that enough justification for the secure receive?

2.8 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

$$\textbf{INIT (START COMPLETE)}^* \textbf{ FREE}.$$

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.

2.8.1 Communication Object Creation

Discussion: This section has not yet been approved

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 context of messages sent with this handle

IN mode send mode (state type, with three values: `MPI_STANDARD`, `MPI_READY` and `MPI_SECURE`)

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

MPI_INIT_RECV (handle, buffer_handle, source, tag, context, mode, 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 `MPI_ANY_SOURCE` (integer).

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

IN context context of messages received with this handle.

IN mode receive mode (state type, with two values: `MPI_STANDARD`, and `MPI_SECURE`)

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

See Section 2.4.2 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.

2.8.2 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.

2.8.3 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.

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_QUERY(handle, len, source, tag)

IN handle handle to return status object

OUT len number of elements received.

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

OUT tag tag of received message (integer).

MPI_CREATE_STAT(handle)

Creates a new return status object.

OUT handle handle to return status object.

Discussion:
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. The main reason for the use of a return-status object is that one wants to be able to use the same `MPI_WAIT` and `MPI_STATUS` calls for checking on different types of operations, e.g. both sends and receives, and perhaps new operations in the future. These different operations return different information (albeit we could decree that a send returns the same information as a receive).

The issue of memory allocation for return_status handles is not yet solved. There is a desire that the user will be able to allocate space for them (e.g., on the stack).

We return number of elements received, rather than bytes received because "number of elements" is closer to the application semantic level (user does not need to be aware of the size of elements), and is more invariant. The return status object returns the number of **elements** received. If there was no truncation, then this is equal to the number of elements sent. On the other hand, the number of bytes collected form the sender memory, the number of bytes sent over the wire, and the number of bytes stored in the receiver memory may all be different, in a heterogeneous environment. The number of elements sent can be computed form the send buffer descriptor; the number of elements received can be computed form the receive buffer descriptor and the length (in bytes) of the received message.

2.8.4 Multiple Completions

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

IN len list length (integer)

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)` has the same effect as 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, and the cancelation of all remaining wait operations.

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

`MPI_WAITANY ( list_of_handles, len, index, return_status_handle)` is

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

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

MPI_STATUSANY (list_of_handles, len, 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.

IN len list length (integer)

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, len, 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.

IN len lists length (integer)

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.

Discussion:

The fairness requirement given in Section 2.13 implies that the use of `WAIT_ANY` cannot lead to starvation: If the sending process has issued a send complete operation, and the receiving process repeatedly post a receive for the message sent, then that message must be eventually received. Section 2.13 has not yet been discussed. The fairness requirement can be attacked either for being too weak (eventually is not good enough), or too strong (hard to implement). In the later case, if the requirement of fairness is droped from MPI implementations, then some mechanism need be provided to the user to achieve fairness by his or her own devices. One such proposal is to be able to specify a rotating priority order on the operations posted by a `MPI_WAITANY`: search the list sequentially, starting form a user specified position.

2.9 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:

$$\begin{bmatrix} \_ \\ \mathbf{R} \\ \mathbf{S} \end{bmatrix} \begin{bmatrix} \mathbf{SEND} \\ \mathbf{RECV} \end{bmatrix}$$

The first letter (void, **R** or **S**) indicates the start mode (standard, ready, or secure). Only two of the combinations (standard and secure) are meaningful for receives.

MPI_SEND (buffer_handle, dest, tag, context) is

```
MPI_INIT_SEND(handle, buffer_handle, dest, tag, context, MPI_REGULAR, MPI_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, MPI_READY, MPI_EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle, null)
```

MPI_SSEND (buffer_handle, dest, tag, context) is

```
MPI_INIT_SEND(handle, buffer_handle, dest, tag, context, MPI_SECURE, MPI_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, MPI_STANDARD, MPI_EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle,return_handle)
```

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

```
MPI_INIT_RECV(handle, buffer_handle, source, tag, context, MPI_SECURE, MPI_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.

2.10 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** (**S**) indicates the `READY` (**SECURE**) 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, MPI_STANDARD, MPI_EPHEMERAL)
MPI_START(handle)
```

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

```
MPI_INIT_SEND(handle, buffer_handle, dest, tag, context, MPI_READY, MPI_EPHEMERAL)
MPI_START(handle)
```

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

```
MPI_INIT_SEND(handle, buffer_handle, dest, tag, context, MPI_SECURE, MPI_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, MPI_STANDARD, MPI_EPHEMERAL)
MPI_START(handle)
```

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

```
MPI_INIT_RECV(handle, buffer_handle, source, tag, context, MPI_SECURE, MPI_EPHEMERAL)
MPI_START(handle)
```

2.11 Contiguous Block Sending Operations

The most frequent type of buffer used is a buffer with one contiguous component. 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 **C** in the function name, for `CONTIGUOUS`.

MPI_SENDC (start, len, type, dest, tag, context) is

```
MPI_CREATE_BUFFER(buffer_handle, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, type)
MPI_SEND (buffer_handle, len, dest, tag, context)
```

MPI_RSENDC (handle, start, len, type, dest, tag, context) is

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

MPI_SSENDC (handle, start, len, type, dest, tag, context) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, type)
MPI_SSEND (buffer_handle, len, dest, tag, context)
```

MPI_RECVC (start, len, type, source, tag, context, return_status_handle) is

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

MPI_SRECVC (start, len, type, source, tag, context, return_status_handle) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, type)
MPI_SRECV(buffer_handle, source, tag, context, return_status_handle)
```

MPI_ISENDC (handle, start, len, type, dest, tag, context) is

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

MPI_IRSENDC (handle, start, len, type, dest, tag, context) is

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

MPI_ISSENDC (handle, start, len, type, dest, tag, context) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, type)
MPI_ISSEND(handle, buffer_handle, dest, tag, context)
```

MPI_IRECVC(handle, start, len, type, source, tag, context, return_status_handle) is

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

MPI_ISRECVC(handle, start, len, type, source, tag, context, return_status_handle) is

```
MPI_CREATE(buffer_handle, MPI_BUFFER, MPI_EPHEMERAL)
MPI_ADD(buffer_handle, start, len, type)
MPI_ISRECV(handle, buffer_handle, source, tag, context)
```

2.12 Probe and Cancel

The MPI_PROBE operation allows to check for incoming messages, without actually receiving them. The user can then decide where to receive them, based on the information returned by the probe (basically, the information on the message envelope). An additional function, MPI_GET_LEN allows to compute the amount of storage needed to receive the message, when this length is not readily computed from the information returned by MPI_PROBE.

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

MPI_PROBE(source, tag, context, flag, type, return_status)

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

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

IN context context of messages received with this handle.

OUT flag (logical)

IN type assumed type of data in message (status variable).

OUT return_handle handle that is associated with return status object.

`MPI_PROBE` returns `flag = true` if a there is a message that can be received and that matches the pattern specified by the parameters `source`, `tag`, and `context`. It returns `flag = false`, otherwise. If `MPI_PROBE` returns `flag = true`, then the length, source and tag of the message matched are returned in the return status object. These are the same values that would have been returned by a call to `MPI_RECV` or to `MPI_SRECV` executed at the same point in the program (with a caveat concerning length; see below). These values can be decoded from the return status object using the `MPI_RETURN_STAT` function. The value returned in the return status object is undefined if `flag=false`.

The length value returned by the return status object is (correctly) the number of elements in the message, provided that all elements in the message are of the type specified by the `type` parameter; otherwise the length value returned is undefined.

A subsequent receive executed with the same context, and the source and tag returned by the call to `MPI_PROBE` will receive the message that was matched by the probe, if no other intervening receive occurred after the probe. If the receiving process is multithreaded, it is the user responsibility to ensure that the last condition holds.

Discussion:

Do we want `MPI_PROBE` to return mode (standard or secure)? If yes, we need to carry mode information with messages (it's deja vu all over again). If not, then it is up to the user to encode this information (in the tag) so that receiver can decide whether to use a secure or standard send.

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

Message passing in MPI can be implemented without appending type information to messages. A message is merely a string of bytes and the interpretation of these bytes into a sequence of typed elements is done using the information in the buffer descriptors at each end. The ability to use such implementation strategy is deemed to be an important goal. In such implementation, when a message arrives, it is not be known how many elements it contains, or even how much storage is needed to receive that message (because of possible representation conversion in a heterogeneous environment). The probe function cannot use a buffer descriptor; this defeats the purpose of probing in order to decide where to receive a message. Therefore, probe cannot, in general, return correct length information. Still, it is often the case that probe is used to decide how much storage to allocate in order to receive a message. Encoding such information in the message tag is deemed to be too akward, and it is deemed important for `MPI_PROBE` to return some useful size information. The current definition of `MPI_PROBE` is a compromise between these two goals. For the most common case of messages where all entries have the same type `MPI_PROBE` returns the correct length information; the more esoteric case is handled by the `MPI_GET_LEN` that is described below. The current solution saves us the need for one additional word per message that would otherwise be needed to transfer the message length (in elements) with the message.

MPI_GET_LEN(len, return_status, buffer_descriptor)

Computes the number of elements that would be received, if the message that is associated with the return status handle would be received in the buffer associated with the buffer descriptor handle.

OUT len number of elements that would be received (integer)

IN return_status handle to return status descriptor

IN buffer_descriptor handle to buffer descriptor

Discussion:
In order to support this function, and additional field is needed to in the return status object, e.g., number of bytes in the incoming message matched by probe, or value of type parameter provided by probe (unless we add this parameter to the `MPI_GET_LEN` function, which is too ugly to bear).

MPI_CANCEL(handle, flag)

IN handle handle to communication object

OUT flag (logical)

A call to `MPI_CANCEL` cancels a pending communication operation (send or receive). The call returns `flag = true` if the cancel operation succeeeded, `flag = false` otherwise. It must be the case that either the cancel operation succeeds or that the pending communication operation completes (but not both). If a communication operation was cancelled successfully, the effect is as if this operation was never executed. If a send was canceleed successfully then the message sent will not be received, the receive buffer of any posted receive for that message will not be altered, and any such receive has to be satisfied by another send; if a receive was cancelled successfully, then the recieve buffer posted will not be altered, and any send matching this receive has to be satisfied by another receive.

Implementation note:
There is not expectation that a cancel operation will be fast.

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 or, in the case of a send, because the matching receive successfully receives another message.

   An enabled operation either completes successfully or becomes permanently disabled.

   I.e., an enabled operation either eventually succeeds, or becomes disabled (progress); and an operation cannot be enabled infinitely many times without ever succeeding (fairness).

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.

2.14 Missing

- send-receive
- Send and receive from same buffer

Chapter 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 communication buffers and their manipulations. The context section describes the formation, manipulation, and query functions (such as group size) that are available for groups and group objects.

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 communication buffer 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.

3.2 Group Functions

A full description of the group formation and manipulation functions can be found in the context chapter of the MPI document. Here we describe only those group functions that are used in the semantic description of the collective communication routines.

An initial group containing all processes is supplied by default in MPI. MPI provides a procedure that returns the handle to this initial group. The processes in the inital group each have a unique rank in the group represented by integers (0, 1, 2, ..., [number-of-processes - 1].

MPI_ALLGROUP(group) Return the descriptor of the inital group containing all processes.

OUT group handle to descriptor object of initial group.

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

IN group group handle

OUT rank integer

MPI_GSIZE(group, size) Return the number of processes that belong to the specified group.

IN group group handle

OUT size integer

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 data. For each communication operation, we list these two level of calls together.

3.4 Synchronization

Barrier synchronization

MPI_BARRIER(group)

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

`MPI_BARRIER( group )` is

```
MPI_CREATE_BUFFER(buffer_handle, MPI_BUFFER, MPI_PERSISTENT);
MPI_GSIZE( group, &size );
MPI_RANK( group, &rank );
if (rank==0)
{
   for (i=1; i < size; i++)
      MPI_RECV(buffer_handle, i, tag, group, return_handle);
   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, return_handle);
}
MPI_FREE(buffer_handle);
```

3.5 Data move functions

Broadcast

MPI_BCAST(buffer_handle, 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 `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 group group handle

IN root rank of broadcast root (integer)

MPI_BCASTC(buf, len, group, root)

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

INOUT buffer Starting address of buffer (choice type)

IN len Number of words in buffer (integer)

IN group group handle

in root rank of broadcast root (integer)

MPI_BCAST(buffer_handle, group, root) is

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

Circular shift

MPI_CSHIFT(inbuf, outbuf, 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 group, and shift. The shift value can be positive, zero, or negative.

IN inbuf handle to input buffer descriptor

IN outbuf handle to output buffer descriptor

IN group handle to group

IN shift integer

MPI_CSHIFT1(buf, group, shift)

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

INOUT buf handle to buffer descriptor

IN group handle to group

IN shift integer

MPI_CSHIFTC(inbuf, outbuf, len, 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, 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 group handle to group

IN shift integer

MPI_CSHIFTC1(buf, len, group, shift)

Behaves like `MPI_CSHIFT1`, with buffers restricted to be blocks of numeric units. All processes make the call with the same values for `len, group`, and `shift`.

INOUT buf initial location of buffer

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

IN group handle to group

IN shift integer

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

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

End-off shift

MPI_EOSHIFT(inbuf, outbuf, group, shift)

Process with rank `i`, $\max(\texttt{0}, -\texttt{shift}) \leq \texttt{i} < \texttt{min}(\texttt{size}, \texttt{size} - \texttt{shift})$, 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 `group`, and `shift`.

IN inbuf handle to input buffer descriptor

IN outbuf handle to output buffer descriptor

IN group handle to group

IN shift integer

MPI_EOSHIFT1(buf, group, shift)

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

INOUT buf handle to buffer descriptor

IN group handle to group

IN shift integer

MPI_EOSHIFTC(inbuf, outbuf, len, 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, 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 group handle to group

IN shift integer

MPI_EOSHIFTC1(buf, len, group, shift)

Behaves like `MPI_EOSHIFT1`, with buffer restricted to be blocks of numeric units. All processes make the call with the same values for `len, group`, and `shift`.

INOUT buf initial location of buffer

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

IN group handle to group

IN shift integer

Gather

MPI_GATHER(inbuf, list_of_outbufs, group, root, return_status)

Each process (including the root process) sends the content of its input buffer to the root process. The root process places all the incoming messages in the location specified by the output buffer handle corresponding to the sender's rank. It is called by all members of group using the same arguments for `group`, and `root`. The input buffer of each process may have a different length.

IN inbuf handle to input buffer descriptor

IN list_of_outbufs list of buffer descriptor handles (root)

IN group group handle

IN root rank of receiving process (integer)

OUT return_status return status handle

MPI_GATHERC(inbuf, inlen, outbuf, group, root)

MPI_GATHERC 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, 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 group group handle

IN root rank of receiving process (integer)

MPI_GATHERC(inbuf, inlen, outbuf, group, root) is

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

Scatter

MPI_SCATTER(list_of_inbufs, outbuf, group, root, return_status)

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 routine is called by all members of the group using the same arguments for group, and root.

IN list_of_inbufs list of buffer descriptor handles

IN outbuf buffer descriptor handle

IN group handle

IN root rank of sending process (integer)

OUT return_status return status handle

MPI_SCATTER(list_of_inbufs, outbuf, group, root, return_status) is

```
MPI_GSIZE( group, &size );
MPI_RANK( group, &rank );
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_SCATTERC(inbuf, outbuf, len, group, root)

MPI_SCATTERC 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 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_SCATTERC(inbuf, outbuf, outlen, group, root) is

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

All-to-all scatter

MPI_ALLSCATTER(list_of_inbufs, list_of_outbufs, group, return_status)

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 places the incoming messages in the location specified by output buffer handle corresponding to the rank of the sender. The routine is called by all members of the group using the same arguments for group.

IN list_of_inbufs list of buffer descriptor handles

IN list_of_outbufs list of buffer descriptor handles

IN group handle

OUT return_status return status handle

MPI_ALLSCATTERC(inbuf, outbuf, len, group)

`MPI_ALLSCATTERC` 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 `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 group handle

`MPI_ALLSCATTERC( inbuf, outbuf, len, group)` is

```
MPI_GSIZE( group, &size );
MPI_RANK( group, &rank );
for (i=0; i < rank; i++)
   {
    MPI_IRECVC(recv_handles[i], outbuf, len, tag, group);
    outbuf += len;
   }
for (i=0; i < size; i++)
   {
    MPI_ISENDC(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, list_of_outbufs, group, return_status)

Each process in the group broadcasts its input buffer to all processes (including itself); each process places the incoming messages in the location specified by output buffer handle corresponding to the rank of the sender. The routine is called by all members of the group using the same arguments for `group`.

IN inbuf buffer descriptor handle for input buffer

IN list_of_outbufs list of buffer descriptor handles

IN group handle

OUT return_status return status handle

MPI_ALLCASTC(inbuf, outbuf, len, group)

MPI_ALLCASTC 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. 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 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 group handle

MPI_ALLCASTC(inbuf, outbuf, len, group) is

```
MPI_GSIZE( group, &size );
MPI_RANK( group, &rank );
for (i=0; i < rank; i++)
   {
    MPI_IRECVC(recv_handles[i], outbuf, len, tag, group);
    outbuf += len;
   }
for (i=0; i < size; i++)
   {
    MPI_ISENDC(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, 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 `group`, `root` and `op`.

IN inbuf handle to input buffer

IN outbuf handle to output buffer – significant only at root

IN group handle to group

IN root rank of root process (integer)

IN op operation (status)

We list below the operations which 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_REDUCEC(inbuf, outbuf, len, 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 group handle to group

IN root rank of root process (integer)

IN op operation (status)

MPI_USER_REDUCE(inbuf, outbuf, 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 value of the function, and the types of all entries in the input and output buffers all agree. The output buffer has the same length as the input buffer.

IN inbuf handle to input buffer

IN outbuf handle to output buffer – significant only at root

IN group handle to group

IN root rank of root process (integer)

IN function user provided function

MPI_USER_REDUCEC(inbuf, outbuf, len, 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 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.) These would be respectively:

MPI_GOP(inbuf, outbuf, group, op)

MPI_GOPC(inbuf, outbuf, len, group, op)

MPI_USER_GOP(inbuf, outbuf, group, function)

MPI_USER_GOPC(inbuf, outbuf, len, group, function)

Do we want a user provided *function* (two IN parameters, one OUT value), or a user provided procedure that overwrites the second input (ie. one IN param, one INOUT param, the equivalent of C `a op= b` type assignment)? The second choice may allow a more efficient implementation, without changing the semantics of the MPI call.

Scan

MPI_SCAN(inbuf, outbuf, 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

IN outbuf handle to output buffer

IN group handle to group

IN op operation (status)

MPI_SCANC(inbuf, outbuf, len, 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 group handle to group

IN op operation (status)

MPI_USER_SCAN(inbuf, outbuf, 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

IN outbuf handle to output buffer

IN group handle to group

IN function user provided function

MPI_USER_SCANC(inbuf, outbuf, len, 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 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.)

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`, `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 `group` and, where relevant, `root` or `op` parameters. Is this acceptable?

A few examples:

```
MPI_BCAST(buf, len, group, 0);
MPI_BCAST(buf, len, 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, group, 0);
MPI_BCAST(buf, len, 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_BCASTC(buf, len, group, 0);
   MPI_SENDC(buf, len, 2, tag, group);
  }
elseif (rank==1)
  {
   MPI_RECVC(buf, len, MPI_DONTCARE, tag, group);
   MPI_BCASTC(buf, len, group, 0);
```

```
   MPI_RECVC(buf, len, MPI_DONTCARE, tag, group);
  }
else
  {
   MPI_SENDC(buf, len, 2, tag, group);
   MPI_BCASTC(buf, len, 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.

```
    0                        1                      2

                / - >  receive            / - send
               /                         /
broadcast    /        broadcast       /   broadcast
            /                        /
  send   -             receive  < -
```

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.

Chapter 4

Process Topologies

Rolf Hempel

4.1 Introduction

The MPI1 draft standard for a user–level message–passing interface [6] 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 [5, 4].

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.

Chapter 5

Language Binding

Chapter 6

Formal Semantics

Chapter 7

Environmental Inquiry

Chapter 8

Profiling

James Cownie

8.1 Requirements

To meet the MPI profiling interface, an implementation of the MPI functions *must*

1. provide a mechanism through which all of the MPI defined functions may be accessed with a name shift (MPI suggests the prefix "**P_**", but any prefix may be used provided that it is documented).

2. ensure that those MPI functions which are not replaced may still be linked into an executable image without causing name clashes.

3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economise by implementing it only for the lowest level routines.

8.2 Discussion

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code which implements MPI on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

We believe that having such an interface is important if MPI is to be attractive to end users, since the availability of many different tools will be a significant factor in attracting users to the MPI standard.

The profiling interface is just that, an interface. It says *nothing* about the way in which it is used. There is therefore no attempt to lay down what information is collected through the interface, or how the collected information is saved, filtered, or displayed.

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as "internetworking" multiple MPI implementations. Since all that is defined is an interface, there is no objection to its being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which executable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the MPI profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this chapter is only present as justification and discussion of the logic for those requirements.

The examples below show one way in which an implementation could be constructed to meet the requirements on a Unix system (there are doubtless others which would be equally valid).

8.3 Logic of the design

Provided that an MPI implementation meets the requirements above, it is possible for the implementor of the profiling system to intercept all of the MPI calls which are made by the user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

8.4 Examples

8.4.1 Profiler implementation

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

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

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

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

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

   return result;
}
```

8.4.2 MPI library implementation

On a Unix system, in which the MPI library is implemented in C, then there are various possible options, of which two of the most obvious are presented here. Which is better depends on whether the linker and compiler support weak symbols.

Systems with weak symbols

If the compiler and linker support weak external symbols (e.g. Solaris 2.x, other system V.4 machines), then only a single library is required through the use of `#pragma weak` thus :–

```
#pragma weak MPI_EXAMPLE = P_MPI_EXAMPLE

int P_MPI_EXAMPLE(/* appropriate args */)
{
    /* Useful content */
}
```

The effect of this `#pragma` is to define the external symbol `MPI_EXAMPLE` as a weak definition, this means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library), however if no other definition exists, then the linker will use the weak definition.

Systems without weak symbols

In the absence of weak symbols then 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)(/* appropriate args */)
{
    /* Useful content */
}
```

The same source file can then be compiled to produce both versions of the library, depending on the state of the `PROFILELIB` macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions which she wishes to intercept, references to any others being fulfilled by the normal MPI library. Therefore the link step can look something like this :–

```
% cc ... -lmyprof -lpmpi -lmpi
```

Here `libmyprof.a` contains the profiler functions which intercept some of the MPI functions. `libpmpi.a` contains the "name shifted" MPI functions, and `libmpi.a` contains the normal definitions of the MPI functions.

8.5 Outstanding issues

1. 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.

Chapter 9

Contexts – Proposal I

Marc Snir

9.1 Contexts

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.

9.1.1 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.

9.1.2 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.

9.1.3 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.

9.1.4 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.

9.2 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.

9.2.1 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.

Chapter 10

Contexts – Proposal VII

Lyndon J Clarke & Rik J Littlefield

10.1 Introduction

This chapter is similar in basic principles to Proposal I and includes all of the functionality of that proposal as a subset — it extends in several ways and differs in some details. Certain features of other, now defunct, proposals discussed in the context subcommittee are included. In particular, this chapter proposes that:

1. Contexts and groups are not identical. A context is always associated with one group, but a group may have several contexts. Properties of groups are inherited by all of the associated contexts, for example process rank.

2. Context and group descriptors can be explicitly transferred to processes that are not members of the context or group.

3. In point-to-point messages, processes can be identified in any of three ways: by process, by rank in a shared context, or by ranks in separate sender and receiver contexts.

4. A "cache" facility is provided that allows modules to attach arbitrary information to both contexts and groups.

These extensions are somewhat independent of each other. The first reflects the observation that multiple modules often operate within each process group, so that context formation should be lighter weight than group formation. The second and third together provide expressive support for communication between modules within different groups of processes. The fourth allows modules to be significantly faster in common cases, without complicating their interface to the application.

Much of this proposal must be viewed as recommendations to other subcommittees of MPI, primarily the point-to-point communication subcommittee and the collective communications subcommittee. Concrete syntax is given in the style of the ANSI C host language, only for purposes of discussion.

10.2 Processes

This proposal views processes in the familiar way, as one thinks of processes in Unix or NX for example. Each process is a distinct space of instructions and data. Each process is allowed to compose multiple concurrent threads and MPI does not distinguish such threads.

Process Identifier

Each process is identified by a process-local *process handle*, which is a reference to a *process descriptor* of undefined size and opaque structure. In a static process model process handles can be obtained by mapping from a group (or context) and rank. In a future extension for dynamic processes, handles may be returned by process creation functions.

MPI provides a procedure which returns a handle for the calling process.

```
process = mpi_my_process()
```

Process Creation & Destruction

This proposal makes no statements regarding creation and destruction of processes.

MPI provides facilities for descriptor transmission allowing the user to explicitly transfer a process decriptor from one process to another. These facilities are described below.

10.3 Process Groups

This proposal views a process group as an ordered collection of (references to) distinct processes, the membership and ordering of which does not change over the lifetime of the group. The canonical representation of a group is a one-to-one map from the integers $(0, 1, \ldots, N-1)$ to handles of the N processes composing the group.

There may be structure associated with a process group defined by a process topology. This proposal makes no further statements regarding such structures.

Group Identifier

Each group is identified by a process-local *group handle*, which is a reference to a *group descriptor* of undefined size and opaque structure.

The initialization of MPI makes each process a member of the "initial" group. MPI provides a procedure that returns a handle to this group.

```
group = mpi_initial_group()
```

MPI provides facilities for descriptor transmission allowing the user to explicitly transfer a group descriptor from one process to another.

Group Creation and Deletion

MPI provides facilities which allow users to dynamically create and delete process groups. The procedures described here generate groups which are static in membership.

MPI provides a procedure which allows users to create one or more groups which are subsets of existing groups.

```
groupb = mpi_group_partition(groupa, key)
```

This procedure creates one or more new groups `groupb` which are distinct subsets of an existing group `groupa` according to the supplied values of `key`. This procedure is called by and synchronises all members of `groupa`.

MPI provides a procedure which allows users to create a group by permutation of an existing group.

```
groupb = mpi_group_permutation(groupa, rank)
```

This procedure creates one new group with the same membership as `groupa` with a permutation of process ranking, and returns the created group descriptor in `groupb`. It is called by and synchronises all members of `groupa`.

MPI provides a procedure which allows users to create a group by explicit definition of its membership as a list of process handles.

```
group = mpi_group_definition(listofprocess)
```

This procedure creates one new group `group` with membership and ordering described by the process handle list `listofprocess`. It is called by and synchronises all processes identified in `listofprocess`.

MPI provides a procedure which allows users to delete user created groups.

```
mpi_group_deletion(group)
```

This procedure deletes an existing group `group`. It is called by and synchronises all members of `group`.

MPI may provide additional procedures which allow users to construct process groups with a process group topology.

Group Attributes

MPI provides a procedure which accepts a valid group handle and returns the rank of the calling process within the identified group.

```
rank = mpi_group_rank(group)
```

MPI provides a procedure which accepts a valid group handle and returns the number of members, or *size*, of the identified group.

```
size = mpi_group_size(group)
```

MPI provides a procedure which accepts a valid group handle and process order number, or *rank*, and returns the valid process handle to which the supplied rank maps within the identified group.

```
process = mpi_group_process(group, rank)
```

MPI may provide additional procedures which allow users to determine the process group topology attributes.

MPI provides a group descriptor cache facility which allows the user to attach attributes to group descriptors.

10.4 Communication Contexts

This proposal views a communication context as the combination of a process group and a protection mechanism that avoids collision between messages sent to different contexts. The context inherits process ranking from its associated group, referred to as a *frame*. Each process group may be used as a frame for multiple contexts.

Context Identifier

Each context is identified by a process-local *context handle*, which is a reference to a *context descriptor* of undefined size and opaque structure.

The creation of a process group allocates a *base context* which inherits the created group as a frame and can be thought of as an attribute of the created group. MPI provides a procedure which accepts a valid group handle and returns a handle to the base context within the identified group.

```
context = mpi_base_context(group)
```

MPI provides facilities for descriptor transmission allowing the user to explicitly transfer a context descriptor from one process to another.

Context Creation and Deletion

MPI provides facilities which allows user to dynamically create and delete contexts in addition to the base context associated with a process group. Contexts created in this fashion can be thought of as copies of the base context of the process group.

MPI provides a procedure which allows users to create contexts. This procedure accepts the handle of a group of which the calling process is a member, and returns a handle to the new context.

```
context = mpi_context_creation(group).
```

This procedure must be called loosely synchronously by all members of `group`. The procedure may not actually synchronize the member processes — it is suggested that this is a lightweight procedure that can be implemented so as to not require interprocess communication.

MPI provides a procedure which allows users to delete user created contexts. The procedure accepts a context handle that was created by the calling process and deletes the identified context.

```
mpi_context_deletion(context)
```

This procedure has the same synchronization behavior as context creation.

Context Attributes

MPI provides a procedure which allows users to determine the process group that is the frame of a context.

```
group = mpi_context_frame(context)
```

MPI provides a group descriptor cache facility which allows user to attach attributes to group descriptors.

10.5 Descriptor Facilities

This section describes the descriptor transmission and user cache facilities.

Transmission Facility

MPI provides a mechanism whereby the user can transmit a valid descriptor in a message such that the received descriptor handle is valid. This can be integrated with the capability to transmit typed messages, and it is suggested that a notional data type should be introduced for this purpose, e.g. `MPI_DSCR_TYPE`. There are other reasonable approaches to providing this facility.

The descriptor is translated as necessary to be meaningful on the destination process, storage is allocated for it, and a handle to that storage is returned. Decorations are not transmitted.

Handles are guaranteed to be unique within each process — if processes A and B independently send to process C a descriptor for an object D, then process C will get two copies of the same handle. As with all transfers of descriptors, the receiving process is responsible for releasing the descriptor and its handle when it is no longer needed or becomes stale. MPI provides a procedure which frees a descriptor.

```
mpi_free_dscr(handle)
```

A descriptor registry service which allows descriptors to be identified by name would be a useful additional feature. This service can be implemented at the user level using the point-to-point chapter of MPI and the descriptor transmission facilities. These services can be deferred in this session of MPI.

Cache Facility

MPI provides a "cache" facility that allows an application to attach arbitrary pieces of information, called *decorations*, to context and group descriptors. Decorations are local to the process and are not included if the descriptor is sent to another process. This facility is intended to support optimizations such as saving persistent communication handles and recording topology-based decisions by adaptive algorithms.

MPI provides the following services related to cacheing:

Generate key: Generate cache key.

```
keyval = mpi_GetDecorationKey()
```

Store decoration: Store decoration in cache by key.

```
mpi_SetDecoration(handle, keyval, decoration_val,
decoration_destructor_routine)
```

Retrieve decoration: Retrieve decoration from cache by key.

```
mpi_TestDecoration(handle,keyval,decoration)
```

Delete decoration: Delete decoration from cache by key.

```
mpi_DeleteDecoration(handle,keyval)
```

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

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

   efficient_global_op (context, ...)  int context_handle; { struct
gop_stuff_type *gop_stuff; /* whatever we need */ int group_handle =
mpi_context_frame(context_handle);

     if (!gop_key_assigned) /* get a key on first call ever */ {
gop_key_assigned = 1; if ( ! (gop_key = MPI_GetDecorationKey()) ) {
MPI_abort ("Insufficient keys available"); } } if (MPI_TestDecoration
(group_handle,gop_key,&gop_stuff)) { /* This module has executed in
this group before.  We will use the cached information */ } else { /*
This is a group that we have not yet cached anything in.  We will now
do so.  */ gop_stuff = /* malloc a gop_stuff_type */

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

       MPI_SetDecoration (group_handle, gop_key, gop_stuff,
gop_stuff_destructor); } /* ... use contents of *gop_stuff to do the
global op ... */ }

    gop_stuff_destructor (gop_stuff) /* called by MPI on group delete
*/ struct gop_stuff_type *gop_stuff; { /* ... free storage pointed to
by gop_stuff ... */ }
```

The cache facility could also be provided for process descriptors, but it is less clear how such provision would be useful. It is suggested that the cache store, retrieve and delete decoration procedures should fail when applied to a process descriptor handle.

10.6 Point-to-Point Communication

This proposal recommends three forms for MPI point-to-point message addressing and selection: null context; closed context; open context. It is further recommended that messages communicated in each form are distinguished such that a `Send` operation of form X cannot match with a `Receive` operation of form Y, requiring that form is embedded into the message envelope.

The three forms are described, followed by considerations of uniform integration of these forms in the point-to-point communication chapter of MPI.

Null Context Form

The *null context* form contains no message context. Message selection and addressing are expressed by

```
(process, tag)
```

where: `process` is a process handle; `tag` is a message tag.

`Send` supplies the `process` of the receiver. `Receive` supplies the `process` of the sender.

`Receive` can wildcard on `process` by supplying the wildcard descriptor handle value `MPI_WILDCARD`. In this case the receiver may have obtained the process descriptor of the sender, and the null descriptor handle `MPI_NULL` is returned in the relevant point-to-point enquiry procedure.

Closed Context Form

The *closed context* form permits communication between members of the same context. Message selection and addressing are expressed by

```
(context, rank, tag)
```

where: `context` is a context handle; `rank` is a process rank in the frame of `context`; `tag` is a message tag. The calling process must be a member of the frame of `context`.

`Send` supplies the `context` of the receiver (and sender), and the `rank` of the receiver. `Receive` supplies the `context` of the sender (and receiver), and the rank of the sender. The `(context, rank)` pair in `Send` (`Receive`) is sufficient to determine the process identifier of the receiver (sender).

`Receive` cannot wildcard on `context`. `Receive` can wildcard on `rank` by supplying the wildcard integer `MPI_DONTCARE`. This proposal makes no statement about the provision for wildcard on `tag`.

Open Context Form

The *open context* form permits communication between members of any two contexts. Message selection and addressing are expressed by

```
(lcontext, rcontext, rank, tag)
```

where: `lcontext` is a context handle; `rcontext` is a context handle; `rank` is a process rank in the frame of `rcontext`; `tag` is a message tag. The calling process must be a member of the frame of `lcontext` and need not be a member of the frame of `rcontext`.

`Send` supplies the context of the sender in `lcontext`, the context of the receiver in `rcontext`, and the `rank` of the receiver in the frame of `rcontext`. `Receive` supplies the context of the receiver in `lcontext`, the context of the sender in `rcontext`, and the `rank` of the sender in the frame of `rcontext`. The `(rcontext, rank)` pair in `Send` (`Receive`) is sufficient to determine the process identifier of the receiver (sender).

`Receive` cannot wildcard on `lcontext`. `Receive` can wildcard on `rcontext` by supplying the wildcard descriptor handle value `MPI_WILDCARD`, in which case it must also wildcard on `rank` since the process descriptor of the sender cannot be determined. In this case the receiver may not have obtained the context descriptor of the sender, and the null descriptor handle `MPI_NULL` is returned in the relevant point-to-point enquiry procedure. `Receive` can wildcard on `rank` by supplying the wildard intger value `MPI_DONTCARE`.

Uniform Integration

The three forms of addressing and selection described have different syntactic frameworks. We can consider integrating these forms into the point-to-point chapter of MPI by defining a further orthogonal axis (as in the multi-level proposal of Gropp & Lusk) which deals with form. This is at the expense of multiplying the number of `Send` and `Receive` procedures by a factor of three, and some further but trivial work with details of the current point-to-point chapter which uniformly assumes a single addressing and selection form.

There are various approaches to unification of the syntactic frameworks which may simplify integration. Two options are now described, each based on retention and extension of the framework of the closed and open contexts forms.

The framework of the open context form could be adopted and extended. The null context form is expressed as `(MPI_NULL, MPI_NULL, process, tag)`, which is a little clumsy. The closed context form is expressed as `(MPI_NULL, context, rank, tag)`, which is marginally inconvenient. The open context form is expressed as `(lcontext, rcontext, rank, tag)`, which is of course natural.

The framework of the closed context form could be adopted and extended. The null context form is expressed as `(MPI_NULL, process, tag)`, which is marginally inconvenient, and requires that descriptor handles are expressed as intgers. The closed context form is expressed as `(context, rank, tag)`, which is of course natural. Expression of the open context form requires a little more work. We can use the `context` field as "shorthand notation" for the `(lcontext, rcontext)` pair at the expense of introducing some trickery. We define a "duplet descriptor" which is formally composed of two references to contexts, and provide a procedure which constructs such a descriptor given two context descriptors. Both `Send` and `Receive` accept a duplet descriptor in `context`, are able to distinguish the duplet descriptor from a singlet descriptor, and treat the duplet as shorthand notation. It is conjectured that using this framework is the best choice for MPI.

10.7 Collective Communication

Symmetric collective communication operations are compliant with the closed context form described above. This proposal recommends that such operations accept a context descriptor which identifies the context (thus frame) in which they are to operate.

MPI does plan to describe symmetric collective communication operations. It is not possible to determine whether this proposal is sufficient to allow implementation of the collective communication chapter of MPI in terms of the point-to-point chapter of MPI without loss of generality, since the collective operations are not yet defined.

Asymmetric collective communication operations, especially those in which sender(s) and receiver(s) are distinct processes, are compliant with the open context form described above. This proposal recommends that such operations accept a pair of context descriptors (a duplet descriptor) which identify the contexts (thus frames) in which they are to operate.

MPI does not plan to describe asymmetric collective communication operations. Such operations are expressive when writing programs beyond the SPMD model, which are composed of communicative functionally distinct process groups. These services can be deferred in this session of `mpi`.

10.8 Conclusion

This chapter presented a proposal for communication contexts and process groups with MPI. In the proposal process groups are created dynamically and are static in membership.

Associated with each process group are one or more communication contexts which inherit process ranking.

The recommendations for point-to-point communication are powerful. The proposal provides process addressed communication which occurs within an extended context. The proposal also contains closed context communication addressed in terms of context and rank which protects messages belonging to one context from those belonging to other contexts. The proposal also contains open context communications adressed in terms of sender context, receiver context, and process rank which provides expressive power for intercommunication between modules within different groups.

The proposal is extensible to a number of features which might be included in future sessions of **MPI**, for example: dynamic processes; dynamic groups; multiple group collective communications.

Chapter 11

Contexts – Proposal III

A. Skjellum *et al.*

11.1 Introduction

This chapter takes a slightly different approach to contexts and groups, than does Proposal VII. It is of roughly equal conceptual "power" as Proposal VII, with some differences. As appropriate, this chapter borrows directly from Proposal VII, by Clarke and Littlefield.

1. Contexts are supported to discriminate between messages in the system. A context is a conceptual extension of the tag space into a system-defined part (not wildcardable), and a totally user-defined part (the traditional 32-bit tag).

2. A context is a lower-level concept than a group, so that contexts not associated with groups are permitted. This permits the user to develop codes that build on the server model, or which build up groups dynamically (not otherwise supported by MPI1).

3. Groups are used to describe cooperative communication in the system. Groups have one or more context of communication associated with them. When created, a group is given a context of communication.

4. Context and group descriptors can be explicitly transferred to processes that are not members of the context or group.

5. In point-to-point messages, processes can be identified in either of two ways: by opaque process identifier, or by rank in a group; in either case, communication scope is within a given context.

6. The cache facility, allowing groups to add additional information (described in Proposal VII) is embraced by this Proposal, with reservations as noted. The possible need to omit this cacheing feature from MPI1 should not invalidate the remainder of Proposal VII from further consideration (severability).

11.2 Processes

This Proposal views processes in the familiar way, as one thinks of processes in Unix or NX for example. Each process is a distinct space of instructions and data. Each process is allowed to compose multiple concurrent threads and MPI does not distinguish such threads. MPI shall be thread-aware, but not thread supporting, so we make every attempt to make thread safe programs possible in defining what follows.

Process Identifier

Each process is identified by an opaque process identifier, which is associable to a *process descriptor* of undefined size and opaque structure, through MPI accessor calls. In a static process model process identifiers can be obtained by mapping from a group and rank. In a future extension for dynamic processes, identifiers may be returned by process creation functions.

MPI provides a procedure that returns an identifier for the calling process.

```
my_process = mpi_my_process()
```

This identifier can be converted to a transmittable form by MPI converter functions, though opaque, it is conceptually a pointer.

Process Creation & Destruction

This proposal makes no statements regarding creation and destruction of processes.

MPI provides facilities for identifier transmission allowing the user explicitly to transfer a process identifier from one process to another. These facilities are described below. MPI also provides means to transfer underlying information about the opaque process descriptor underlying.

11.3 Process Groups

This proposal views a process group as an ordered collection of distinct processes (via process identifiers), the membership and ordering of which does not change over the lifetime of the group. The canonical representation of a group is a one-to-one map from the integers $(0, 1, \ldots, N-1)$ to identifiers of the N processes composing the group.

There may be structure associated with a process group defined by a process topology. This proposal makes no further statements regarding such structures.

There may be non-enumerative ways to construct and manipulate special groups, or for special machine architectures (*e.g.*, cohorts). This proposal makes no further statements about such special groups, other than the desirability of avoiding group-name enumeration, when possible.

Group Identifier

Each group is identified by an opaque *group identifier*, which is a associable to a *group descriptor* of undefined size and opaque structure, through MPI accessor functions.

The initialization of MPI makes each process a member of the "initial" group. MPI provides a procedure that returns an identifier to this group.

```
group_ident = mpi_initial_group()
```

MPI provides facilities for descriptor transmission allowing the user explicitly transfer a group descriptor from one process to another.

Group Creation and Deletion

MPI provides facilities which allow users dynamically to create and delete process groups. The procedures described here generate groups which are static in membership.

MPI provides a procedure that allows users to create one or more groups which are subsets of existing groups.

```
new_group = mpi_group_partition(old_group, key)
```

This procedure creates one or more new groups `new_group` which are distinct subsets of an existing group `old_group` according to the supplied values of `key`. This procedure is called by and synchronises all members of `new_group`. No overlapping is permitted, so that exactly one `new_group` is achieved in each `old_group` process. The new groups have new contexts of communication. The number of new contexts depends on the number of different key values asserted.

MPI provides a procedure which allows users to create a group by permutation of an existing group.

```
new_group = mpi_group_permutation(old_group, rank)
```

This procedure creates one new group with the same membership as `old_group` with a permutation of process ranking, and returns the created group descriptor in `new_group`. It is called by and synchronises all members of `gha`. The new group has a new context of communication.

MPI provides a procedure that allows users to create a group by explicit definition of its membership as a list of process identifiers.

```
    new_group = mpi_group_definition(array_of_process_ids,length,
context_to_use)
```

This procedure creates one new group `new_group` with membership and ordering described by the process handle list `array_of_process_ids`, of length `length`. If `context_to_use` is specified as the special context `MPI_GET_CONTEXT`, then the system allocates the new context. Else, the system trusts the user to have allocated the context legally (see below). This procedure must be called by and synchronises all processes identified in the list. A further approach to new group definition is as follows

```
    new_group = mpi_group_def_by_leader(leader_id, in_length,
in_array_of_process_ids, context_to_use, out_array_of_process_ids,
out_length)
```

This weaker form requires all future group members to have identified only a leader (specified by `leader_id`). The leader knows the length a names of all participants. This is a synchronization of all participants. Same semantics for `context_to_use` as above.

MPI provides a way formally to duplicate a group, in order to obtain a separate context of communication. This can be achieved by using other operations, but this procedure allows optimization in some implementations (no explicit group copy, for instance).

```
new_group = mpi_group_duplicate(old_group)
```

The only difference between the old and new groups is that there is a new context of communication.

MPI provides a procedure which allows users to delete user created groups.

```
mpi_group_deletion(group)
```

This procedure deletes an existing group `group`. It is called by and synchronises all members of `group`.

MPI may provide additional procedures which allow users to construct process groups with a process group topology.

Group Attributes/Accessors

MPI provides a procedure that accepts a valid group identifier and returns the rank of the calling process within the identified group.

```
rank = mpi_group_rank(group)
```

MPI provides a procedure that accepts a valid group identifier and returns the number of members, or *size*, of the identified group.

```
size = mpi_group_size(group)
```

MPI provides a procedure that accepts a valid group identifier *rank-in-group*, and returns the valid process identifier to which the supplied rank maps within the identified group.

```
process_id = mpi_group_process(group, rank)
```

MPI may provide additional procedures which allow users to determine the process group topology attributes.

MPI provides a group descriptor cache facility thath allows the user to attach attributes to group descriptors. See Proposal VII for details.

11.4 Communication Contexts

This proposal views a communication context as a partition of the tag space, which is a protection mechanism that avoids collision between messages sent between processes. Process groups have one or more contexts in *MPI*. Unlike Proposal VII, more contexts are obtained for a group using the above-discussed group creation and replication functions. Replication may only be formal for good implementations.

Context Identifier

Each context is identified by an opaque process identifier. It is conceptually an integer assigned by the system to partition a large tag space into a user-defined and system-controlled subspaces. This stategy provides the minimal level of isolation needed to build large libraries, and is close to practice.

Context Creation and Deletion

MPI provides facilities that allow user dynamically to allocate and free contexts. When contexts are used with groups, these calls are not needed. For more advanced users (such as building your own dynamic groups), these calls will be used. Above, where `context_to_use` appears as an argument, the following call would have been used to secure such a context in advance.

```
    mpi_context_creation(number_of_contexts_wanted,array_of_contexts,
number_of_contexts_provided)
```

This call is called by any process, with no synchronization to other processes.

MPI provides a procedure that allows users to delete user- created contexts. The procedure accepts a context identifier array, containing zero or more contexts created previously in the system.

```
    mpi_context_deletion(context_array,length)
```

No synchronization occurs here. The user can do erroneous things by freeing contexts that are still in use.

For general applications, it may be nice to have a name service for contexts (necessary for building dynamic groups and servers, for yourself). Herewith:

```
    mpi_associate_contexts_with_name(string_name,context_array,length)
mpi_disassociate_contexts_with_name(string_name)
mpi_get_contexts_by_name(string_name,max_length,out_length,
context_array
```

As with context generation, the above calls assume a simple reactive, global server, or shared name space mechanism (both achieveable easily in practice).

11.5 Descriptor Facilities

This section describes the descriptor transmission and user cache facilities.

Conversion Facility

MPI provides a mechanism whereby the user can convert a valid descriptor (*e.g.*, a group descriptor) idenfied through an identifier (*e.g.*, a group identifier) for use in a message such that the received descriptor can be reconstructed on the remote end. This can be integrated with message transmission as the user sees fit, without additional complication to the send/receive semantics of MPI. An example follows:

```
    error = mpi_group_group_transmit(group, group_buffer, max_length,
act_length)
```

If the buffer is not long enough to hold the information, an error occurs. A network independent format can be assumed in the `group_buffer`. Cached “attributes” are not transmitted (see below).

Cache Facility

MPI provides a “cache” facility that allows an application to attach arbitrary pieces of information, called *attributes*, to context and group descriptors. Attributes are local to the process and are not included if the descriptor is sent to another process. This facility is intended to support optimizations such as saving persistent communication handles and recording topology-based decisions by adaptive algorithms.

MPI provides the following services related to cacheing. We call our attributes ‘attributes’; Proposal VII calls them (equivalently) decorations (no big difference, except naming, is anticipated).

Generate key: Generate cache key.

```
keyval = mpi_get_attribute_key()
```

Store attribute: Store attribute in cache by key.

```
mpi_set_attribute(handle, keyval, attribute_val,
attribute_destructor_routine)
```

Retrieve attribute: Retrieve attribute from cache by key.

```
mpi_Test_Attribute(handle,keyval,attribute)
```

Delete attribute: Delete attribute from cache by key.

```
mpi_delete_attribute(handle,keyval)
```

Each attribute consists of a pointer or a value of the same size as a pointer, and would typically be a reference to a larger block of storage managed by the module. Our example will appear in a later draft, because we have semantic differences from some of the ancillary aspects of the example of Proposal VII.

The cache facility could also be provided for process identifiers, but it is less clear how such provision would be useful. It is suggested that the cache store, retrieve and delete attribute procedures should fail when applied to a process identifiers.

Implementations should use AVL trees, or similar efficient data structures to provide relatively efficient access to attributes.

11.6 Point-to-Point Communication

This proposal recommends two forms for MPI point-to-point message addressing and selection: by-group notation, by process-ID notation; always, one is working in a context, as this is the fundamental management tactic of MPI for messages. As a group always has a context at creation, and an ALL group is anticipated, this should prove fine for a static process model. We disagree significantly from Proposal VII in what follows.

The two forms are described, followed by considerations of uniform integration of these forms in the point-to-point communication chapter of MPI.

Group-Rank Form

The *group-rank* form permits communication between members of the same context and group. Message selection and addressing are expressed by

`(group, rank, tag)`

where: `group` is a group identifier; `rank` is a process rank in that group; `tag` is a message tag. The calling process must be a member of the frame of `context`.

`Send` determines the context using information in the group identifier. It does all necessary mappings to the process identifier space. `Receive` cannot wildcard on context, so a valid matching receive must refer to the same group information. `Receive` can wildcard on `rank` by supplying the wildcard integer `MPI_DONTCARE`. This proposal makes the following statement about the provision for wildcard on `tag`. Two integer-form wildcard is needed for layering: `care_bits`, `dont_care_bits`. Tags are matched if and only if

$$(received\_tag AND NOT dont\_care_bits) XOR care\_bits == 0 \tag{11.1}$$

This general format can be used to partition the tag space for virtual topologies or other user-defined needs, and is quite important to the standard's flexibility.

Process-Identifier Form

Communication takes place using the following parameters:

```
(context, process_identifier, tag)
```

where: `context` is a context identifier, `process_identifier` is a process identifier, `tag` is a message tag. The calling process must be a member of the same context as the recipient. There is no reference to groups here. Contexts can have been shared by using a least common ancestor prior to this call, or by the above-mentioned context naming service.

There is never wildcarding on context. Wildcarding on `process_identifier` is through `MPI_DONTCARE`. Tag wildcarding is through the integer pair described above.

Uniform Integration

The two forms of addressing and selection described have different syntactic frameworks. We can consider integrating these forms into the point-to-point chapter of MPI by defining a further orthogonal axis (as in the multi-level proposal of Gropp & Lusk) which deals with form. This is at the expense of multiplying the number of `Send` and `Receive` procedures by a factor of two, and some further but trivial work with details of the current point-to-point chapter which uniformly assumes a single addressing and selection form. No further details, other than naming that disambiguates the rank-group form from the process-id-context form is really needed, and the naming would seem uncontroversial.

11.7 Collective Communication

Symmetric collective communication operations are compliant with the group-rank form described above. This proposal recommends that such operations accept a group-identifier (which contains context and other information) needed to operate correctly. We recommend that the tag argument be included in collective calls where this could help with debugging.

MPI does plan to describe symmetric collective communication operations. It is impossible to determine whether this proposal is sufficient to allow implementation of the collective communication chapter of MPI in terms of the point-to-point chapter of MPI without loss of generality, since the collective operations are not yet defined.

Asymmetric collective communication operations, especially those in which sender(s) and receiver(s) are distinct processes, should be made compliant with the group-rank form described above.

MPI1 should forego non-blocking collective operations, but ask vendors to support thread models in lieu of such operations.

11.8 Conclusion

This proposal is substantially different than either Proposal VII or I. Contexts are integer tag partitions here, and are fundamentally lower-level objects than groups. Groups have contexts in this proposal, but not all operations require the presence of group-scope. To avoid invidious comparisons here, more substantial comparisons of all three proposals are deferred the following appendix.

Chapter 12

Summary of context subcommittee proposals

The three proposals in the context subcommittee share common features, and have differences both in concept and detail. Two of these proposals contain features which are "separable" and could equally appear as components of one or more other proposals. This summary identifies feature of proposals as: Common Features; Separable Features; Concept Differences; Detail Differences.

Hopefully the summary will: (a) help us to discuss the important differnces between the proposals and make agreements on how we should proceed with respect to those issues; (b) help us to isolate the separable points and make separate agreements on those issues.

I hope that the summary is both accurate and complete. Please make corrections and additions if you discover such. I apologise in advance for my errors, which are surely inevitable.

12.1 Common Features

12.1.1 Process group management

In each proposal groups are created dynamically and have static membership. In each proposal a group can be created as a partition of an existing group and as a permutation of an existing group. In each proposal there is a defined group containing all (or perhaps all initial) processes. Each proposal allows (or suggests) that a group can be created as an explicit list of processes.

12.1.2 Provision for point-to-point communication within group

In each proposal point-to-point communcation of scope closed within a group can be expressed in terms of a reference to a group coupled with a process rank within the group.

12.1.3 Provision for collective communication within group

In each proposal collective communication of scope closed within a group can be expressed in terms of a reference to a group.

12.1.4 Opacity of group and process description

In each proposal the description of groups and processes is opaque. Groups and processes are referred to by a handle like object.

12.1.5 Fields of point-to-point communication

In each proposal point-to-point communication accepts three fields, inclusive of message tag, in addressing and selection.

12.2 Separable Features

12.2.1 Tag usage in point-to-point communication

Proposal III describes tag selection for Receive in a two-integer form. Proposals I and VII say nothing about tag usage.

This feature can be placed in all Proposals I, III and VII.

12.2.2 Tag usage in collective communication

Proposal III suggests that tag should be used as an argument to collective communication where this will assist debugging. Proposals I and VII say nothing about tag usage.

This feature can be placed in all Proposals I, III and VII.

12.2.3 Context or Group cache

Proposal VII describes a ”cache” facility associated with contexts and groups. Proposal III describes a similar ”cache” facility associated with groups.

This feature can be placed in all Proposals I, III and VII.

12.2.4 Opaque object (descriptor) transmission

Proposal VII suggests that opaque object transmission can be provided by integration with transmission of typed data. Proposal III suggests that opaque transmission is provided by a mechanism for flattening a descriptor into a memory buffer. These are details of different ways of providing the feature.

This feature can be placed in Proposals III and VII. This feature cannot be placed in Proposal I.

12.2.5 Context registry

Proposal III describes a context name registry service. Proposal VII indicates that such a service would be useful.

This feature can be placed in Proposals III and VII. This feature cannot be placed in Proposal I.

12.3 Concept Differences

12.3.1 Concept of CONTEXT and GROUP

In Proposal I CONTEXT and GROUP are identical concepts and are not distinguished.

In Proposal III CONTEXT is a lower degree concept than GROUP. The GROUP concept inherits aspects of the CONTEXT concept.

In Proposal VII CONTEXT is a higher concept than GROUP. The CONTEXT concept inherits aspects of the GROUP concept.

12.3.2 Scope of point-to-point communication

In Proposal I the scope of point-to-point communication is limited to the group. Processes which are members of distinct groups can only communicate through a common ancestor group.

In Proposals III and VII the scope of point-to-point communication is not limited. Processes which are members of distinct groups can communicate without reference to a common ancestor group.

12.3.3 Transmission of group or context

In Proposal I the CONTEXT cannot be transmitted from one process to another.

In Proposals VII and III both CONTEXT and GROUP can be transmitted from one process to another. In Proposal VII PROCESS can alo be transmitted (Proposal III suggests such but makes no specific provision, presumably a small oversight?)

12.4 Detail differences

12.4.1 Manifestation of context

In Proposals I and VII context is an opaque object.

In Proposal III context is an integer.

12.4.2 Deletion of group

In Proposals VII and III groups can be deleted.

In Proposal I there is no provision for group deletion (possibly a small oversight?).

12.4.3 Duplication of group

In Proposals I and III there is explicit provision for duplication of an existing group to form a new (distinct, homomorphic) group.

In Proposal VII there is no such provision as similar funtionality is provided by the context (although the provision for group partition, permutation and definition can be used to create a snapshot copy of a group).

12.4.4 Global shared variables

Proposals I and VII do not require global shared variables.

Proposal III requires a global shared variable (which can be implemented as such or of course in the traditional approach as a global service process.)

12.4.5 Process identifier addressed communication

Proposal I does not make provision for process identifier addressed communication.

Proposal III makes provision for process identifier addressed communication within multiple distinct tag spaces.

Proposal VII makes provision for process identifier addressed communication within a single distinct tag space.

12.4.6 Inter-group communication

Proposal I does not provide inter-group communication as it limits the scope of point-to-point communication to be closed within a group.

Proposal VII provides inter-group communication in a triplet addressing form: sender (receiver) group, receiver (sender) group, sender (receiver) rank.

Proposal III provides inter-group communication as process identifier addressed communication.

Bibliography

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

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

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

[4] S. Chittor and R. J. Enbody. Performance evaluation of mesh–connected wormhole–routed networks for interprocessor communication in multicomputers. In *Proceedings of the 1990 Supercomputing Conference*, pages 647–656, 1990.

[5] S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software)*, pages II–1 – II–4, 1991.

[6] 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.

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

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

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

[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.

Appendix 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.