D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

June 21, 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 June 21, 1993.

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

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

Acknowledgments

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

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point to Point Communications
- Al Geist, Marc Snir, 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	Joe Baron	Eric Barszcz
Rob Bjornson	Anne Elster	Jim Feeney
Sam Fineberg	Jon Flower	Daniel Frye
Ian Glendinning	Adam Greenberg	Robert Harrison
Leslie Hart	Tom Haupt	Don Heller
Tom Henderson	C. T. Howard Ho	Steven Huss-Lederman
John Kapenga	Bob Leary	Arthur Maccabe
Peter Madams	Oliver McBryan	Phil McKinley
Chuck Mosher	Dan Nessett	Peter Pacheco
Paul Pierce	Sanjay Ranka	Peter Rigsbee
Arch Robison	Ambuj Singh	Alan Sussman
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 chapter is a draft of the current proposal for point-to-point communication. The MPI procedures are specified using a language independent notation. The parameters of procedure calls are marked as `IN` or `OUT`. The call does not update a parameter marked `IN` and may update a parameter marked `OUT`. If a parameter is a handle to an opaque object, and the object is updated by the procedure call, then the parameter is marked `OUT`. The Fortran 77 and C bindings are described in Chapter 5.

Discussions that are not part of the standard are indicated by 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 is described in Chapter 5.

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 calling suitable MPI procedures, and providing the handle as parameter. A handle to an opaque object is significant only at the process where the object was created, and cannot be transferred to another process (objects are local).

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 destroyed 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 object associated with the handle for destruction. The object can be destroyed as soon as there there is no pending operation that is using this object, and the handle becomes undefined. 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` – one, in fact, must complete the communication with a successful call to `MPI_WAIT` or `MPI_STATUS`. However, once the operation completes, it is erroneous to reuse the freed handle for a new communication.

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

DISCUSS

Discussion: We do not have yet a mechanism to control what storage is allocated for opaque objects. Some believe it is desirable to allow the user to control this allocation.

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 is a regular array with entries of type `handle` in consecutive initial locations in the array. Whenever such array is used, an additional `len` parameter is required to indicate the number of valid entries (unless this number can be derived otherwise).

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. The alternative is for array-of-handles to be an opaque object, manipulated 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_INTEGER`, `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 wildcard value, `MPI_ANY_TAG`. Such parameters will have a range of regular values, which is a proper subrange of the range of values of the corresponding basic type; special values (such as `MPI_ANY_TAG`) will be outside the regular range. The range of regular values can be queried, and sometimes set, using environment inquiry or environment setting functions (Chapter 7).

Discussion:

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

2.2.5 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. 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 each execute their own code, in a MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each processor executes in its own address space, although shared-memory implementations of MPI are possible. This document specifies the behavior of a parallel program assuming that only MPI calls are used for communication. The interaction of an MPI program with other possible means of communication (e.g., shared memory) is not specified.

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

MPI does not specify either the interaction of processes with signals, in a Unix environment, or other events that do not relate to MPI communication. I.e., signals are not significant from the view point of MPI, and implementers should attempt to implement MPI so that signals are transparent: an MPI call suspended by a signal should resume and complete after the signal is handled.

MPI does not provide mechanisms to specify the initial allocation of processes to an MPI computation and their binding to physical processors. It is expected that vendors will provide mechanisms to do so either at load time or at run time. Such mechanisms will allow the specification of the initial number of required processes, the code to be executed by each initial process, and the allocation of processes to processors. Also, the current proposal does not provide for dynamic creation or deletion of processes during program execution (the total number of processes is fixed), although it is intended to be consistent with such 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 group, so that, effectively, processes are identified by consecutive integers.

2.3.1 Error Handling

Discussion: This section should be moved to a general introduction.

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 will provide generous limits on some of the resources so as to alleviate the portability problem this represents. One can also write **safe** programs, that are not subject to resource errors.

All MPI calls return an error parameter that indicates successful completion of the operation, or the error condition that occurred, otherwise. However, MPI does not require the user to check error parameters. It is recommended to provide an implementation where the environment for an MPI execution can be set so that any error encountered in an MPI call will cause a program to fail with a suitable error message returned to the user. The environment for an MPI execution can be also set so that (nonfatal) errors do not cause a program to fail, and the user can provide his or her own error recovery code.

MPI calls may initiate operations that continue asynchronously after the call returned. Thus, the operation may return with a code indicating successful completion, yet later cause an error exception to be raised. If there is a subsequent call that relates to the same operation (e.g., a call that verifies that an asynchronous operation has completed) then the error parameter associated with this call will be used to indicate the nature of the error. In a few cases, the error may occur after all calls that relate to the operation have completed, so that no error parameter can be used to indicate the nature of the error (e.g., an error in a send with the ready mode). Such error must be treated as fatal, since information cannot be returned for the user to recover from it.

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

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. Different machines may have different representations for the same character, or may use more than one byte to represent 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 of the sending process (integer)

destination The rank of the receiving process (integer)

tag User defined (integer)

context communication 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 Chapter 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.

Discussion: "Context" has now become "communicator", but this word is akward to use. Suggestions?

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. The syntax is designed to be extendable so that new types of buffers can added by vendors.

2.5.1 Communication buffer components

A buffer consists of a sequence of **buffer components**. Each component consists of a sequence of (not necessarily distinct) variables of the same basic datatype. The datatype of such a variable is specified by a state parameter. The possible values for Fortran are `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE`, `MPI_COMPLEX`, `MPI_DOUBLE_COMPLEX`, `MPI_LOGICAL`, `MPI_CHARACTER` and `MPI_BYTE`, corresponding 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 signed/unsigned in C. More elaborate definitions belong to the language binding.

There are five kinds of buffer components:

Contiguous component

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

start Initial element

count Number of elements (count $\geq$ 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 component

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

start Initial element

count Number of elements (count $\geq$ 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 count 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 component

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

start initial element

count Number of elements (`count` $\geq$ `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 component with entries `A(5), A(2), A(5), A(4)`.

`Vector` can be thought of as a special case of `indexed` in which the indices are generated with a constant stride.

Discussion: The above statement is not quite true, since there is no `lenblk` parameter on the indexed component. It was suggested that indexed will provide a list of (position, block_length) pairs.

Heterogeneous vector (hvector) component

Same as `vector`, except that `stride` is given in multiple of bytes, rather than multiple of elements:

start Initial element

count Number of elements (`count` $\geq$ `0`) (integer)

stride Number of **bytes** between the start of each block (integer)

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

datatype Type of elements (state)

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 `count` 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, a `double` occupies eight bytes, and no padding is needed. 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 heterogeneous data.

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

Heterogeneous indexed (hindexed) component

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

start initial element

count Number of elements (`count` $\geq$ `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 `heterogeneous vector`, `heterogeneous indexed` extends the `indexed` facility so as to allow the extraction of homogeneous data from a heterogeneous 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 different 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 arithmetics, we decided to provide separately the more advanced function to "expert programmers".

2.5.2 Buffer operations

A buffer is described by an opaque **buffer descriptor** object accessed via a **buffer handle**. Such an 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_BUFFER`. Once committed, the descriptor cannot be modified any more.

Note that the commit operation does not create yet a message, and does not bind the data to be sent. It is a commit for the buffer descriptor, not a commit for the buffer content.

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:
Function really is `create_buffer_descriptor`. May want to change the name.

MPI_APPEND_CONTIGUOUS(buffer, start, count, datatype)

Append a contigous component to buffer. The parameters are:

INOUT buffer buffer handle

IN start buffer component initial element (choice)

IN count Number of elements (integer)

IN datatype datatype identifier (status)

MPI_APPEND_VEC(buffer, start, count, 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 count 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)

Discussion:
Which order is more natural: `stride, lenblk` or `lenblk, stride`?

MPI_APPEND_INDEXED(buffer, start, count, 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 count Number of elements (integer)

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

IN datatype datatype identifier (status)

The index values used to create this component are the values provided at the call; subsequent updates of the array of indices will not affect the buffer descriptor component.

MPI_APPEND_HVEC(buffer, start, count, stride, lenblk, datatype)

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

INOUT buffer buffer handle

IN start buffer component initial element (choice)

IN count 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, count, array_of_indices, datatype)

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

INOUT buffer buffer handle

IN start initial position for indexing (choice)

IN count Number of elements (integer)

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

IN datatype datatype identifier (status)

MPI_COMMIT_BUFFER(buffer)

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.

The current proposal does not allow to create a buffer descriptor, use it in a communication after committed, then modify it partially and use it for another communication – a new buffer descriptor has to be created. Of course, the same buffer descriptor can be reused many times in successive communications that send data from the same buffers; e.g., a new send with a buffer descriptor handle can be initiated once the previous send with this handle has completed.

Example 1:
Consider the following fragment of Fortran code

```
DOUBLE PRECISION A(10,20)
INTEGER B, C(5,10), I(3)
INTEGER BH
DATA (I(J), J=1,3) /9,2,1/
...
CALL MPI_CREATE_BUFFER(BH, MPI_PERSISTENT)
CALL MPI_APPEND_CONTIGUOUS (BH, B, 1, MPI_INTEGER)
CALL MPI_APPEND_VEC (BH, A(1,3), 11, 4, 2, MPI_DOUBLE)
CALL MPI_APPEND_INDEXED(BH, C(3,7), 3, I, MPI_INTEGER)
```

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.

Example 2:
Suppose one has an array of data declared in C as

```
struct {
   int type;
   double position[3];
   double velocity[3];
       } particle[1000]
```

and one wants to transfer in one message the number of particles of type 1, the number of particles of type 2, and the positions of all the particles of these two types. The following code fragment creates a suitable buffer descriptor.

```
int count1 = 0;
int count2 = 0;
...
mpi_create_buffer( &handle, MPI_PERSISTENT);
mpi_append_contiguous( handle, (void *)&count1, MPI_INT);
```

```
mpi_append_contiguous( handle, (void *)&count2, MPI_INT);
for (i=0; i<1000; i++) {
   if (particle[i].type == 1) {
      count1++;
      mpi_append_contiguous( &handle, (void *)&particle[i].position, 3,
MPI_DOUBLE); }
   elseif (particle[i].type == 2) {
      count2++;
      mpi_append_contiguous( &handle, (void *)&particle[i].position, 3,
MPI_DOUBLE); }
   }
mpi_commit_buffer(handle)
...
```

The buffer descriptor handle can now be used to send (repeatedly) the particle counts for type 1 and 2, and the positions of the particles with these two types, as long as the types of the particles is not modified.

Suppose the incoming data is to be stored at the receiver into the two count variables `count1` and `count2` and into the `position` components of the initial entries of the array of particles. The following code generates a suitable buffer descriptor for the receive operation (we assume 4 bytes for integer and 8 bytes for double).

```
...
mpi_create_buffer( &handle, MPI_PERSISTENT);
mpi_append_contiguous( &handle, (void *)&count1, MPI_INT);
mpi_append_contiguous( handle, (void *)&count2, MPI_INT);
mpi_append_hvec( handle, (void *)&particle[0].position, 1000, 52, 3,
MPI_DOUBLE); mpi_commit_buffer( handle)
...
```

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_APPEND_xxx` call). Such a handle is associated with an empty buffer, and a message created from it contains no data. The handle still need be committed before it is used.

The same entry may appear 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.

2.5.3 Communication Buffers – Extension

TO DISCUSS

Discussion:

James Cownie proposes the following extension to buffer descriptors. Its main feature is the addition of a buffer component that allows for the efficient representation and manipulation of arrays of C structures. This extension has not been discussed by MPIF.

(The original proposal has been reprocessed by Rusty and Marc.)

Fortran and C both allow to provide hierarchical definitions of composite data structures; on the other hand, buffer descriptors are flat structures. Thus, it is inconvenient to use buffer descriptors in order to transfer records.

Discussion:

There also is an efficiency consideration. Consider the transfer of an array of structures, as in the example given for an `hvector` component. The array is covered with strided vectors with homogeneous components; the fact that the entire data structure is contiguous in memory is likely to be lost.

Buffer descriptors perform two functions :

1. They describe the buffer **type**, i.e. the data type of each element of the buffer, and the relative displacement of the elements in memory.
2. They describe the address in memory of each element of the buffer.

Consider, for example, the following fragment of code.

```
int a[10];
...
mpi_create_buffer(&b1, MPI_EPHEMERAL);
mpi_append_contiguous(b1, (void *)a, 3, MPI_INT);
mpi_create_buffer(&b2, MPI_EPHEMERAL);
mpi_append_contiguous(b2, (void *)a, 3, MPI_INT);
mpi_create_buffer(&b3, MPI_EPHEMERAL);
mpi_append_contiguous(b3, (void *)(a+2), 3, MPI_INT);
```

Then `b1` and `b2` both point to buffer descriptors for the same buffer, whereas `b3` points to a buffer descriptor for another buffer; however, all three buffer descriptors have the same type.

In order to provide a recursive definition of buffer descriptors, one needs an object that specifies the buffer type, but not its location. By analogy, in a language such as C, a struct definition provides the structure type, and its declaration provides (implicitly) the address of the structure. The struct definition can be used recursively as a component of a new struct definition.

Rather then adding new functions and new objects, we propose to use buffer descriptors in both roles: "bound", to indicate the type and location of a buffer, and "unbound", to indicate a type, that can be used as a component of a new buffer definition.

Formally, the `MPI_APPEND_xxx` functions detailed above take a handle to a previously defined, uncommitted buffer descriptor as their `datatype` argument (rather than a state variable). The (composite) element described by this argument is an element of the same type as the buffer argument.

Example 2:

```
int i[2] = { 2, 0};
int a[100];
...
mpi_create_buffer( &b1, MPI_PERSISTENT);
```

```
mpi_append_indexed( b1, (void *)a, 2, i, MPI_INT);

mpi_create_buffer( &b2, MPI_PERSISTENT);
mpi_append_contiguous( b2, (void *)a, 3, b1);

mpi_create_buffer( &b3, MPI_PERSISTENT);
mpi_append_indexed( b3, (void *)a, 2, i, b2);
mpi_append_vec( b3, (void *)(a+50), 3, 3, 2, b1);
```

`b1` is associated with a buffer consisting of the two elements `a[2], a[0]`. When used unbound, it indicates a component with two integers that are one integer appart; this component occupies three integer storage units (12 bytes, if integers occupy four bytes).

The buffer associated with `b2` has 3 components of the type indicated by `b1`. It consists of the array entries `a[2], a[0], a[5], a[3], a[8], a[6]`. When used unbound, it indicates a component with six integers that occupy nine integer storage units.

The buffer associated with `b3` has two component of the type indicated by `b2` and three components of the type indicated by `b1`. It consists of the array entries

```
a[20], a[18], a[23], a[21], a[26], a[24],
 a[2],  a[0],  a[5],  a[3],  a[8],  a[6],
a[52], a[50], a[55], a[53], a[59], a[57].
```

Any uncommitted buffer descriptor may be used as a data type. MPI predefines (handles to) uncommitted buffer descriptors `MPI_INT`, `MPI_FLOAT`, etc., for each basic datatype of Fortran and C. Each of these descriptors can be thought as describing a buffer with one element of the corresponding type.

When a buffer descriptor isd used as a type descriptor then MPI ensures the alignments required by the underlying architecture and add, if required, padding space.

Example 3:
Consider the following C code which builds a buffer descriptor for an array of structures.

```
/* A brain damaged structure which needs both internal and external
 * padding on a 32 bit machine with strict alignment requirements
 */
struct foostruct {
   int i;
   /* four bytes of padding to get back to double alignment */
   double d[5];
   char b;
   /* trailing 7 bytes of padding to enforce alignment again */
};

struct foostruct foo[100];

MPI_BUFFER t;
MPI_BUFFER fb;

mpi_create_buffer(&t,  MPI_EPHEMERAL);
```

```
mpi_create_buffer(&fb, MPI_PERSISTENT);

/* Describe the data type in t */
mpi_append_contiguous(t, &((struct foostruct *)0)->i,1,MPI_INT);
mpi_append_contiguous(t, &((struct foostruct *)0)->d,5,MPI_DOUBLE);
mpi_append_contiguous(t, &((struct foostruct *)0)->b,1,MPI_BYTE);

/* Now describe foo itself, a contiguous array of 100 elements
 * of the derived type.
 */
mpi_append_contiguous(fb,&foo,100,t);
mpi_commit(fb);
```

Example 4:
Consider the following code which expresses as a single buffer descriptor a subset of an arbitrary dimensional array.

```
    real a(100,100,100)
c   declare the buffer descriptors
    integer onedslice
    integer twodslice
    integer message

c   Slice out the section a(1:21:2,1:10,1:11) as a single buffer descriptor

    call mpi_create_buffer(ondeslice, MPI_EPHEMERAL)
    call mpi_create_buffer(twodslice, MPI_EPHEMERAL)
    call mpi_create_buffer(message,   MPI_EPHEMERAL)

    call mpi_append_vec (onedslice, a, 1+(21-1)/2,
   $                     2, 1, MPI_REAL)
    call mpi_append_hvec(twodslice, a,        10,
   $                     100*MPI_REAL_SIZE, 1, ondeslice)
    call mpi_append_hvec(message,   a,               11,
   $                     100*100*MPI_REAL_SIZE,1,twodslice)

    call mpi_commit(message)

c   all done...
```

Discussion:
For portability this example requires access to the size in bytes of an MPI data type. (In general a buffer descriptor). We should probably provide an `MPI_SIZEOF( bufhandle )` function to achieve this.

Discussion:

We should probably restrict a buffer descriptor that is used as a type parameter to describe a buffer that is contained within one (composite) variable (array, structure, etc.). The worry is that on a machine with a segmented address space different variables may not belong to the same address segment, so that their relative displacement is not defined (think of a buffer that contains a heap variable, a stack variable and a static variable).

Alternative within alternative

A variant on the previous proposal is to keep unbound buffer descriptors syntactically distinct from bound buffer descriptors. A buffer descriptor is **unbound** if it is not associated with a fixed memory address. A buffer descriptor can be used as a datatype if and only if it is unbound. This prevents the possible confusion arising from using the same object for two different functions, one where the starting address is significant and one where the starting address should be ignored.

An unbound buffer descriptor is created by using the special `MPI_NULL` value as the `start` parameter in `MPI_APPEND_xxx` operations that create this descriptor. An unbound buffer descriptor can also be created by **unbinding** a bound buffer descriptor.

MPI_UNBIND(handlein, handleout)

IN handlein handle to bound buffer descriptor

OUT handleout handle to unbound buffer descriptor with the datatype of the input buffer descriptor

Discussion:
The `MPI_UNBIND` function allows to create a buffer datatype after the pattern of an existing buffer and, therefor, after the pattern of an existing data structure in memory.

We repeat the last three examples, using this alternative notation.

Example 2':

```
int i[2] = { 2, 0};
int a[100];
...
mpi_create_buffer( &b1, MPI_PERSISTENT);
mpi_append_indexed( b1, MPI_NULL, 2, i, MPI_INT);

mpi_create_buffer( &b2, MPI_PERSISTENT);
mpi_append_contiguous( b2, MPI_NULL, 3, b1);

mpi_create_buffer( &b3, MPI_PERSISTENT);
mpi_append_indexed( b3, (void *)a, 2, i, b2);
mpi_append_vec( b3, (void *)(a+50), 3, 3, 2, b1);
```

Example 3:

```
struct foostruct {
   int i;
   double d[5];
   char b;
};

struct foostruct foo[100];

MPI_BUFFER t, tt;
MPI_BUFFER fb;

mpi_create_buffer(&t,  MPI_EPHEMERAL);
mpi_create_buffer(&fb, MPI_PERSISTENT);

/* Describe the data type in t */
mpi_append_contiguous(t, &((struct foostruct *)0)->i,1,MPI_INT);
mpi_append_contiguous(t, &((struct foostruct *)0)->d,5,MPI_DOUBLE);
mpi_append_contiguous(t, &((struct foostruct *)0)->b,1,MPI_BYTE);

/* Abstract buffer datatype */
mpi_unbind(t, tt);

/* Now describe foo itself, a contiguous array of 100 elements
 * of the derived type.
 */
mpi_append_contiguous(fb,&foo,100,tt);
mpi_commit(fb);
```

Example 4:

```
    real a(100,100,100)
c   declare the buffer descriptors
    integer onedslice
    integer twodslice
    integer message

c   Slice out the section a(1:21:2,1:10,1:11) as a single buffer descriptor

    call mpi_create_buffer(ondeslice, MPI_EPHEMERAL)
    call mpi_create_buffer(twodslice, MPI_EPHEMERAL)
    call mpi_create_buffer(message,   MPI_EPHEMERAL)

    call mpi_append_vec (onedslice, MPI_NULL, 1+(21-1)/2,
   $                     2, 1, MPI_REAL)
    call mpi_append_hvec(twodslice, MPI_NULL,  10,
   $                     100*MPI_REAL_SIZE, 1, ondeslice)
    call mpi_append_hvec(message, a,           11,
```

```
   $                      100*100*MPI_REAL_SIZE,1,twodslice)

    call mpi_commit(message)

c   all done...
```

Type matching

The rules for buffer type matching are extended to composite buffer descriptors: A send buffer descriptor matches a receive buffer descriptor is the successive basic entries in each descriptor have identical names.

Example:

```
struct {
   int a[5];
   char c;
       } st[100];


...
mpi_create_buffer(&b1, MPI_PERSISTENT);
mpi_append_contiguous(b1, (void *)st, 5, MPI_INT);
mpi_append_contiguous(b1, (void *)&(st.c), 1, MPI_CHAR);
mpi_append_contiguous(b1, (void *)st, 5, MPI_INT);
mpi_append_contiguous(b1, (void *)&(st.c), 1, MPI_CHAR);
...
mpi_create_buffer(&b2, MPI_PERSISTENT);
mpi_append_contiguous(b2, (void *)st, 5, MPI_INT);
mpi_append_contiguous(b2, (void *)&(st.c), 1, MPI_CHAR);

mpi_create_buffer(&b3, MPI_PERSISTENT);
mpi_append_contiguous(b3, (void *)st, 2, b2);
```

The buffer descriptors associated with `b1` and `b3` have matching types. They both correspond to a sequence of `int, int, char, int, int, char`. (in fact, both buffers consist of the same sequence of variables).

Discussion:

A more restrictive definition of type matching is to require "structural matching": I.e., two descriptors match if all corresponding top (composite) components recursively match pairwise. With such definition, the two buffer descriptors in the last example don't have matching types. I (Marc) believe that the weaker definition makes more sense in our environment, since we do not, in general, require any corrspondance between data alignement at sender and data alignment at receiver.

Type matching of buffer descriptor and host language is dealt similarly.

Message length

The length of a message is defined as the number of basic elements (components with basic datatypes) in the message. This is the value returned in the status of a receive operation for that message.

Discussion:
An alternative definition is to count message length as the number of top components in the message. If we accept the weaker definition of type compatibility, this has the effect that the length of a message sent may be very different from the length of the message received – even in a homogeneous environment. Consider in the previous example the case of message sent with `b1` and received with `b3` – the length has halved.

On the other hand, if we accept the stronger definition of type compatibility, counting top elements is plausible. Since top components are required to match pairwise, the message length is integer, and identical at sending and receiving sites.

Buffer descriptors as datatypes in communication operations

We propose to allow the use of buffer descriptor handles as datatypes in contiguous buffer communication operations (e.g. in `MPI_SENDC`)

If we do so, then we don't need anymore the more general communication operations with a `buffer_handle` argument!!!
`MPI_SEND(buffer_handle, dest, tag, context)`
is replaced by
`MPI_SENDC(start, 1, buffer_handle, dest, tag, context)`,
where `start` is the first buffer location. (If unbound buffer descriptors are kept distinct from bound buffer descriptors, then the argument `buffer_handle` gets replaced by the unbound datatype of the buffer.)

Such simplification would be a big win, as communication with general buffers becomes a special case of communication with contiguous buffers. One caveat: When a buffer descriptor is used as a datatype, we may have restrictions on where from the different variables appearing in the buffer descriptor come from: They need to all come from the same contiguous address sapce, so that their relative displacement is well defined. It means that we shall only be able to pack in one message variables that belong to the same contiguous address space. We have to think about syntactic rules to enforce this restriction: E.g., is it sufficient to say in C that a buffer is either all in stack, or all in heap, or all in static? What is the equivalent Fortran rule? Are such rules acceptable?

Discussion: OPINIONS?

Other issues

An ephemeral buffer descriptor can be freed as soon as it has been used as a type parameter in a buffer descriptor creation function.

A buffer descriptor handle cannot be passed to the same `MPI_APPEND_xxx` function both as a buffer descriptor handle and as a datatype (no self-references).

Discussion: The last constraint can be removed if somebody cares.

2.5.4 Type Matching

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

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

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type 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 receiver buffer descriptor. A program that fails to observe these three rules is erroneous.

To define type matching more precisely, we need to deal with two issues: matching of types of the host language with buffer descriptor types; and matching of buffer descriptor types at sender and receiver. We restrict ourselves to the case where the sending and receiving programs 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 if they have identical names: `MPI_INTEGER` matches `MPI_INTEGER`, `MPI_REAL` matches `MPI_REAL`, and so on.

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

We thus have two cases:

- Communication of typed values (e.g., with datatype different from `MPI_BYTE`), where the datatypes of the corresponding entries in the sender program, in the sender buffer descriptor, in the receiver buffer descriptor and in the receiver program should all match.
- Communication of untyped values (e.g., of datatype `MPI_BYTE`), where the the corresponding entries in the sender and the receiver buffer descriptors have both datatype `MPI_BYTE`. In this case, there are no requirements on the datatypes of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.

2.5.5 Data conversion

We use the following terminology:

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

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

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

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. An exception occurring during representation conversion results in a failure of the communication; an error occurs either in the send operation, or the receive operation, or both.

If a value sent in a message is untyped (i.e., of type `MPI_BYTE`), then the binary representation of the byte stored at the receiver is identical to the binary representation of the byte loaded at the sender. This holds true, whether sender and receiver run in the same or in distinct environments; no representation conversion is required.

Note that no conversions ever occur when an MPI program executes in a homogeneous system, where all processes run in the same environment. Also note the different behavior of `MPI_BYTE` and of `MPI_CHARACTER`. A buffer descriptor entry with datatype of `MPI_CHARACTER` can only match a Fortran variable of type `CHARACTER`; and representation conversion may occur when values of type `MPI_CHARACTER` are transferred., e.g., from an EBCDIC encoding to an ASCII encoding.

Consider the following examples.

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

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 desirable 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_APPEND_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_APPEND_CONTIGUOUS (BH1, A(1), 1, MPI_INTEGER)
...
CALL MPI_CREATE_BUFFER(BH2, MPI_PERSISTENT)
CALL MPI_APPEND_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 (desirable but not required) 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.

Data representation conversion also applies to the envelope of a message: source, destination and tag are all integers that may need to be converted.

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.

Discussion:
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. One proposal is to allow both Fortran routines to specify C datatypes in their buffer descriptors, and for C routines to specify Fortran dataypes. The draft will specify rules for type matching between Fortran variables and C datatypes in the buffer descriptors and between C variables and Fortran datatypes in the buffer descriptor. (E.g., a Fortran variable of type REAL may match an entry of type MPI_FLOAT in a buffer descriptor, but not an entry of type MPI_INTEGER. Data representation conversion occurs if corresponding C and Fortran types have different representations on the same process. There still is the requirement that buffer descriptor entries on the send side match verbatim buffer descriptor entries on the receive side, and there is no change in interprocess communication. TO DISCUSS

This proposal gives a precise meaning to interlanguage communication. It has two drawbacks: It may require additional data conversions when messages are assembled or disassembled, and it imposes a fixed correspondence between C and Fortran type which may not be the preferred one on the target machine. (Some machines may want Fortran `REAL` to match C `FLOAT`; other may want it to match C `double`; MPI will enforce one of the two.)

Another (non)solution is to ignore the issue and 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 wildcard `MPI_ANY_SOURCE` value for `source`, and/or a wildcard `MPI_ANY_TAG` value for `tag`, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value for `context` 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=`MPI_ANY_SOURCE` in the pattern, and has a matching tag unless tag=`MPI_ANY_TAG` in the pattern.

The length of the received message must be less then 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 be started whether or not a matching receive has been posted.

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

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

The start and completion of a **standard send** do not depend on the status of a matching receive.

A **ready send** should be started only if a matching receive is already posted; otherwise the operation is erroneous and its outcome is undefined. In some systems, this allows the removal of a hand-shake operation that is otherwise required, and results in improved performance. Its completion does not depend on the status of a matching receive.

A **synchronous send** can be started irrespective of the status of a matching receive. It will complete only if a matching receive is posted, and the receive operation is committed to receive the message sent by the secure send. (I.e., the receive has been posted, the incoming message has been matched to this posted receive, and the receive operation has already started and cannot be cancelled anymore.)

There is only one receive mode, which can match any of the send modes.

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

Implementation note: A ready send can be implemented as a regular send; in such case there will be no performance advantage (or disadvantage) for the use of ready send; and some erroneous programs may go undetected (and may still produce correct results).

A possible communication protocol for the various communication modes is outlined below:

`ready send`: The message is sent asap; an acknowledge is returned by receiver. The protocol is (send, ack).

`synchronous send:` The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message. The protocol is (req, ack-req, send, ack).

`standard send:` First protocol may be used for short messages, and second protocol for long messages.

Of course, there are many other possible choices.

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 an object persists only until the communication operation is completed, at which point it is destroyed. Thus the correct invocation of suboperations with an ephemeral object is **INIT START COMPLETE**.

A user may directly invoke these suboperations. This would allow the amortization of the overhead of setting up a communication over many successive uses of the same handle, and allows the overlap of 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 has been invoked; the operation is **completed** once the **complete** suboperation has completed. 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

An object for a send operation is created by a call to **MPI_INIT_xSEND**. The prefix to `SEND` indicates the communication mode: none for `standard`, `R` for `ready`, and `S` for `synchronous`. 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, persistence)

Creates a communication object for standard mode send. 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 persistence handle persistence (state type, with two values: `MPI_PERSISTENT` and `MPI_EPHEMERAL`)

MPI_INIT_RSEND (handle, buffer_handle, dest, tag, context, persistence)

Creates a communication object for standard mode send. 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 persistence handle persistence (state type, with two values: `MPI_PERSISTENT` and `MPI_EPHEMERAL`)

MPI_INIT_SSEND (handle, buffer_handle, dest, tag, context, persistence)

Creates a communication object for synchronous mode send. 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 persistence handle persistence (state type, with two values: `MPI_PERSISTENT` and `MPI_EPHEMERAL`)

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

Creates 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 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:
Ephemeral "persistent handles" are not terribly useful as such. But their existence allow us to describe all the higher level send/receive operations in terms of basic operations on handles.

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 send can be started provided that the previous send using the same handle has completed, or as soon as the handle has been created, if there was no preceding send. If the send mode is `ready` then, in addition, a matching receive should have been already posted. The send buffer should not be updated after the send is started, until the operation completes.

A receive can be started as soon as the preceding receive using the same handle has completed, or as soon as the handle has been created is there was no preceding receive. The receive buffer should not be accessed after the receive is started, until the operation completes.

A program that does not satisfy these conditions is erroneous and its outcome is undetermined.

The call returns as soon as possible. A call that starts a send can return before the data to be sent has been cleared out of the send buffer and before any matching receive has been posted (this is true for all send modes, including `synchronous`). A call that starts a receive can return before andy data has been received in the receive buffer.

2.8.3 Communication Completion

The completion of a send operation indicates that the sender is now free to update the locations in the send buffer, or any other location that can be referenced by the send operation. It does not indicate that the message has been received; rather, it may have been buffered by the communication subsystem. However, if a `synchronous` mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive.

The completion of a receive operation indicates that the 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 (but indicates, of course, that the send was initiated).

MPI_WAIT (handle, status)

IN handle communication handle

OUT status handle that is associated with return status object. The return status object is updated by call.

A call to **MPI_WAIT** returns when the send operation identified by `handle` is complete.

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

MPI_TEST (handle, flag, status)

IN handle communication handle

OUT flag logical

OUT status handle that is associated with return status object.

A call to **MPI_TEST** 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 operation. 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 multithreaded, 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_TEST`) allows the user to schedule alternative activities within a single thread of execution.

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

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

MPI_RETURN_STATUS(handle, count, source, tag)

IN handle handle to return status object

OUT count number of elements received.

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

OUT tag tag of received message (integer).

MPI_CREATE_STATUS(handle)

Creates a new return status object.

OUT handle handle to return status object.

TO DISCUSS

Discussion:
Should we have persistent/ephemeral status objects?

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

MPI_WAITANY (list_of_handles, count, index, status)

Blocks until one of the operations associated with the communication handles in the array has completed. Returns the index of that handle in the array, and returns the status of that operation in the object associated with the status. The parameters are:

IN list_of_handles list of handles to communication objects.

IN count list length (integer)

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

OUT status handle that is associated with return status object. Object is set to return status of operation that completed.

The successful execution of `MPI_WAITANY(list_of_handles, index, status)` has the same effect as the successful execution of `MPI_WAIT(handle[i], status)`, where `i` is the value returned by `index` and `handle[i]` is the `i`-th handle in the list, and the cancellation 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 Chapter 6).

`MPI_WAITANY ( list_of_handles, count, index, status)` is

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

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

MPI_TESTANY (list_of_handles, count, index, status)

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 latter case, it returns a value of -1 in `index` and `status` is undefined. The parameters are:

IN list_of_handles list of handles to communication objects.

IN count list length (integer)

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

OUT status 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, count, list_of_statuss)

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 count lists length (integer)

OUT list_of_status 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 Chapter 6 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 6 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 dropped 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 from a user specified position. TO DISCUSS

A proposal was made for an `MPI_TESTALL` function that returns successfully (`flag = true`) if all operations in the list have terminated, unsuccessfully (`flag = false`), otherwise.

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 same naming convention as for communication handle creation: a prefix of `R` indicates the `ready` mode, and a prefix of `S` indicates the synchronous mode.

MPI_SEND (buffer_handle, dest, tag, context)

Blocking standard send.

`MPI_SEND (buffer_handle, dest, tag, context)` is

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

MPI_RSEND (buffer_handle, dest, tag, context)

blocking ready send
`MPI_RSEND (buffer_handle, dest, tag, context)` is

```
MPI_INIT_RSEND(handle, buffer_handle, dest, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle, null)
```

MPI_SSEND (buffer_handle, dest, tag, context)

Blocking synchronous send.
`MPI_SSEND (buffer_handle, dest, tag, context)` is

```
MPI_INIT_SSEND(handle, buffer_handle, dest, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle, null)
```

MPI_RECV(buffer_handle, source, tag, context, status)

Blocking receive
`MPI_RECV(buffer_handle, source, tag, context, status)` is

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

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_TEST`, `MPI_WAITANY`, `MPI_TESTANY`, 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` (**SYNCHRONOUS**) mode. In addition, a prefix of **I** is used to indicate *immediate* (i.e., nonblocking) execution.

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

Nonblocking standard send.

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

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

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

Nonblocking ready send.

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

```
MPI_INIT_RSEND(handle, buffer_handle, dest, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
```

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

Nonblocking synchronous send.

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

```
MPI_INIT_SSEND(handle, buffer_handle, dest, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
```

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

Nonblocking receive

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

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

2.11 Contiguous Buffer Communication 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, count, datatype, dest, tag, context)

Blocking send of contiguous buffer in standard mode.

`MPI_SENDC (start, count, datatype, dest, tag, context)` is

```
MPI_CREATE_BUFFER( buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS( buffer_handle, start, count, datatype)
MPI_COMMIT_BUFFER( buffer_handle)
MPI_SEND (buffer_handle, count, dest, tag, context)
```

MPI_RSENDC (start, count, datatype, dest, tag, context)

Blocking send of contiguous buffer in ready mode.

`MPI_RSENDC (start, count, datatype, dest, tag, context)` is

```
MPI_CREATE_BUFFER( buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS( buffer_handle, start, count, datatype)
MPI_COMMIT_BUFFER( buffer_handle)
MPI_RSEND( buffer_handle, count, dest, tag, context)
```

MPI_SSENDC (start, count, datatype, dest, tag, context)

Blocking send of contiguous buffer in synchronuous mode.

`MPI_SSENDC (start, count, datatype, dest, tag, context)` is

```
MPI_CREATE_BUFFER( buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS( buffer_handle, start, count, datatype)
MPI_COMMIT_BUFFER( buffer_handle)
MPI_SSEND( buffer_handle, count, dest, tag, context)
```

MPI_RECVC (start, count, datatype, source, tag, context, status)

Blocking receive of contiguous buffer.

`MPI_RECVC (start, count, datatype, source, tag, context, status)` is

```
MPI_CREATE_BUFFER( buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS( buffer_handle, start, count, datatype)
MPI_COMMIT_BUFFER( buffer_handle)
MPI_RECV( buffer_handle, source, tag, context, return_handle)
```

MPI_ISENDC (handle, start, count, datatype, dest, tag, context)

Nonblocking send of contiguous buffer in standard mode.

`MPI_ISENDC (handle, start, count, datatype, dest, tag, context)` is

```
MPI_CREATE_BUFFER( buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS( buffer_handle, start, count, datatype)
MPI_COMMIT_BUFFER( buffer_handle)
MPI_ISEND( handle, buffer_handle, dest, tag, context)
```

MPI_IRSENDC (handle, start, count, datatype, dest, tag, context)

Nonblocking send of contiguous buffer in ready mode.

`MPI_IRSENDC (handle, start, count, datatype, dest, tag, context)` is

```
MPI_CREATE_BUFFER( buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS( buffer_handle, start, count, datatype)
MPI_COMMIT_BUFFER( buffer_handle)
MPI_IRSEND( handle, buffer_handle, dest, tag, context)
```

MPI_ISSENDC (handle, start, count, datatype, dest, tag, context)

Nonblocking send of contiguous buffer in synchronous mode.

`MPI_ISSENDC (handle, start, count, datatype, dest, tag, context)` is

```
MPI_CREATE_BUFFER( buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS( buffer_handle, start, count, datatype)
MPI_COMMIT_BUFFER( buffer_handle)
MPI_ISSEND( handle, buffer_handle, dest, tag, context)
```

MPI_IRECVC(handle, start, count, datatype, source, tag, context)

Nonblocking receive of contiguous buffer.

`MPI_IRECVC(handle, start, count, datatype, source, tag, context)` is

```
MPI_CREATE_BUFFER (buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS( buffer_handle, start, count, datatype)
MPI_COMMIT_BUFFER( buffer_handle)
MPI_IRECV( handle, buffer_handle, source, tag, context)
```

2.12 Probe and Cancel

The `MPI_PROBE` operation allows incoming messages to be checked for, 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 the amount of storage needed to receive the message to be computed, when this length is not readily computed from the information returned by `MPI_PROBE`.

The `MPI_CANCEL` operation allows pending communications to be cancelled. 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_IPROBE(source, tag, context, flag, type, 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 status handle that is associated with return status object.

`MPI_IPROBE` returns `flag = true` if there is a message that can be received and that matches the pattern specified by the parameters `source`, `tag`, and `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 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_IPROBE` will receive the message that was matched by the probe, if no other intervening receive occurred after the probe. If the receiving process is multithreaded, it is the user responsibility to ensure that the last condition holds.

MPI_PROBE(source, tag, context, type, 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.

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

OUT status handle that is associated with return status object.

`MPI_PROBE` behaves like `MPI_IPROBE` except that it is a blocking call which returns only after a matching message has been found.

Discussion:

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

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 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 awkward, 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(count, status, buffer_descriptor)

Computes the number of elements that were to 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 count number of elements that were to be received (integer)

IN status handle to return status descriptor

IN buffer_descriptor handle to buffer descriptor

Discussion:
In order to support this function, an additional field is needed in the return status object, i.e., number of bytes in the incoming message matched by probe.

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 succeeded, `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 send was cancelled 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 receive buffer posted will not be altered, and any send matching this receive has to be satisfied by another receive.

A successful cancel operation does not complete the cancelled communication; the operation still needs to be completed with an `MPI_WAIT` or `MPI_TEST` call, if it is an nonblocking communication. TO DISCUSS

Discussion:
We may want a nonblocking cancel, that returns asap, since, anyhow, one has to wait for the `MPI_TEST` or `MPI_WAIT` to return before the buffer is reused. If we do so, then the `MPI_CANCEL` call will not return information on the success or failure of the cancel; this will be returned by the `MPI_WAIT` or `MPI_TEST` operation.

The `MPI_TEST` executed after an operation has been cancelled needs to return an indication that the operation failed. On the other hand, we do not want the program to blow up in such situation. Thus, this indication has to be handled as a nonfatal error, even in the execution mode where any unsuccessful communication is fatal. Do we want this information to be returned in the error code, or to be returned as part of the status?

Can a thread cancel a blocking communication executed by another thread?

Do we want `MPI_CANCEL` to free the communication object?

Implementation note:
The cancel operation may be expensive to implement on some systems.

2.13 send-receive

DISCUSS

Discussion:
This section has not yet been discussed by MPIF.

The **send-receive** operations combines in one call the sending of a message to one destination and the receiving of another message, form another destination, possibly the same. The **exchange** operations are the same as send-receive, except that the send buffer and the receive buffer are identical. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such shift, then one needs to order correctly the sends and receives (e.g. even sends, next receives, odd receives first, next sends) so as to prevent cyclic dependencies that lead to deadlock. When a send-receive ro exchange operation is used, the communication subsystem takes care of these issues.

A handle for a send-receive (exchange) operation is created by a call to `MPI_INIT_SENDRECV` (`MPI_INIT_EXCHANGE`). The operations can be started and completed like regular communication operations, using the `MPI_START`, `MPI_WAIT`, `MPI_WAITALL`, etc. In addition, operations are provided for blocking and nonblocking send-receive and exchange operations, with general and contiguous buffers.

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

Discussion:
The correctness chapter need a section on send-receives – the correctness issue is not entirely trivial, because of possible cyclic dependencies. Even in an implementation where messages are not buffered, MPi has to guarantee that blocking calls to send-receive that collectively execute a shift on a cycle will not deadlock. Some system buffering is, in fact, necessary when exchanges are used.

MPI_INIT_SENDRECV(handle, send_buffer_handle, dest, recv_buffer_handle, source, tag, context, persistence)

Creates a communication object for a send-receive operation.

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

IN send_buffer_handle handle to send buffer descriptor

IN dest rank in context of destination (integer)

IN recv_buffer_handle handle to receive buffer descriptor

IN source rank of source in context (integer)

IN tag tag used both for send and for receive operation (integer)

IN context context both for send and receive operation (handle)

IN persistence communication object persistence

MPI_INIT_EXCHANGE(handle, buffer_handle, dest, source, tag, context, persistence)

Creates a communication object for an exchange operation.

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

IN buffer_handle handle to send and receive buffer descriptor

IN dest rank in context of destination (integer)

IN source rank of source in context (integer)

IN tag tag used both for send and for receive operation (integer)

IN context context both for send and receive operation (handle)

IN persistence communication object persistence

Discussion:
Do we also want send-receive and exchange operations with the `ready` and `synchronous` mode?
Do we want `tag`? (Collective communication don't use tag; but tag is needed if we want to be able to mix send-receive and exchange with regular sends and receives.)
Do we want to allow wildcard source and/or tag? Since I assumed that the message sent has the same tag as the message received, then widcard tag does not make much sense.

MPI_SENDRECV(handle, send_buffer_handle, dest, recv_buffer_handle, source, tag, context)

Blocking send-receive operation. The operation returns when both send and receive have completed.

`MPI_SENDRECV(handle, send_buffer_handle, dest, recv_buffer_handle, source, tag, context, status)` is

```
MPI_INIT_SENDRECV(handle, send_buffer_handle, dest,
   recv_buffer_handle, source, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle, status)
```

MPI_EXCHANGE(handle, buffer_handle, dest, source, tag, context, status)

Blocking exchange operation. The operation returns when both send and receive have completed.

`MPI_EXCHANGE(handle, buffer_handle, dest, source, tag, context, status)` is

```
MPI_INIT_EXCHANGE(handle, buffer_handle, dest,
   source, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle, status)
```

MPI_ISENDRECV(handle, send_buffer_handle, dest, recv_buffer_handle, source, tag, context)

Nonblocking send-receive operation.

`MPI_ISENDRECV(handle, send_buffer_handle, dest, recv_buffer_handle, source, tag, context)` is

```
MPI_INIT_SENDRECV(handle, send_buffer_handle, dest,
   recv_buffer_handle, source, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
```

MPI_IEXCHANGE(handle, buffer_handle, dest, source, tag, context)

Nonblocking exchange operation.

`MPI_IEXCHANGE(handle, buffer_handle, dest, source, tag, context)` is

```
MPI_INIT_EXCHANGE(handle, buffer_handle, dest,
   source, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
```

MPI_SENDRECVC(handle, send_start, send_count, send_datatype, dest, recv_start, recv_count, recv_datatype, source, tag, context, status)

Blocking send-receive with contiguous buffers.

`MPI_SENDRECVC(handle, send_start, send_count, send_datatype, dest, recv_start, recv_count, recv_datatype, source, tag, context, status)` is

```
MPI_CREATE_BUFFER(send_buffer_handle, MPI_EPHEMERAL)
MPI_CREATE_BUFFER(recv_buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS(send_buffer_handle, send_start, send_count,
     send_datatype)
MPI_APPEND_CONTIGUOUS(recv_buffer_handle, recv_start, recv_count,
     recv_datatype)
MPI_COMMIT(send_buffer_handle)
MPI_COMMIT(recv_buffer_handle)
MPI_INIT_SENDRECV(handle, send_buffer_handle, dest,
   recv_buffer_handle, source, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle, status)
```

MPI_EXCHANGEC(handle, start, count, datatype, dest, source, tag, context)

Blocking exchange with contiguous buffer

MPI_EXCHANGEC(handle, start, count, datatype, dest, source, tag, context) is

```
MPI_CREATE_BUFFER(buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS(buffer_handle, start, count, datatype)
MPI_COMMIT(buffer_handle)
MPI_INIT_EXCHANGE(handle, buffer_handle, dest,
   source, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
MPI_WAIT(handle, status)
```

MPI_ISENDRECVC(handle, send_start, send_count, send_datatype, dest, recv_start, recv_count, recv_datatype, source, tag, context)

Nonblocking send-receive with contiguous buffers.

MPI_ISENDRECVC(handle, send_start, send_count, send_datatype, dest, recv_start, recv_count, recv_datatype, source, tag, context) is

```
MPI_CREATE_BUFFER(send_buffer_handle, MPI_EPHEMERAL)
MPI_CREATE_BUFFER(recv_buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS(send_buffer_handle, send_start, send_count,
     send_datatype)
MPI_APPEND_CONTIGUOUS(recv_buffer_handle, recv_start, recv_count,
     recv_datatype)
MPI_COMMIT(send_buffer_handle)
MPI_COMMIT(recv_buffer_handle)
MPI_INIT_SENDRECV(handle, send_buffer_handle, dest,
   recv_buffer_handle, source, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
```

MPI_IEXCHANGEC(handle, start, count, datatype, dest, source, tag, context)

Nonblocking exchange with contiguous buffer

MPI_IEXCHANGEC(handle, start, count, datatype, dest, source, tag, context) is

```
MPI_CREATE_BUFFER(buffer_handle, MPI_EPHEMERAL)
MPI_APPEND_CONTIGUOUS(buffer_handle, start, count, datatype)
MPI_INIT_EXCHANGE(handle, buffer_handle, dest,
   source, tag, context, MPI_EPHEMERAL)
MPI_START(handle)
```

Discussion:

Can we cancel a send-receive or exchange? If yes, should it be the case that either we cancel both send and receive or neither? This seems natural, but may cause problems (The receive has already executed, but the send is hung. How do we cancel?)

This is the preamble to a much more general question: can we cancel collective operations?

2.14 Null processes

Discussion: This section has not been reviewed by MPIF.

In many instances, it is convenient to specify a "dummy" source or destination for communication. This simplifies the code that is needed for dealing with boundaries, e.g., in the case of a noncircular shift done with calls to send-receive.

The special value `MPI_PROCNULL` can be used instead of a rank wherever a source or a destination parameter is required in a call. A communication with process `MPI_PROCNULL` has no effect: a send to `MPI_PROCNULL` succeeds and returns asap. A receive from `MPI_PROCNULL` succeeds and returns asap with no modifications to the receive buffer.

Discussion:
Need to decide what is returned in the status object: I suggest `source = MPI_PROCNULL`, `tag=0` and `count=0`.

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_COMM_INIT(group) Return the communicator of the inital group containing all processes.

OUT group group communicator.

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

IN group group communicator

OUT rank integer

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

IN group group communicator

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 communicator

The following shows one possible implementation of the barrier function. Similarly, many of the other collective communication functions are illustrated by possible point-to-point implementations. These are ment for illustration and are not suggested to be the most efficient ways to implement the given functions. `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);
   for (i=1; i < size; i++)
      MPI_SEND(buffer_handle, i, tag, group);
}
else
{
   MPI_SEND(buffer_handle, 0, tag, group);
   MPI_RECV(buffer_handle, 0, tag, group);
}
MPI_FREE(buffer_handle);
```

MPI_SYNCH(group, map)

MPI_SYNCH synchronizes only the subgroup of processes specified by map. Map is an integer array containing the ranks of the processes in group that are to be synchronized. This function is needed to bootstrap the safe formation of subgroups.

IN group group communicator

IN map an array of integers.

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 the buffer of the calling process.

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

IN group group communicator

IN root rank of broadcast root (integer)

MPI_BCASTC(buf, len, type, 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 (matches type)

IN len Number of entries in buffer (integer)

IN type data type of buffer

IN group group communicator

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

Gather

MPI_GATHER(inhandle, list_of_outhandles, group, root)

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. For example, the root places the data from process with rank 3 in the location specified by the third buffer descriptor in the list of outhandles. The list_of_outhandles argument is ignored for all non-root processes.

The routine 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.

Discussion: Jim Cownie and Al Geist have proposed that MPI allow a handle to point to a list of handles. This feature would clean up the collective communication interface and add additional functionality to MPI. Because the list of handles concept first appears in the point to point chapter under MPI_WAITANY, this proposal as written up by Jim should be discussed there.

IN inhandle handle to input buffer descriptor

IN list_of_outhandles list of buffer descriptor handles (root)

IN group group communicator

IN root rank of receiving process (integer)

MPI_GATHERC(inbuf, outbuf, inlen, type, 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 (matches type)

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

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

IN type data type of buffer

IN group group communicator

IN root rank of receiving process (integer)

`MPI_GATHERC( inbuf, outbuf, inlen, type, 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);
      outbuf += inlen;
   }
MPI_WAIT(handle, return_handle);
```

Scatter

MPI_SCATTER(list_of_inhandles, outhandle, group, root)

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_inhandles list of buffer descriptor handles (only significant on the root)

IN outhandle buffer descriptor handle

IN group group communicator

IN root rank of sending process (integer)

`MPI_SCATTER( list_of_inhandles, outhandle, group, root )` is

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

MPI_SCATTERC(inbuf, outbuf, len, type, 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 (matches type).

OUT outbuf first entry in output buffer (matches type).

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

IN type data type of buffer

IN group group communicator

IN root rank of sending process (integer)

`MPI_SCATTERC( inbuf, outbuf, outlen, type, group, root)` is

```
MPI_GSIZE( &size, group);
MPI_RANK( &rank, group);
MPI_IRECVC( handle, outbuf, outlen, type, root, tag, group);
```

```
if (rank=root)
   for (i=0; i < size; i++)
   {
      MPI_SENDC(inbuf, outlen, type, i, tag, group);
      inbuf += outlen;
   }
MPI_WAIT(handle);
```

All-to-all scatter

MPI_ALLSCATTER(list_of_inhandles, list_of_outhandles, group)

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. For example, each process places the data from process with rank 3 in the location specified by the third buffer descriptor in the list of outbufs. The routine is called by all members of the group using the same arguments for `group`.

IN list_of_inhandles list of buffer descriptor handles

IN list_of_outhandles list of buffer descriptor handles

IN group group communicator

MPI_ALLSCATTERC(inbuf, outbuf, len, type, 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 `i`-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 (matches type).

OUT outbuf first entry in output buffer (matches type).

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

IN type data type of buffer

IN group communicator

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

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

```
    outbuf += len;
   }
for (i=0; i < size; i++)
   {
    MPI_ISENDC(send_handles[i], inbuf, len, type, i, tag, group);
    inbuf += len;
   }
MPI_WAITALL(send_handles, size, return_handles);
MPI_WAITALL(recv_handles, size, return_handles);
```

All-to-all broadcast

MPI_ALLCAST(inhandle, list_of_outhandles, group)

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. For example, each process places the data from process with rank 3 in the location specified by the third buffer descriptor in the list of outbufs. The routine is called by all members of the group using the same arguments for `group`.

IN inhandle buffer descriptor handle for input buffer

OUT list_of_outhandles list of buffer descriptor handles

IN group handle

MPI_ALLCASTC(inbuf, outbuf, len, type, 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 (matches type).

OUT outbuf first entry in output buffer (matches type).

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

IN type data type of buffer

IN group group communicator

`MPI_ALLCASTC( inbuf, outbuf, len, type, group)` is

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

```
    outbuf += len;
   }
for (i=0; i < size; i++)
   {
    MPI_ISENDC(send_handle[i], inbuf, len, type, i, tag, group);
   }
MPI_WAITALL(send_handle);
MPI_WAITALL(recv_handle);
```

3.6 Global Compute Operations

Reduce

MPI_REDUCE(inhandle, outhandle, 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 inhandle handle to input buffer

IN outhandle handle to output buffer – significant only at root

IN group group communicator

IN root rank of root process (integer)

IN op operation

The buffer descriptor contains data type information so that the correct form of the operation can be performed. We list below the operations which are supported.

MPI_MAX maximum

MPI_MIN minimum

MPI_SUM sum

MPI_PROD product

MPI_AND and (logical or bit-wise integer)

MPI_OR or (logical or bit-wise integer)

MPI_XOR xor (logical or bit-wise integer)

MPI_MAXLOC maximum value and rank of process with maximum value

MPI_MINLOC minimum value and rank of process with minimum value

MPI_REDUCEC(inbuf, outbuf, len, type, 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 type data type of buffer

IN group group communicator

IN root rank of root process (integer)

IN op operation

MPI_USER_REDUCE(inhandle, outhandle, group, root, function, unitsize)

Same as the reduce operation function above except that a user supplied function is used. `function` is an associative and commutative function with three arguments. `f( invec, inoutvec, *len)`

IN inhandle handle to input buffer

IN outhandle handle to output buffer – significant only at root

IN group group communicator

IN root rank of root process (integer)

IN function user defined function

IN unitsize minimum size of data that function requires.

MPI_USER_REDUCEC(inbuf, outbuf, len, type, group, root, function, unitsize)

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 type data type of buffer

IN group group communicator

IN root rank of root process (integer)

IN function user provided function

IN unitsize minimum size of data that function requires.

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

MPI_ALLREDUCE(inhandle, outhandle, group, op)

Same as the reduce operation function above except that the result appears in the outbuf of all the group members.

MPI_ALLREDUCEC(inbuf, outbuf, len, type, group, op)

Same as the reducec operation function above except that the result appears in the outbuf of all the group members.

MPI_USER_ALLREDUCE(inhandle, outhandle, group, function)

Same as the user_reduce operation function above except that the result appears in the outbuf of all the group members.

MPI_USER_ALLREDUCEC(inbuf, outbuf, len, type, group, function)

Same as the user_reducec operation function above except that the result appears in the outbuf of all the group members.

Scan

MPI_SCAN(inhandle, outhandle, 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 inhandle handle to input buffer

IN outhandle handle to output buffer

IN group group communicator

IN op operation

MPI_SCANC(inbuf, outbuf, len, type, group, op)

Same as `MPI_SCAN`, restricted to block buffers.

IN inbuf first input buffer element (matches type)

OUT outbuf first output buffer element (matches type)

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

IN type data type of buffer

IN group group communicator

IN op operation

MPI_USER_SCAN(inhandle, outhandle, 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 an input vector, an inout vector, and a length argument. The types of the two vectors and of the returned values all agree.

IN inhandle handle to input buffer

IN outhandle handle to output buffer

IN group group communicator

IN function user provided function

MPI_USER_SCANC(inbuf, outbuf, len, type, 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 type data type of buffer

IN group group communicator

IN function user provided function

Chapter 4

Process Topologies

Rolf Hempel

4.1 Introduction

A process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In the majority of parallel applications this linear ordering does not reflect the communication structure of the processes which is usually determined by the underlying problem geometry and the numerical algorithm used. Often the processes are arranged in topological patterns as simple as a two– or three–dimensional grid.

For the user who programs an application on such a process structure the most natural way of referring to individual processes is to specify their coordinates in the process grid. Usually a process does not have to know the 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, if this helps to improve the communication performance on a given machine. How this mapping is done in detail is outside the scope of this standardization proposal, since it is too much dependent on the particular hardware. The description of a "virtual process topology", however, only depends on the application, and is, therefore, machine–independent. The functions which are proposed in this paper only deal with this side of the mapping problem.

There are well–known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. But even for more complicated graph structures good heuristics often yield near–optimal results [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. On some machines this will lead to unnecessary contention in the interconnect network. Some details about predicted and measured performance improvements which result from good process–to–processor mapping on modern wormhole–routing architectures can be found in [5, 4].

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. The inverse function MPI_INQLOC translates a pid number into the corresponding cartesian coordinates.

Discussion: In the PT2PT discussion it is not yet decided how processes are addressed. This could, for example, be done by unique identifiers, or by their ranks in a group. The topology handling will be adapted to whatever decision will be taken in PT2PT.

4.3 Embedding in MPI

The support for virtual topologies has to be fully integrated in the framework of MPI. In particular, the virtual topology functions do not interfere with the functions for process group creation and with those for collective communication. The current design of the process topology functions is based on the following assumptions:

- MPI is free to choose any process in a group for any location in the topology. There are no special processes which have to be aligned with topology substructures. This assumption fits best to a "local SPMD model" in which all processes in the same group execute the same program.
- One important feature of MPI are the process groups. The treatment of virtual topologies has to take this concept into account. The functions presented in this paper are not only compatible with the process groups, they even take advantage of them for the definition of mapping overlays and dynamic 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 virtual topology handling follows the general MPI rule of functional layering. At the bottom it is based on the general process group creation function MPI_CREATE_GROUP plus a few topology specific mapping functions. In principle this is sufficient for defining topologies and use them in applications.

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

4.4 Overview of the proposed MPI functions

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

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

   MPI_CREATE_GROUP(gid_old, gid_new, group_key, order_key) with

   IN gid_old group id of parent group,

   OUT gid_new group id of newly created subgroup of which the calling process is a member,

   IN group_key integer value used for the partitioning,

   IN order_key integer value used for determining the ranks in the new group (see below),

   This function is called by all processes in group *gid_old* loosely synchronously. It creates as many different subgroups as there are different values of *group_key*. The ranking in the newly created groups is determined by the *order_key* values, with the process ranks in the parent group for tie breaking.

2. If there is a standardized internal information cacheing facility, it can be used by MPI routines to store and retrieve information related to a process group. This facility was proposed in the Context Subcommittee. High–level topology functions store structure information for a process group, which is then later used by topology inquiry functions.

   In case the cacheing mechanism does not become part of MPI, the topology information can internally be stored with other group information in an implementation–dependent way, which each vendor can do as is most convenient.

There has been a general decision in MPI to start all numbering with 0 instead of 1. Thus, coordinates in a cartesian structure are 0,1, ..., dim-1. In the same way, nodes in a general graph are numbered zero up.

4.4.1 Low–level topology functions

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

For cartesian structures the function is

MPI_MAP_CART(gid_old, ndims, dims, period, newrank) with

IN gid_old group id of parent group,

IN ndims number of dimensions of cartesian structure,

IN dims(*) number of processes in each coordinate direction,

IN period(*) periodicity specification in each coordinate direction.

OUT newrank reordered rank of the calling process.

The corresponding function for general graph structures is

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

IN gid_old group id of parent group,

IN nodes number of graph nodes,

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

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

OUT newrank reordered rank of the calling process.

The definitions of the arguments `nodes`, `index`, and `nglist` are illustrated with the following simple example:

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

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

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

4.4.2 High–level topology functions

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

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

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

As specified for the low–level functions it is an error if one attempts to define a topology with more locations than there are processes in the parent process group. If it does not use all available processes, the unused process will return MPI_NORANK as their rank in the new group.

MPI_CART is defined as

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

IN gid_old group id of parent group,

OUT gid_cart group id of cartesian group (if calling process is part of it, else id of rest group),

IN ndims, dims(*), period(*) definition of cartesian structure, (for details see the definition of MPI_MAP_CART)

IN reorder ranking may be reordered or not,

OUT newrank rank of the calling process in the cartesian group (or MPI_NORANK).

Apart from some local computations, MPI_CART is based on the two function calls

MPI_MAP_CART (gid_old, ndims, dims, period, newrank)

and

MPI_CREATE_GROUP (gid_old, gid_cart, 0, newrank) .

For graph structures the corresponding routine is

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

IN gid_old group id of parent group,

OUT gid_graph group id of graph group (if calling process is part of it, else id of rest group),

IN nodes, index(*), nglist(*) definition of graph structure, (for details see the definition of MPI_MAP_GRAPH)

IN reorder ranking may be reordered or not.

OUT newrank rank of the calling process in the graph topology group (or MPI_NORANK).

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

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

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

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

4.4.3 Topology inquiry functions

If a topology has been defined with one of the above high–level functions, the topology information can be looked up using inquiry functions. They all are local calls, and do not require any communication.

The function MPI_INQMAP returns the type of mapping which is assigned to a group.

MPI_INQMAP(gid, map).

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

- MPI_CARTTOP: cartesian topology,
- MPI_GRAPHTOP: graph topology,
- MPI_NOTOP: no topology.

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

MPI_INQDIM(gid, dim) with

IN gid group id of group with cartesian structure,

OUT dim dimension of the cartesian structure.

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

IN gid group id of group with cartesian structure,

IN maxdim length of vectors size(*), period(*), and pos(*) in the calling program,

OUT size(*) number of processes for each coordinate direction,

OUT period(*) periodicity yes/no for each coordinate direction,

OUT pos(*) position of calling process in cartesian structure.

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

For a process group with cartesian structure, the function MPI_INQPID translates the logical process coordinates to process ids as they are used by the point–to–point routines:

MPI_INQPID(gid, location, pid) with

IN gid group id of group with cartesian structure,

IN location coordinate vector of process,

OUT pid process id.

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

MPI_INQLOC(gid, rank, location, maxdim) with

IN gid group id of group with cartesian structure,

IN rank rank of a process within group `gid`,

OUT location(*) coordinate vector of process,

IN maxdim length of vector `location` in the calling program (only for error checking).

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

Discussion: Depending on the decisions taken in the PT2PT subcommittee, MPI_INQPID might be changed to return the rank in the group rather than a pid.

4.4.4 Partitioning of cartesian structures

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

MPI_PARTC(gid, lostdim, gid_new) with

IN gid group id of group with cartesian structure,

IN lostdim direction perpendicular to resulting hyperplanes.

OUT gid_new group identifier of the hyperplane group in which the calling process is contained.

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

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

If the function MPI_SHIFT is called for a cartesian process group, it provides the calling process with the above identifiers, which then can be passed to the send–receive function. The user specifies the coordinate direction for the shift and the step length:

MPI_SHIFT(gid, direction, steps, pid_source, pid_dest) with

IN gid group id of group with cartesian structure,

IN direction coordinate direction of shift,

IN steps step length (> 0: upwards shift, < 0: downwards shift),

OUT pid_source process identifier of source process,

OUT pid_dest process identifier of destination process.

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

In a portable program the number of available processors usually is not known at compile time. In the case of a cartesian topology the function MPI_MAKDIM helps in setting the numbers of processes per coodinate direction, depending on the number of processes in the group and optional constraints which can be specified by the user:

MPI_MAKDIM(gid, ndims, dims) with

IN gid group id of group which is to be structured,

IN ndims number of cartesian dimensions,

IN/OUT dims number of processes in each coordinate direction.

The processes of the group with ID number `gid` are divided as evenly as possible among `ndims` 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 `dims(*)` is suitable for use as input to routine MPI_CART.

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_SENDC(...,neigh_pid(1),...) to send updated values to the left–hand neighbor (i-1,j). Note that there is only local communication in this exchange operation.

```
      integer ndims, num_neigh
      logical reorder
      parameter (ndims=2, num_neigh=4, reorder=.true.)
      integer gid, gid_cart, dims(ndims), neigh_def(2), neigh_pid(num_neigh)
      integer own_position(ndims), period(ndims), i, j, rank
      real*8 u(0:101,0:101), f(0:101,0:101)
      data dims / ndims * 0 /
C     Create a grid structure in ALL group and inquire about own position
      gid = MPI_ALL
C     Set process grid size and periodicity
      MPI_MAKDIM(gid, ndims, dims)
      period(1) = MPI_PER
      period(2) = MPI_PER
      MPI_CART (gid, gid_cart, ndims, dims, period, reorder, rank)
      MPI_INQCART (gid_cart, ndims, dims, period, own_position)
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) = i-1
      neigh_def(2) = j
      MPI_INQPID (gid, neigh_def, neigh_pid(1))
      neigh_def(1) = i+1
      neigh_def(2) = j
      MPI_INQPID (gid, neigh_def, neigh_pid(2))
      neigh_def(1) = i
      neigh_def(2) = j-1
      MPI_INQPID (gid, neigh_def, neigh_pid(3))
      neigh_def(1) = i
      neigh_def(2) = j+1
      MPI_INQPID (gid, neigh_def, neigh_pid(4))
C     Initialize the grid functions and start the iteration
      call init (u, f)
      do 10 it=1,100
         call relax (u, f)
C     Exchange data with neighbor processes
         call exchange (u, neigh_pid, num_neigh)
10    continue
      call output (u)
      end
```

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

Chapter 5

Language Binding

Scott Berryman
Ewing Lusk

5.1 Introduction

This chapter defines the rules for MPI language binding in general and for Fortran 77, Fortran 90, ANSI C, and ANSI C++ in particular. Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

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

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

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

5.2 Data Types and Naming Conventions

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

Handles are represented as integers in Fortran and as pointers to structures in C. Typed communication routines support **strings**, **double precision**, **real**, and **integer** data in Fortran. Null-terminated **strings**, **double**, **real**, **short**, **char**, and **long** are supported

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

Figure 5.1: This is an example of calling a routine with mismatched formal and actual parameters.

in C. In addition, a special **byte** type is supported in both languages. This type allows communication to be performed with no type conversion at all. In C, the definitions of named constants, function prototypes, and aggregate type information must be supplied in an include file `mpi.h`.

5.3 Fortran 77 Binding Issues

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

- Identifiers are limited to 31, not six, significant characters.
- Identifiers may contain underscores after the first character.
- The types of the formal parameter list may not match the calling parameter list. (An example is given in figure 5.3.)
- Although not required, it is strongly suggested that named constants be provided in an include file.

5.4 C Binding Issues

We use the ANSI C declaration format. All MPI names have an `MPI_` prefix, defined constants are in all capital letters, and defined types have one capital letter after the prefix.

5.5 Specific Bindings

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

5.5.1 C bindings for Point-to-Point Communication

These are in alphabetical order.

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

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

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

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

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

int MPI_associated(MPI_Handle handle, MPI_Handle handle_type )

int MPI_cancel( MPI_Handle handle, int flag )

int MPI_commit_buffer( MPI_Bd bd )

int MPI_create_status( MPI_Handle handle )

int MPI_create_buffer( MPI_Bd *bd, MPI_LIFETIME persistence )

int MPI_free( MPI_Handle handle )

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

int MPI_start( MPI_Handle handle )

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

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

int MPI_wait ( MPI_Handle handle, MPI_Status *status )

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

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

5.5.2 C Bindings for Collective Communication Chapter

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

```
int MPI_comm_init( MPI_Group group )

int MPI_rank( MPI_Group group, int rank )

int MPI_gsize( MPI_Group group, int size )

int MPI_barrier( MPI_Group group )

int MPI_synch( MPI_Group group, MPI_MAP map )

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.5.3 Fortran Bindings for Point-to-Point Routines

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

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

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

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

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

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

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

mpi_commit_buffer( bd, rc )
    integer bd, rc

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

mpi_start( handle, rc )
    integer handle, rc

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

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

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

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

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

5.5.4 Fortran Bindings for Collective Communciation Chapter

These are not done yet.

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

```
mpi_rank( group, rank, rc )

mpi_gsize( group, size, rc )

mpi_barrier( group, rc )

mpi_synch( group, map, rc )

mpi_bcast( buffer_handle, group, root, rc )

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

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

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

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

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

mpi_allscatter( list_of_inhandles, list_of_outhandles, group, rc )

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

mpi_allcast( inhandle, list_of_outhandles, group, rc )

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 6

Correctness Issues

Ewing Lusk

This chapter will evolve to include discussions and examples related to the issue of correct and safe MPI programs. It may eventually include a formal specification for the semantics of MPI routines. For the time being, in includes discussions and examples that have up to now appeared in other chapters.

6.0.5 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 send, 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 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.

6.0.6 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 at run time. Thus, this 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.

6.0.7 Collective Communications

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

Missing: This chapter should have a discussion of fairness for the MPI_WAIT_ANY function. Do we want MPI to ensure fairness? I propose not.

Chapter 7

Environmental Management and Inquiry

Bill Gropp

7.1 Introduction

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

7.1.1 Program Startup Issues

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

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

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

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

MPI_INIT

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

MPI_MAIN

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

Note that the `MPI_main` interface can be implemented with `MPI_call` through

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

MPI_CALL

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

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

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

All three of these approaches are currently in use.

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

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

7.2 MPI Program Startup Proposal–MPI_INIT

One goal of MPI is to achieve *source code portability.* By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine `MPI_init`.

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

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

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

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

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

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

7.3 MPI-Specific Enquiry Functions

These routines are specific to the operation of MPI.

The first two routines are absolutely essential; they are:

MPI_rank My process id relative to a group

MPI_size The number of process id's in a group

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

MPI_ValidTags The range (or other description) of valid tags

MPI_NumGroups Maximum number of groups

MPI_NumCntxs Maximum number of contexts

MPI_BufferSize Buffer space for unsent/unreceived messages

MPI_BufferManagement Buffer management type

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

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

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

MPI_BufferSizePairs Between pairs of processors

MPI_BufferSizeAll Between "me" and all others

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

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

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

There is also some desire for

MPI_IOmode Whether IO is per node or per group/context

MPI_ErrorMode Various choices of error handling

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

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

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

MPI_Has_Nonblocking Whether nonblocking operations are simulated with blocking sends and receives

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

In addition, it is useful to be able to unpack the various opaque objects into individual variables, such as buffer descriptors and contexts, and the message queue. Operations should include

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

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

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

MPI_UnpackContext Takes a context and returns members of group/context

MPI_StatusOfHandles I've forgotten what this is for.

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

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

7.3.1 Parallel programming

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

MPI_GetNbrs Return the "neighbors" in the underlying physical topology

MPI_GetPhysNode Return a unique specifier for the actual (as opposed to virtual) node. This should be a character string for maximum flexibility. ¿From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). A return value of "not applicable" is allowed but strongly discouraged.

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

MPI_GetDistance Return the "distance" from one processor to another

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

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

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

7.3.2 non-MPI

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

Interaction with signals

The intent of MPI is to be thread and signal safe. This has a number of subtle effects. For example, on Unix systems, a catchable signal such as SIGALRM (an alarm signal) must not cause an MPI routine to behave differently than it would have in the absence of the signal. Of course, if the signal handler issues MPI calls or changes the environment in which the MPI routine is operating (for example, consuming all available memory space), the MPI routine should behave as appropriate for that situation (in particular, in this case, the behavior should be the same as for a multithreaded MPI implementation).

A second effect is that a signal handler that does MPI calls must not interfere with the operation of MPI. For example, an MPI receive of any type that occurs within a signal handler must not cause erroneous behavior by the MPI implementation. Note that an implementation is permitted to prohibit the use of MPI calls from within a signal handler.

Chapter 8

Profiling

8.1 Requirements

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

1. provide a mechanism through which all of the MPI defined functions may be accessed with a name shift. Thus all of the MPI functions (which normally start with the prefix "`MPI_`") should also be accessable with the prefix "`PMPI_`".

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

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

8.2 Discussion

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

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

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

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

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified 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 = PMPI_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 = PMPI_EXAMPLE

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

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

Systems without weak symbols

In the absence of weak symbols then one possible solution would be to use the C macro pre-processor thus :–

```
#ifdef PROFILELIB
#    ifdef __STDC__
#        define FUNCTION(name) P##name
#    else
#        define FUNCTION(name) P/**/name
#    endif
#else
#    define FUNCTION(name) name
#endif
```

Each of the user visible functions in the library would then be declared thus

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

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

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

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

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

8.5 Multiple levels of interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation which would allow multiple levels of call interception, however we were unable to construct an implementation of this which did not have the following disadvantages

- assuming a particular implementation language.
- imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we decided to accept the scheme outlined above.

Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function.

Unfortunately such an implementation may require more co-operation between the different profiling libraries than is required for the single level implementation detailed above.

Chapter 9

Maps, Groups and Contexts

Lyndon Clarke, Mark Sears, Tony Skjellum, Marc Snir

9.1 Introduction

MPI provides support for the execution of **parallel procedures**. A parallel procedure is executed collectively by a set of communicating processes. Transfer of control to the parallel procedure, and back, is achieved by having each executing process transfer control to the local procedure code, and return from it. Not all processes need transfer control to the same parallel procedure; the parallel procedure may involve only a subset of processes, or different parallel calls may be executed by different subsets of processes. The collective communication calls provided by MPI are examples of such parallel procedures. Libraries such as parallel scientific libraries will be another example.

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

Another important goal is the support of a **loosely synchronous** transfer of control: no synchronization occurs either before or after the call. Thus, a process may start sending messages that pertain to the execution of a parallel procedure before all other participating processes have joined the execution; a process may receive a message that pertains to the execution of a parallel procedure, while participating in another parallel execution. A **communication context** mechanism is needed to distinguish communication that belongs to distinct parallel procedure executions. (Even if parallel transfer of control is executed synchroneously, one still needs a context mechanism for the synchronization calls themselves.)

Normally, a parallel procedure is written so that all messages produced during its execution are also consumed by the processes that execute the procedure. However, if one parallel procedure calls another, then it might be desirable to allow such call to proceed while messages are pending (the messages will be consumed by the procedure after the call returns). In such case, a new communication context is needed for the called parallel procedure, even if the transfer of control is synchronized.

A **context** is the MPI mechanism for partitioning communication space. A send made in a context cannot be received in another context. Contexts are identified in MPI using integer-valued **context_id**'s.

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

MPI does not provide for absolute process names. Rather, processes are always identified by their rank inside a group. A universal group that includes all processes available when computation starts is the nearest equivalent to an absolute process name space in MPI. Relative naming is consistent with MPI implementations where processes can be added dynamically.

New process groups are built by subsetting and reordering processes within existing groups (as defined by communicators). A new group is defined from an old group by specifying the correspondence between the rank of each process in the new group and its rank in the old group. Such correspondence is called a **map**; it is represented in MPI by an opaque object.

9.2 Maps

A **map** is a one-to one-mapping from $0...m-1$ into the set of non negative integers; m is the **size** of map f. A possible representation for such map is a list of m integers; it is often convenient to think of a map as being such list. Another possible representation is as an algorithmic specification of the map (e.g., hash function). A map is represented by an opaque map object.

Maps are used in order to represent groups. If processes are numbered from 0 to $n-1$, then a map $f: 0...m-1 \rightarrow 0...n-1$ represents a subset group of m processes: process $f(i)$ has rank i in the new group. A map, by itself, does not represent a group; it does so only relative to a pre-defined containing group, i.e., relative to another ranking of the processes.

Discussion: Should decide if opaque objects are justified, or whether an array of indices is OK. On the "opaque object" side: May have algorithmic map definitions and may spread the definition over more than one node, on a very large system. On the explicit representation side: The representation is simple, we avoid an additional object, and there is no much waste. Also, with an explicit representation, the user can easily add his/her own map constructors, and building communicators is faster. Need to understand what an opaque map would be in Fortran.

If maps are explictly represented as arrays, then one needs map size to be an additional parameter.

For each `m`, an identity map `MPI_IDENT(m)` of size `m` is predefined: `MPI_IDENT(m)`$(i) = i$, $i = 0, ..., m-1$.

Discussion: If maps are non opaque then the ident map is just a long array with i stored at the i-th entry, and it is just `MPI_IDENT`. If maps are opaque, we need a different map for each domain size (a function from integers to maps?) Need to think of Fortran implementation.

9.2.1 Operations on maps

The following operations are defined on maps. Each function is invoked locally by a process.

MPI_MAP_SIZE(map, size)

IN map handle to map object

OUT size size of map (integer)

MPI_MAP_APPLY(map, argument, value)

IN map handle to map object

IN argument map argument (integer)

OUT value image of `argument` under `map` (integer)

MPI_MAP_INVERSE(map, argument, value)

IN map handle to map object

OUT argument inverse image of `value` under `map`; -1 if `value` is not in the range of `map`.

IN value map value (integer)

MPI_MAP_LIST(map, len, array_of_integer, size)

Creates an explicit list representation of a map.

IN map handle to map object

IN LEN length of array (integer)

OUT array_of_integer list of values in the range of `map`

OUT size length of returned list – size of map (integer).

Discussion:
I changed terminology from “element, rank” to “argument, value” since both argument and value are ranks.

9.2.2 Map constructors

The map constructors may be either local or may require communication.

MPI_MAP_BUILD(map, array_of_integer, size)

Build a map from an explicit list representation. This function is called locally by one process.

OUT map handle to map object

IN array_of_integer list of values in the range of `map`

IN size number of entries in array – map size (integer)

MPI_MAP_SPLIT(comm, map, key, index, newmap)

This is a collective function that is called by all processes in the group associated with `(comm, map)`. All calling processes provide the same values for the parameters `comm` and `map`. A separate group of processes is formed for each distinct value of `key`, with the processes ordered according to the value of `index`. Each process in such group is returned in `newmap` the map of this group within `comm`.

IN comm communicator object handle

IN map map object handle

IN key (integer)

IN index (integer)

OUT newmap map object handle

Additional map creation functions may be provided. Follows a possible list constructors.

Discussion: Do we want to mandate them in MPI?

MPI_MAP_UNION(map1, map2, newmap)

`map1` and `map2` are maps with disjoint ranges. The resulting concatenation is a map with a range which is the union of the ranges of `map1` and `map2`, with elements in the range of `map1` listed first, and original order preserved within each range. The union operation is associative but not commutative.

$$\texttt{newmap}(i) = \begin{cases} \texttt{map1}(i) & \text{if } i < size(\texttt{map1}) \\ \texttt{map2}(i - size(\texttt{map1})) & \text{otherwise} \end{cases}$$

IN map1 map object handle

IN map2 map object handle

OUT newmap map object handle

MPI_MAP_PRODUCT(map1, map2, range, newmap)

`map1` is a map with values in the range $0, ..., \texttt{range} - 1$. `newmap` is the cartesian product of `map1` and `map2`, which maps the pair (i, j) into the pair $(\texttt{map1}(i), \texttt{map2}(j))$. Pairs in the domain and in the range of this mapping are numbered in row major order, i.e., pair (i, j) has number $i \cdot \texttt{range} + j$. The product operation is associative but not commutative.

$$\texttt{newmap}(i \cdot \texttt{range} + j) = \texttt{map1}(i) \cdot \texttt{range} + \texttt{map2}(j)$$

IN map1 map object handle

IN map2 map object handle

OUT newmap map object handle

9.3 Context_id

A **context_id** is an integer. The range of valid values for `context_id` is implementation dependent, and can be found by calling a suitable query function, as described in 7.

9.3.1 Operations on context_id's

MPI_CONTEXT_ALLOCATE(comm, map, array_of_contextids, len)

Allocates an array of context_id's. This is a collective operation that is executed by all processes in the group defined by `(comm, map)`. The context_ids that are returned are unique within the group associated with `(comm, map)`. The array returned is the same on all processes that call the function (same order, same number of elements). The call may block until all processes within the executing group have invoked the call.

IN comm communicator object handle

IN map map object handle

OUT array_of_context_ids (array of integers)

IN len number of context_id's to allocate (integer)

MPI_CONTEXT_RESERVE(context_id)

Reserve a context_id value. This is a local call that is invoked by a single process. A reserved context_id value will not be allocated by a subsequent call to `MPI_CONTEXT_ALLOCATE` by the same process. It is erroneous to reserve a context_id that has been already been allocated by `MPI_CONTEXT_ALLOCATE`.

IN context_id (integer)

Discussion:
May want to reserve an array of values, rather than one value.

Implementations may have there own reserved values. The effect is as if calls to MPI_CONTEXT_RESERVE where executed in a preamble.

MPI_CONTEXT_FREE(context_id)

Free a reserved context_id value. The value becomes available for a subsequent reservation by MPI_CONTEXT_RESERVE or subsequent allocation by MPI_CONTEXT_ALLOCATE. It is erroneous to free a value that is associated with an active communicator.

IN context_id (integer)

Discussion:
Here, too, we might prefer to free an array of values, rather than one.

Communicators, like all opaque objects, are freed by a call to MPI_FREE. We may prefer to have that same call free also the associated context_id. With the current design, if we want to free both communicator and context_id we need to store the context_id value, free the communicator, next free the context_id.

9.4 Communicators

A **communicator** is an opaque object that identifies a group of processes and a communication context for that group. Like other opaque objects, communicators cannot be transfered between processes; context_id's are used to transfer information on context.

As a short-hand, we shall identifiy the group of processes associated with a communicator comm with the communicator itself. Thus, "the process with rank i in comm" should be understood as meaning " the process with rank i in the group associated with comm". In the same manner, "the communication context comm" should be understood to mean "the communication context associated with comm".

An initial communicator MPI_COMM_INIT is defined when the program starts. Its associated group contains all processes that start the computation. Applications that do not need multiple process groups or multiple contexts, will only use this communicator.

Let comm be a communicator that is associated with a group of size n, and let map : $0..m-1 \rightarrow 0..n-1$ be a map. Then the pair (comm, map) defines a subgroup, namely the subgroup of processes with ranks map$(0), ...,$ map$(m-1)$ in comm.

9.4.1 Operations on communicators

MPI_COMM_MAP(comm, subcomm, map)

Returns a map such that (comm, map) is the group associated with subcomm; i.e., if i is the rank of a process in the group associated with subcomm, then map(i) is the rank of that same process in the group associated with comm. The call is erroneous if subcomm has a process that is not member of comm.

IN comm communicator object handle

IN subcomm communicator object handle

OUT map map object handle

The `MPI_COMM_MAP` function retrieves the group of processes that is associated with a communicator. However, since absolute process names are not visible in MPI, the group can only be defined relative to another encompassing group. The "absolute" process number is obtained by using `MPI_COMM_INIT` as a reference communicator.

MPI_COMM_SIZE(comm, size)

Returns the size of the group associated with `comm`.

IN comm handle to communicator

OUT size group size (integer)

MPI_COMM_CONTEXTID(comm, context_id)

Returns the context_id associated with the communicator `comm`.

IN comm communicator object handle

OUT context_id context_id

9.4.2 Communicator constructors

MPI_COMM_MAKE(comm, map, context_id, newcomm)

Creates a new communicator object, which is associated with the group defined by `(comm, map)`, and the context_id `context_id`. This is a local call executed by one process. However, the new communicator object should not be used for communication between two processes unless they both have called the function.

It is erroneous to create on the same process two distinct communicators with the same context_id.

IN comm communicator object handle

IN map map object handle

IN context_id context_id

OUT newcomm communicator object handle

The contect_id used in this call may be an integer returned by a previous call to `MPI_ALLOC_CONTEXT`, executed within the same group. In such case, it is guaranteed that the context_id is unique, and has not been used already to create a communicator. However, the user is free to manage on it own the context_id's, and use other mechanisms for their allocation. For example, one could have a single **context_id server** generate unique context_id's for the entire system; one can preallocate statically some context_id values, for the use of libraries; etc.

Discussion:

Do we want to prohibit implementations that synchronize the creation of communicators (i.e., where a call to `MPI_MAKE_COMM` blocks until all members of the group have made the call)?

Is it preferable to have a `array_of_comm` as parameter, rather than one communicator?

We don't have now the ability to have two different groups with the same context; e.g., two different numberings of the processes, used with the same communication context. The alternative is to remove the restriction that two distinct communicators cannot use the same context_id. It is now the user responsibility to disambiguate these two communicators; the system does not guarantee that a message sent with one communicator can be received only with this same communicator, if they use the same context_id. We save on context_id's but loose safety (the sender and receiver may be using different process numberings, if two different communicators are used – the receiver may not identify correctly the sender).

9.5 Working without context_id's

It is possible to create communicators directly, without using context_id's. A single function combines the allocation of a new context_id and the generation of a new communicator. This allows a "naive" user that does not need to customize context_id allocation to ignore this MPI feature. Direct communicator creation is safer, since uniqueness of context_id's is guaranteed, by construction.

MPI_COMM_SAFEMAKE(comm, map, newcomm)

Creates a new communicator (with an attached context_id) that is associated with the group defined by the pair `(comm, map)`. This is a collective call that has to be invoked by all members in this group. All of them are providing the same input parameters `comm` and `map`. The call may block until all processes in the group have invoked it. The new communicator may be safely used for communication with any member of the new group, once the call returns.

IN comm communicator handle

IN map map handle

OUT newcomm communicator handle

`MPI_COMM_SAFEMAKE` is

```
MPI_CONTEXT_ALLOC(comm, map, context_id, 1);
MPI_COMM_MAKE(comm, map, context_id, newcomm);
``MPI_SYNCH(comm,map)'' /* by which we mean a synchronization
                     operation in the subgroup defined by (comm,map) */
```

Discussion:

Do we prefer to create an array of communicators, rather than one? (To be consistent with the `MPI_CONTEXT_ALLOC` function).

`MPI_SYNCH(comm,map)` is not currently defined in the collective communication library; it can be coded using point-to-point calls, but seems important enough to be included there.

General thought – if we support collective communication calls with a map as an additional parameter, then we have disassociated the "context" of a collective communication, from its supporting "group". We need to do it when we build new communicators, in order to bootstrap – if we want the call that generates a new communicator to involve only the processes in the new group.

9.6 Working without maps and context_id's

It is possible to avoid both the use of map objects and of context_id's altogether. A single function combines the generation of a map, the allocation of a context_id and the generation of a new communicator. This allows "naive" users to use MPI without being exposed to maps or context_id's.

MPI_COMM_BUILD(comm, array_of_ranks, size, newcomm)

Build a map from an explicit list representation. This is a collective function that is called by all members of the new group. All provide the same parameters for `comm`, `array_of_ranks` and `size`. The call may block until it was invoked on all the group members. When it returns, the new communicator may be used to communicate with any member of the new group.

IN comm handle to communicator

IN array_of_ranks list of values in the range of `map`

IN size number of entries in array – map size (integer)

OUT newcomm handle to new communicator

`MPI_COMM_BUILD` is

```
MPI_MAP_BUILD(comm, array_of_ranks, size, newmap);
MPI_COMM_SAFEMAKE(comm, newmap, newcomm);
```

MPI_COMM_COPY(comm, newcomm)

Creates a new communicator with the same group as the old communicator. This is a collective function that is called by all members of new group. All provide the same parameters for `comm`. The call may block until it was invoked on all the group members. When it returns, the new communicator may be used to communicate with any member of the group.

IN comm handle to communicator

OUT newcomm handle to new communicator

`MPI_COMM_COPY` is

```
MPI_COMM_SIZE(comm, size);
MPI_COMM_SAFEMAKE(comm, MPI_IDENT(size), newcomm);
```

MPI_COMM_SPLIT(comm, key, index, newcomm)

Split the group associated with `comm`; creates a new group for each distinct value of `key` that contains the processes that supplied that key value; the processes are ranked according to the `index` values they supplied. A new communicator is created for each subgroup. Each process is returned a handle to the communicator for the new subgroup it belongs to. This is a collective call executed by all processes in the group associated with `comm`. They call may block until all processes have invoked the function. When the call returns the new communicator may be safely used for communication in the new group.

IN comm handle to communicator

IN key (integer)

IN index (integer)

OUT newcomm handle to new communicator

`MPI_COMM_SPLIT(comm, key, index, newcomm)` is

```
MPI_COMM_SIZE(comm, size);
MPI_MAP_SPLIT(comm, MPI_IDENT(size), key, index, newmap);
MPI_COMM_SAFEMAKE(comm, newmap, newcomm);
```

Discussion:
May want a special DONTCARE key value that indicates that the caller need not have a new communicator

9.7 Examples

We say that a parallel procedure is *active* at a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active at a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

9.7.1 Nonreentrant parallel procedures

This covers the case where, at any point in time, at most one invokation of a parallel procedure can be active at any process. I.e., concurrent invokations of the same parallel procedure may occur only within disjoint groups of processes. For example, all invokations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invokations.

In such a case, a context_id can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. Or, it can be done a compile/link time, if the implementation has additional mechanisms to reserve context_id values. Communicators to be used by the different procedures can be build in a preamble, if the executing groups are statically defined; if the executing groups change dynamically, then a new communicator has to be built whenever the executing group changes, but this new communicator can be built using the same preallocated context_id. If the parallel procedures can be organized into libraries, so that only one procedure of each library can be concurrently active at each processor, then it is sufficient to allocate one context per library.

9.7.2 Parallel procedures that are nonreentrant within each executing group

This covers the case where, at any point in time, for each process group, there can be at most one active invokation of a parallel procedure by a process member. However, it might be possible that the same procedure is concurrently invoked in two partially (or completely) overlapping groups. For example, the same collective communication function may be concurrently invoked on two partially overlapping groups.

In such a case, a context_id is associated with each parallel procedure and each executing group, so that overlapping execution groups have distinct communication contexts. (One does not need a different context_id from each group; one merely needs a "coloring" of the groups, so that One can generate the communicators for each parallel procedure when the execution groups are defined. Here, again, one only need one context for each library, if no two procedures from the same library can be concurrently active in the same group.

Note that, for collective communication libraries, we do allow several concurrent invokations within the same group: a broadcast in a group may be started at a process before the previous broadcast in that group ended at another process. In such a case, one cannot rely on context mechanisms to disambiguate successive invokations of the same parallel procedure within the same group: the procedure need be implemented so as to avoid confusion. E.g., for broadcast, one may need to carry additional information in messages, such as the broadcast root, to help in such disambiguation; one also relies on preservation of message order by MPI. With such an approach, we may be gaining performance, but we loose modularity. It is not sufficient to implement the parallel procedure so that it works correctly in isolation, when invoked only once; it needs to be implemented so that any number of successive invokations will execute correctly. Of course, the same approach can be used for other parallel libraries.

9.7.3 Well nested parallel procedures

Calls of parallel procedures are well nested if a new parallel procedure is always invoked in a subset of a group executing the same parallel procedure. Thus, processes that execute the same parallel procedure have the same execution stack.

In such a case, a new context need to be dynamically allocated for each new invokation of a parallel procedure. However, a stack mechanism can be used for allocating new contexts. Thus, a possible mechanism is to allocate first a large number of context_id's (up to the upper bound on the depth of nested parallel procedure calls), and then use a local stack management of these context_id's on each process to create a new communicator (using

MPI_COMM_MAKE) for each new invokation.

Discussion: General case

In the general case, there may be multiple concurrently active invokations of the same parallel procedure within the same group; invokations may not be well nested. A new context need to be created for each invokation. It is the user responsibility to make sure that, if two distinct parallel procedures are invoked concurrently on overlapping sets of processes, then context_id allocation or communicator creation is properly coordinated.

9.8 Left

Remote procedure call and C2 objects.

How to handle growing process set.

Chapter 10

Initial Implementation Subset

Steven Huss-Lederman

10.1 Introduction

This chapter defines a minimal subset of MPI for initial implementation. This subset is being defined so that consistent implementations can appear more rapidly. It was recognized early in the process that MPI needed to appear as quickly as possible and practical. The creation of a subset will hopefully allow users earlier access to the standard and still allow for the writing of portable message passing codes.

This subset should not in any way be construed as a limitation on MPI implementations. It is strictly the minimum necessary to have an initial MPI subset conforming implementation. It is hoped that an officially sanctioned subset will encourage and allow implementors to introduce MPI in a more timely fashion. It should be noted, however, that implementation of the subset does not make an implementation conform with MPI. The subset is only a potential first step in the process. All implementations must ultimately conform to the entire standard; implementations are encouraged to do the full standard as rapidly as possible.

The subset presented is consistent with the complete MPI standard. This was an important goal so the additional MPI features could be added without changing any functionality from the user's perspective. Thus, users can be assured that programs written now for the subset will run without modification under the full MPI standard. Furthermore, using additional features of the full MPI standard in the future will not require changes to code already written. Users may use MPI features outside this subset that are offered by various implementors. However, people who require portability during the early development of MPI may experience some difficulties until later in the development process.

10.2 Criteria and Rational

Having the subset be consistent with the entire MPI standard was considered critical to the effort. In addition to this, there were many possible criteria to use in determining the elements of MPI to include in the subset. The main criteria used were that the MPI subset should

1. contain routines that are as close as possible to current standard practice to minimize the effort to port codes.

2. contain as many new and important MPI features as possible.

3. allow developers to be able to have the subset show up as rapidly as possible while still meeting the other criteria.

There were many rationales for these criteria. It was recognized that it is impossible to come up with an ideal subset and compromises were necessary. It was felt that current users should be comfortable in migrating to MPI. Thus, the subset should contain the message passing elements that represent common practice today. Furthermore, it was felt that the initial (and later) version of MPI should maintain efficiency. Otherwise users could suffer from disenchantment with MPI on their first usage and may never give the standard a second try. Having the standard routines in the subset should encourage developers to optimized these routines in the initial version of MPI. Also, though some felt that the portability offered by the MPI subset was sufficient to entice users, many felt that the subset should contain some new feature(s) over and above current common practice as an added incentive. The additional items selected were meant to represent some of the significant new features of MPI. Balanced against these first two goals is the need for the subset to show up in a timely fashion. Thus, each feature chosen for inclusion in the subset was deemed of sufficient importance to outweigh its added complexity in implementing the subset. Though some functions seem easy to implement there are often overlooked costs in testing, documentation and development that was considered before a feature was added to the MPI subset. Inclusion of too many routines might lead to moderately efficient implementation of them all instead of a very efficient and possibly more useful implementation of a smaller subset.

Discussion: Some felt that we should aim the subset at typical users. Should this be another criteria? Is the compatibility with current practice enough?

Discussion: Should it be a criteria that the subset and common practice features be implemented efficiently. The text touches on this but it is not a specific item. I did not give it a bullet since it seems like a quality of implementation issue. Poorly implemented subsets will be forced out by the market.

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

10.3 Subset Functionality

Discussion: This section only discuss the pt-to-pt and collcomm chapters. The other chapters will be added after they are read at meetings and finalized.

10.3.1 Itemized Functionality

The Table below summarizes the specific functionality included and excluded from the subset.

Section	Operation	Included	Excluded
		Point to Point Communication	
1.5.2	Buffer Operations	"All" functions	Multiple appends, h versions(?)
1.7	Comm. Mode	Standard	Ready, Synchronous
1.8.1	Comm. Object Creation	All(?)	
1.8.2	Comm. Start	START(if have INIT)	
1.8.3	Comm. Completion	All	
1.8.4	Multiple Completion	All	
1.9	Blocking Comm.	Std. version	Ready, Synch. version
1.10	Nonblocking Comm.	Std. version	Ready, Synch. version
1.11	Contiguous Buffer Comm.	Std. version	Ready, Synch. version
1.12		PROBE	CANCEL,GET_LEN
		Collective Communication	
2.4	Synchronization	All	
2.5	Data Move	All(?)	
2.6	Global Compute	MPI Ops	User Ops

All other general functionality is included in the subset. Thus, all matching criteria, order, etc. are part of the subset.

10.3.2 Point to Point Functionality

The operations involving contiguous buffers are included since they represent standard practice today. In addition, the communications buffers are also included because they offer an important new feature in MPI. However, to simplify the implementation, the communication buffer will be limited to one component, e.g., only one append is allowed. This limitation will still allow the user access to the strided and indexed cases but require more complex cases to be done by the user.

Discussion: If we eliminated communication buffers then the number of routines would be greatly reduced. We would only have the "C" routines in collective communications and contiguous in pt-2-pt. Right now I think we should leave it in but I point this out

Discussion: The heterogeneous cases (hvector, hindexed) seem very easy to code but are another set of functions. They allow elements of a C type structure to be pulled out in the subset. It has been pointed out that even simple routines have a cost of testing, documentation, etc. that can easily outweigh the coding cost.

Of the three types of communication modes, the STANDARD mode represents standard practice and is included. The other two modes are not used as commonly and therefore excluded. This applies to all of the point-to-point communications routines.

The common versions of blocking and nonblocking send and receive are included. In addition, the low level routines of INIT and START are included. These should help users who wish greater control over communications for optimization purposes. For example, this would allow for the reuse of a persistent handle.

Discussion: Using INIT, START, COMPLETE, FREE is not what most people do today but does allow for greater speed. It seems as if you would have this in a system anyway and it would be useful to the user. Should we include it?

10.3.3 Collective Communication Functionality

The collective communication routines have the same general restrictions as the point to point routines. Therefore, the buffer descriptors will be limited to one entry.

A collective communication operates on a group. This defines the scope of the operation. Many current users only operate on all the processes. This is the MPI_ALL group in MPI. To simplify the subset, groups will be limited to be MPI_ALL.

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

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

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.