ORNL/TM-12231

Engineering Physics and Mathematics Division

Mathematical Sciences Section

A PROPOSAL FOR A USER-LEVEL, MESSAGE-PASSING INTERFACE IN A DISTRIBUTED MEMORY ENVIRONMENT

Jack J. Dongarra [‡§]
Rolf Hempel [¶]
Anthony J. G. Hey [†]
David W. Walker [§]

[‡] Department of Computer Science
107 Ayres Hall
Knoxville, TN 37996-1301

[†] Department of Electronics and Computer Sciences
University of Southampton
Southampton, SO9 5NH
United Kingdom

[¶] Gesellschaft für Mathematik und Datenverarbeitung mbH
P. O. Box 1316
D-5205 Sankt Augustin 1
Germany

[§] Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

Date Published: February, 1993

Research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy, by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office, and by the Center for Research on Parallel Computing.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

A PROPOSAL FOR A USER-LEVEL, MESSAGE-PASSING INTERFACE IN A DISTRIBUTED MEMORY ENVIRONMENT

Jack J. Dongarra [‡§]
Rolf Hempel
Anthony J. G. Hey
David W. Walker

Abstract

This paper describes Message Passing Interface 1 (MPI1), a proposed library interface standard for supporting point-to-point message passing. The intended standard will be provided with Fortran 77 and C interfaces, and will form the basis of a standard high level communication environment featuring collective communication and data distribution transformations. The standard proposed here provides blocking and nonblocking message passing between pairs of processes, with message selectivity by source process and message type. Provision is made for noncontiguous messages. Context control provides a convenient means of avoiding message selectivity conflicts between different phases of an application. The ability to form and manipulate process groups permits task parallelism to be exploited, and is a useful abstraction in controlling certain types of collective communication.

1. Introduction

This paper documents a proposal, initially made in November 1992, for a standard for performing point-to-point message passing between pairs of processes in a MIMD distributed memory computing system. Some modifications were made in January 1993, particularly in the approach to process groups, following input from a number of colleagues. An effort is currently underway to develop a more comprehensive standard for message-passing on distributed memory systems by July 1993. This effort involves a team of about 60 people made up of hardware and software vendors, and researchers from universities and government laboratories.

A small set of typed message passing routines form the core of the standard, and are augmented by support for features such as noncontiguous messages, communication contexts, and process groups. The proposed standard, called Message Passing Interface 1 (MPI1), includes only message passing between distinct pairs of processes, and thus does not address collective communication of any type, including broadcasts and reduction operations. We expect these types of communication will be included in a higher level standard to be defined subsequently. Other important standardization issues not addressed in detail include support for virtual communication channels, active messages, heterogeneous computing, performance tracing, and parallel I/O. Thus, while MPI1 does not at this stage provide the flexibility and range of functionality that one would expect from a complete message passing environment, we regard it as forming the core of such an environment. In designing MPI1 we have tried to avoid imposing constraints that would hinder the future extensions necessary to address the issues mentioned above.

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 built 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.

In designing MPI1 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, MPI1 has been strongly influenced by work at the IBM T. J. Watson Research Center by Bala, Kipnis, Snir and colleagues [2, 3], Intel's NX/2 [18], Express [17], nCUBE's Vertex [15], and PARMACS [11, 13]. Other important contributions have come from Zipcode [19, 20], Chimp [6, 7], PVM [8, 21], and PICL [9].

One of the objectives of this paper is to promote a discussion within the concurrent computing research community of the issues that must be addressed in establishing a practical, portable, and flexible standard for message passing. This cooperative process began with a workshop on standards for message passing held in April 1992 [22], and continued with a second meeting in November 1992 when an organizational structure for developing a standard message passing interface was created. We believe the draft of MPI1 proposed here provides a good, concrete basis for continued discussion, and that it will contribute over the next few months to the development of an intermediate level message passing standard.

In Section 2 the rationale for an intermediate level standard is given. Section 3 presents the programming model assumed, and describes the main features of MPI1. Section 4 discusses the main decisions and compromises made in designing MPI1. Some important unresolved issues that must be addressed before MPI1 can be regarded as complete are presented in Section 5. These include support for application topologies and heterogeneous computing, and a more general approach to process groups. Finally, Section 6 presents concluding remarks, and solicits involvement from the research community in the development of a standard for a comprehensive message passing interface. Detailed specifications of the MPI1 routines are given in Appendix A in the form of manual pages.

2. General Overview

It is possible to consider defining a message passing standard at a number of levels. At the lowest level, closest to the hardware, might be syntactically simple routines for moving packets along wires. Above this channel-addressed level might be a process-addressed level (where there may be more than one process on each physical processor), such as that defined by NX or Vertex on the iPSC and nCUBE machines, the commercially-available *Express* communication environment, or the PARMACS message passing macros. Higher-level abstractions, for example, Linda [4, 10], MetaMP [16], or Shared Objects [1, 14], would lie above this level. Each level could be built using the level beneath, provided that the overhead in doing this was sufficiently low that the cumulative overhead incurred at the higher levels was small. These successive software levels form a series of layers, that with some stretch of the imagination resemble the multiple skins of an onion, with the hardware being at the center. We, therefore, call this the "Onion Skin Model" of the distributed communication environment. In deciding at which level to try to impose a standard it should be noted that different people might favor different types of standard. For example, a non-expert user would prefer to use high-level abstractions, such as virtual shared memory, so that details of the message passing are hidden. On the other hand, a compiler writer would like to produce a portable parallel compiler, and would like to use small, fast messages such as might be provided by a low-level standard. Finally, an expert application developer might be prepared to sacrifice some ease-of-use for additional speed, and so would prefer a intermediate level standard that provides a set of efficient primitives for point-to-point message passing. The standard proposed here is intended for use by such an application developer.

If the Onion Skin model is valid, then it makes sense to impose a standard that is also layered. However, the hardware of different distributed memory computing systems is sufficiently varied that it is difficult to impose a low-level standard that is efficient on all machines. Therefore, it is more appropriate to define a standard at an intermediate level, and to implement this as efficiently as possible on each machine. There is still the possibility of defining higher-level standards on top of this intermediate level. Thus, the intermediate-level standard will be open and extendable. It is the standardization of this intermediate level of point-to-point message passing between pairs of processes that is the focus of this paper.

3. Features of the Standard

Our programming model assumes some set of processes that communicate by point-to-point message passing. With each process is associated some memory directly accessible only by that process – there is no shared memory. In MPI1 it is assumed that processes are single threaded, though we expect the final MPI standard to permit multithreaded processes. Although the message passing paradigm is usually associated with distributed memory systems, it is not necessary to make any strong assumptions about the underlying hardware. The proposed message passing standard could also be implemented on shared memory machines and uniprocessor workstations. Note that the standard does not address the issue of how the processes are assigned to physical processing nodes. In general, this issue requires the development of machine-dependent static and dynamic load balancers, and lies outside the scope of the proposed standard.

MPI1 provides some support for task parallelism. To this end each process is assumed to be a member of one or more process groups, each of which is identified by a unique process Group ID number, or *GID*. The processes in a group can cooperate to perform tasks completely independently of other processes, and in this sense each group can behave like a distinct virtual machine. The concept of process groups is also important when designing collective communication routines.

3.1. Basic Message Passing Routines

We now introduce the basic message passing routines that form the core of the proposed standard. These routines permit point-to-point message passing between pairs of processes, with message selectivity based explicitly on message type and source process, and implicitly on communication context. Communication contexts are explained in more detail in Section 3.3.

MPI1 provides three modes for sending and receiving messages: blocking, nonblocking, and synchronized. These different communication modes are explained below. The mode is passed as an argument to the send or receive routine. A nonblocking or blocking send routine may be matched by a nonblocking or blocking receive routine in any combination. However, a synchronized send must be matched by a synchronized receive.

Noncontiguous messages are handled by providing three variants of the send and receive routines. The first variant assumes contiguous messages, and MPI1 provides the routines

MPI_CSEND and MPI_CRECV for such messages. The second deals with messages that are gathered from, or scattered to, a buffer with constant stride. This type of routine may be used when communicating rows of a distributed matrix that is stored by columns. The routines MPI_SSEND and MPI_SRECV are used in this case. The third variant deals with messages that are gathered from, or scattered to, a buffer in an arbitrary way. MPI1 provides the routines MPI_GSEND and MPI_GRECV for this purpose. This last case provides a mechanism for doing point-to-point scatter/gather operations between pairs of processes. The data blocks comprising the message may be of differing sizes and lie at arbitrary locations in the buffer gathered from or scattered to. The scatter/gather operations are controlled by a pair of arrays. The first of these arrays contains pointers into a buffer that indicates where the data for the message is coming from, or going to. The second array indicates how many data items are to be extracted from, or stored to, each location pointed to. For example, suppose in some spatially decomposed particle simulation we build a list of the particles that must be migrated to another process in each time step. This list is a set of indices into the data structure containing the particle information. The Fortran language requires that the scatter/gather locations be specified by an indirection vector that applies to a specific buffer. The C language permits pointer manipulation, so the memory location from which data are gathered, or to which data are scattered, can be more naturally expressed as an array of pointers. This is one of the few significant syntactic differences between the C and Fortran versions of MPI1.

3.1.1. Receiving messages

The receipt of a message is said to be blocking if the receiving process suspends execution until all of the message has been received, i.e., until it has been placed in an application buffer on the receiving process. If a process attempts to perform a blocking receive that is not matched by a corresponding loosely synchronous send, execution will be suspended indefinitely on that process, resulting in full or partial deadlock.

A nonblocking receive takes place in two phases. First, a receive is posted on the receiving process, that is, the application provides a buffer that is to be used to store a specified incoming message. After this the receiving process can then continue to do useful work. However, at this stage receipt of the message is not guaranteed, and the data in the message should not yet be used by the receiving process. The nonblocking receive must be completed in a second phase that either calls the routine MPI_WAIT that blocks until the message is received, or periodically calls the routine MPI_STATS that checks on whether the message has been received into an application buffer. Between these periodic checks useful work can continue to be done by the receiving process, and once receipt is confirmed the message may be processed. Using the blocking mechanism (MPI_WAIT) to complete a nonblocking receive is usually done immediately before the message is to be used on the receiving process, thereby allowing the maximum potential for overlap of computation and communication. This approach is common when the amount of work that could be done between posting the receive and actually using the received data can be quantified at compile time. In more dynamic situations there may be an almost arbitrary amount of work that a process could do until an anticipated message arrives. In such cases it is common to periodically check for message receipt using MPI_STATS. At the application level, a blocking receive is conceptually the same as a nonblocking receive in which no useful work is done between the two phases, i.e., a call to an MPI1 receive routine in nonblocking mode immediately followed by a call to MPI_WAIT.

When a message is received in synchronized mode, the receiving process sends an acknowledgment to the sending process once the message has been completely received and placed in an application buffer. In the absence of hardware failures, and provided valid arguments are passed to the send and receive routines, message receipt is guaranteed.

3.1.2. Sending messages

The sending of a message is said to be blocking if the sending process suspends execution until all of the message has been sent, i.e., until the application buffer containing the message on the sending process is available for reuse. When this has occurred we say that "the message has cleared the buffer." It is not guaranteed that the message will actually be delivered to the destination process, and unless the application performs some additional handshaking, the sending process cannot know if the message was delivered.

A nonblocking send takes place in two phases. In the first phase the user calls an MPI1 send routine in nonblocking mode which initiates transmission of a specified message buffer to the destination process, and then returns. The sending process can then continue to do useful work, but during this time it is not guaranteed that the message has cleared the buffer, and it is a programming error to change it in any way. The nonblocking send must be completed in a second phase that either calls the routine MPI_WAIT that blocks until the message has been sent, or periodically calls the routine MPI_STATS that checks on whether the message has been sent or not. Between these periodic checks useful work can continue to be done by the sending process, and once the message has been sent the message buffer may then be safely modified. The routine MPI_STATS may be used to check for completion of a nonblocking send when there is an arbitrary amount of work that can be done between initiating and completing the send operation. A blocking send is conceptually the same as a nonblocking send in which no useful work is done between the two phases, i.e., a call to an MPI1 send routine in nonblocking mode immediately followed by a call to MPI_WAIT.

When a message is sent in synchronized mode, execution is suspended on the sending process until an acknowledgment has been received from the destination process indicating that message receipt has completed. For a message sent in synchronized mode the message is not buffered by the system, and upon delivery to the the destination process it is placed directly into the supplied application buffer.

3.1.3. Other message passing utilities

On systems that provide buffering for messages (see Section 3.1.4) it is sometimes necessary for a process to check whether it has any pending messages satisfying given selection criteria. MPI1 provides the routine MPI_PROBE for this purpose. A pending message is one that was sent in blocking or nonblocking mode, but for which a corresponding receive has not yet been posted on the destination process. Such messages may be buffered by the system on the destination process, thus MPI_PROBE queries the contents of the system message buffers. Note that MPI_PROBE differs from MPI_STATS which checks for delivery of a message into an application buffer.

Either, or both, of the type and source message selection criteria specified in an MPI1 receive routine, or the routine MPI_PROBE can have wildcard values. A wildcard value for the type or source indicates that this criterion is to be ignored in selecting messages on a destination process, so it is possible to select messages regardless of type and/or source. After it has been ascertained by a process that it has received a wildcarded message, or that it has such a message pending, the actual length, type, and/or source of the message can be determined by calling MPI_INFOL, MPI_INFOT, and MPI_INFOS, respectively.

The routine MPI_CANCEL can be used to cancel a specified nonblocking send or receive operation initiated previously. After returning from MPI_CANCEL the nonblocking operation is no longer active, and the status of the nonblocking operation is left indeterminate.

3.1.4. Buffering of messages by the system

In describing MPI1's message passing routines, we have tried to avoid making any unnecessary assumptions about the underlying communication mechanism. In this section we will touch on some implementation issues that affect application portability, and whether message delivery is guaranteed.

In general, a communication system has some buffering capacity, as would usually be the case if the underlying communication mechanism was asynchronous. In such cases, when a message sent in blocking or nonblocking mode arrives at a destination process it is placed directly in an application buffer if a corresponding receive has already been posted; otherwise, it is placed in a system buffer. Messages in a system buffer are referred to as "pending messages," and remain in a system buffer until a corresponding receive is posted, at which point they are moved to an application buffer, and effectively deleted from the system buffer. Since the system can only provide a finite amount of buffer space for pending messages, an asynchronous communication mechanism must deal with the possibility that an incoming message would cause a system buffer to overflow. A simple recourse in such a situation is to discard the message, and flag an error condition on the receiving process. It should be noted that this would not be detected as an error by the sending process.

MPI can also be implemented on top of a synchronous communication system with no buffering capacity. In this case there are no system buffers, so the possibility of one overflowing does not arise. On such systems, a message buffer remains volatile on the sending process until a corresponding receive is posted on the destination process, at which point the message is delivered. Since messages are not buffered, the routine MPI_PROBE always indicates that there no pending messages.

To write applications that are portable between machines with different underlying communication mechanisms, and between machines whose communication systems have differing (and usually unknown) buffering capacities, reliance on system buffering should be avoided [5].

Although a synchronous communication system can guarantee message delivery (in the absence of hardware failures and software bugs), it is more difficult for an asynchronous system to do so. Thus, requiring guaranteed message delivery as part of a message passing standard may not be appropriate.

3.2. Process Groups

3.2.1. Creating and Managing Process Groups

Process groups provide a means of handling task parallelism, as well as controlling which processes cooperate in collective communication tasks, such as broadcast and reduction operations. MPI1 does not include collective communication routines, however, the support provided for process groups in MPI1 is intended to be fully consistent with the use of process groups in collective communications, a standard for which we expect to be defined subsequently. Thus, within the context of MPI1 process groups are provided solely as a means of supporting task parallelism, in which different process groups work on different tasks.

A process group is identified by a unique process group ID, or GID, which is an integer greater than zero. When a parallel program starts up, the processes allocated to an application belong to the predefined group with GID = ALL, where ALL is some integer assigned by the system. MPI1 provides two basic methods for creating a new group or groups. A new group can be created by each process in the group synchronously calling the routine MPI_DEFNG, which takes as its arguments the number of processes in the new group, and a list of the processes making up the group. A second routine, MPI_PARTG, is provided that allows a group to be partitioned into distinct subgroups based on the value of a specified key.

Information about group membership can be obtained using the routines MPI_GETID and MPI_INFOG. Given a process group with n members, the processes in the group are uniquely labeled $0, 1, \ldots, n-1$. These labels may be regarded as process ID numbers that are specific to a particular group, and will be referred to as Group Context Process ID numbers, or *GCPIDs*. A process has a different GCPID for each group of which it is a member. The routine MPI_GETID returns the GCPID of the calling process in a given group, or -1 if the process in not in the group. The routine MPI_INFOG can be used to determine which processes belong to a specified group of which the calling process is a member.

System memory is required to store information about the configuration of all currently defined groups. In order to make efficient use of this memory groups that are no longer needed by an application can be discarded, thereby freeing some memory for reuse. MPI provides the routine MPI_FREEG to discard a specified group. The routine MPI_FREEG must be called synchronously by all processes in the discarded group.

All processes that are involved in an operation to produce or discard a group or groups must perform the operation loosely synchronously, or full or partial deadlock may result.

Finally, the routine MPI_SYNCG imposes a barrier synchronization on a specified group of processes.

3.2.2. Task parallelism

All the routines discussed in Section 3.2.1 are concerned with creating and inquiring about process groups. The use of groups to manage task parallelism will now be discussed. We consider three types of task parallelism, corresponding to the SIMD, SPMD, and MIMD programming models, each of which subsumes the former. In SIMD task parallelism each group of processes executes the same instructions on different data. For example, suppose we have two groups of processes of the same size, and want to find the fast Fourier transform (FFT) of two vectors of the same length. Then, one FFT can be done by one group and the other FFT by the second group, and processes in each group with the same GCPIDs will execute the same instructions. In SPMD task parallelism each process executes the same code, but different groups may execute different instructions. The groups are not required to be of the same size, but must be distinct. Finally, in MIMD task parallelism different executable programs are loaded into each group. It should be noted that MIMD task parallelism can be mimicked by SPMD task parallelism by having each group execute different branches of a conditional statement within a single executable program. As currently defined MPI1 supports SPMD task parallelism, but not MIMD task parallelism.

Two routines specifically for using groups to manage the SPMD style of task parallelism will now be introduced. MPI_PUSHG establishes an environment in which a specified group of processes is treated as if it were the only processes in use by the application, i.e., it establishes a *process group context*. MPI_POPG re-establishes the process group context in effect prior to the corresponding preceding call to MPI_PUSHG. The use of these routines is, perhaps, best demonstrated with an example. Suppose we have a piece of software that performs some task in parallel on n processes, where n is an input parameter passed to the software. In executing the parallel software, communication between the processes is based on the assumption that they are numbered $0, 1, \ldots, n-1$. However, the actual PIDs of the processes in the group executing the software, in general, will not be labeled in this way since we are able to construct groups with arbitrary membership. However, the GCPIDs of the processes do run from 0 to $n-1$, so whenever the software refers to a source or destination process in the range 0 to $n-1$ this must

PID	GID	GCPID
0	1	0
1	1	1
2	1	2
3	1	3
4	2	0
5	2	1
6	2	2
7	2	3

Table 1: Mapping of group context PIDs to PIDs

be interpreted as a GCPID, which is then mapped to the corresponding PID. Thus, between a call to MPI_PUSHG and the corresponding subsequent call to MPI_POPG any reference to a process ID number is interpreted as a GCPID and is automatically mapped to the appropriate process ID number. For example, suppose the ALL group consists of 8 processes with process ID numbers $0, 1, \ldots, 7$. Now suppose further that these processes have been partitioned so that the first four form one group with GID=1, and the others form a second group with GID=2, and that the contexts for these groups have been established by calls to MPI_PUSHG. Then the GCPID associated with each process is as given in Table 1. Now, for example, if in the second group process 1 is required to send a message to process 3, the process ID numbers are interpreted so the communication actually takes place between processes 5 and 7. In this way a piece of software designed to execute on n processes with PIDs 0 through $n-1$ will perform correctly within any group context.

After a call to MPI_PUSHG the predefined group ALL refers to the group whose context has just been established, and not to the original set of processes. The group can then be partitioned, and subgroups can be used to form new root groups, by calling the routines MPI_PARTG and MPI_DEFNG. No reference may be made to any process or group outside the current group context. Group contexts may be nested.

A process must not be involved in any outstanding nonblocking communications within the current communication context (see Section 3.3) when calling MPI_PUSHG or MPI_POPG. All processes that are involved in an operation that changes the group context must perform the operation loosely synchronously, or full or partial deadlock may result.

3.2.3. Examples of the use of subgroups

To further clarify the use of subgroups in managing task parallelism we shall consider now some specific examples that use the MPI routines introduced in Sections 3.2.1 and 3.2.2. The first example is the solution of the shallow water equations on a sphere by the spectral transform method [25, 23]. An important computational kernel of this application is the the spectral transformation of a state variable defined on a rectangular longitude/latitude grid into a set of spectral coefficients. The spectral transform is evaluated in two phases. In the first phase a fast Fourier transform (FFT) is performed along each line of constant latitude in the grid. In the

```
ROWGRP = MPI_PARTG (ALL, MY_ROW)
COLGRP = MPI_PARTG (ALL, MY_COL)
```

Figure 1: Creation of row and column groups. Here `MPI_ROW` and `MY_COL` are the position of a process in the logical $P \times Q$ process mesh.

```
INFO = MPI_PUSHG (ROWGRP)
do 1D FFTs over longitude
INFO = MPI_POPG ( )
INFO = MPI_PUSHG (COLGRP)
do summation over latitude
INFO = MPI_POPG ( )
```

Figure 2: Pseudocode outline showing the use of process groups in the shallow water equation application.

second phase the spectral transform is completed by taking a weighted integral over latitude of the Fourier coefficients. Numerically this is performed by weighted summation.

Suppose that the longitude/latitude grid is distributed in blocks over a two-dimensional, logical mesh of $P \times Q$ processes. Currently MPI1 does not provide a mechanism for establishing process topologies of this type, however, a proposal for extending MPI1 to do this has been suggested by Hempel [12]. The processes in each row of the process mesh cooperate to evaluate the FFTs along a set of latitude lines. Then, the processes in each column cooperate to evaluate the spectral coefficients for a set of wavenumbers. The two phases of the spectral transform algorithm can be managed by partitioning the processes into row groups and column groups by making two calls to the routine MPI_PARTG, as shown in Figure 1.

The calls to MPI_PARTG are made once at the start of the application. Thereafter, the spectral transform of a state variable can be found by first establishing a process group context for the rows, and doing the FFTs over longitude for each latitude using a generic parallel FFT routine that assumes processes are numbered $0, 1, \ldots, Q-1$. Then, a process group context for the columns is established, and the summation over latitude for each wavenumber is performed using a parallel routine that assumes processes are numbered $0, 1, \ldots, P-1$. Thus, the pseudocode for the spectral transform algorithm is as as shown in Figure 2.

A second example of an application that might make use of process groups is an event-based circuit simulation code [24]. We are grateful to K. Yelick of the University of California, Berkeley, for suggesting this example. The circuit is decomposed into loosely coupled subcircuits with different computational loads. Each subcircuit is assigned to a process group, where the appropriate size of each group is determined by the computational load associated with the subcircuit. Thus, the groups are of different sizes. The subcircuits communicate whenever there is a significant change in voltage, however, once the input voltages for a time step are known the linear system associated with each subcircuit can be solved independently using LU factorization. Suppose, for example, the circuit may be split into three subcircuits with computational loads for the LU factorization in the ratio 1:9:16. Then, we might assign one process to the first subcircuit, 9 processes to the second, and 16 processes to the third, as shown schematically in Figure 3. In the parallel LU solver each group of processes needs to be arranged as a two-dimensional mesh, so as in the shallow water equation example, it is necessary to be able to specify a topology for a group of processes. Given an initial set of processes, these can be divided into process groups by calling MPI_PARTG or MPI_DEFNG. Each process then establishes a group context prior to performing the LU solve phase in order to determine with which processes it must cooperate to solve the linear system for the subcircuit to which it is assigned. Each group calls the same parallel LU solve routine, which in its simplest form has as its arguments the coefficient matrix, the righthand side vector, the size of the matrix, and the number of processes, P and Q, in each direction of the logical process mesh. Thus, the code would look something like the following,

```
LUGRP = MPI_PARTG (ALL, KEY)
      ⋮
INFO = MPI_PUSHG (LUGRP)
CALL LU_SOLVE (COEFFS, RHS, M, P, Q)
INFO = MPI_POPG ( )
```

Note that the parallel LU solver may itself use row and column oriented subgroups. These would be set up within the parallel LU solve routine.

3.3. Communication Contexts

It is sometimes necessary to ensure that different streams of communication do not interfere with one another. For example, in an application with two distinct phases, each involving nonblocking communication, there is the possibility that one phase may intercept messages intended for the other phase. This situation can arise if the message selectivity criteria of the two phases overlap, as may be the case when using a "canned" concurrent software library in which the selectivity criteria, in general, are unknown. Communication contexts, first used in the *Zipcode* message passing system [19, 20], provide a means of disambiguating such situations. In effect, a communication context provides a third selectivity criterion, in addition to type and source process, that may be used to control the receipt of messages. A communication context is uniquely labeled by a strictly positive integer called the Communication Context ID, or *CCID*. In MPI1 a communication context may be created by a call to MPI_NEWC, and a list of the current valid contexts may be obtained by calling MPI_INFOC. After invoking a previously created communication context by calling MPI_PUSHC, all messages subsequently sent are tagged with that context, and only those messages so tagged may be received. The current

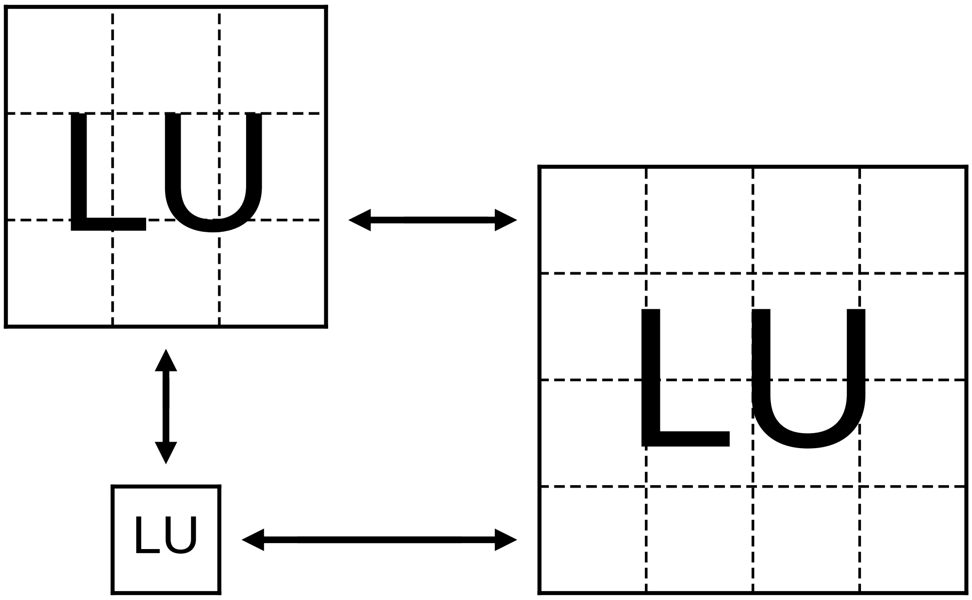

Figure 3: The division of processes into three groups of 1×1, 3×3, and 4×4 processes. Each group is assigned to a subcircuit, and independently performs a parallel LU solve. The arrows indicate the need for intermittent communication between the groups.

communication context is terminated by a call to MPI_POPC, which restores the communication context in effect prior to the preceding call to MPI_PUSHC. Communication contexts may be nested.

As an example, suppose we want to evaluate $D = AB + C^T$, where A, B, C, and D are all matrices. Then we might proceed as follows:

1. Initiate a nonblocking transpose of C
2. Call a concurrent library routine to find AB
3. Block until transpose of C is complete
4. Add C^T to AB to form D

Here the task of transposing matrix C, which requires interprocess communication, is overlapped with the distinct task of evaluating the matrix product AB, which also requires communication. If the message selectivity criteria within the two tasks are not unique there is the possibility that one task will receive messages intended for the other task. Note that this example assumes a sophisticated communication processor that not only knows what messages need to be sent for the transpose, but also interleaves them with those of the matrix communication. Potential message conflicts can be avoided by establishing different contexts for the matrix multiplication and matrix transpose tasks. The MPI1 code fragment for this example would be as shown in Figure 4.

In the above example, the communication context ICC is first created by calling MPI_NEWC. The transposition of matrix C is then initiated, with the communication context for this op-

```
ICC = MPI_NEWC ( )
BEGIN_TRANSPOSE (C)
IOK = MPI_PUSHC (ICC)
CALL MATMUL (D, A, B)
IOK = MPI_POPC ( )
END_TRANSPOSE (C)
D = D + C
```

Figure 4: Code fragment illustrating the use of communication contexts

eration being the default context. Next, the routine MPI_PUSHC is called to establish the communication context with CCID number ICC. When MATMUL is then called only messages labeled with this communication context will be visible to the application, thereby, insulating the messages associated with the matrix multiplication from those of the matrix transposition. When MATMUL returns, the routine MPI_POPC is called to restore the default communication context. The routine END_TRANSPOSE blocks, if necessary, until the transposition is completed. If the communication associated with the transpose has already completed following the return from MPI_POPC, then END_TRANSPOSE just copies C from a system to an application buffer.

Upon entering a program, or establishing a process group context by a call to MPI_PUSHG (see Section 3.2.2), a unique default communication context is established. A default communication context cannot be discarded, so a call to MPI_POPC when the current communication context is the default has no effect. When exiting a process group context by a call to MPI_POPG the communication context in effect prior to the preceding call to MPI_PUSHG is restored. Communication and process group contexts may be nested, but not misaligned.

3.4. Noncontiguous Messages

As discussed in Section 3.1, point-to-point scatter/gather types of communication, in which data are gathered from a message buffer on the sending process, and subsequently scattered into a buffer on the receiving process, may be performed using different variants of the send and receive routines. Sometimes it may be necessary to gather/scatter data between multiple buffers that may be of differing data types. In the Fortran language this cannot be done by a single call to the MPI1 send/receive routines.

In this section we introduce routines that (1) gather data from a buffer and pack it contiguously into another buffer, and (2) scatter data into a buffer from a contiguous buffer. In all cases the buffers are on the same process, and no interprocess communication is required. These routines allow complex messages to be packed into a contiguous buffer on the sending process. This message can then be sent to the destination process using the routines MPI_CSEND and MPI_CRECV, where it can then be unpacked.

Two sets of pack/unpack routines are provided, and their syntax is very similar to that of the corresponding noncontiguous send/receive routines. The first pair of routines, MPI_SPACK and MPI_SUNPACK, handles the case in which data blocks of constant size are respectively gathered from, or scattered to, a buffer with constant stride. The second pair of routines, MPI_GPACK and MPI_GUNPACK, handles the case in which the data blocks may be of differing sizes and lie at arbitrary locations in the buffer gathered from, or scattered to.

3.5. Utilities

This section outlines a set of utility routines. The routine MPI_MACHINE returns a string giving the machine name, type, Internet address, and other pertinent information about the machine that the calling process is running on. More detailed information can be obtained with the routine MPI_INFOMN which returns an integer array whose entries contains things like the number of processors in use, the maximum number of processors available, the number of the processor that the calling process is running on, the amount of memory per process and per processor, and other similar information. A complete list of the items included in the array returned by MPI_INFOMN has not yet been agreed on.

The routines MPI_DATE and MPI_TIME return the the date and time, respectively, as strings. User CPU time and elapsed wallclock time are returned as double precision seconds by the routines MPI_CPU and MPI_WALL, respectively.

Most of the routines in MPI1 return a value of -1 to indicate that an error has occurred. The nature and/or cause of the error can be determined by calling the routine MPI_ERROR. This returns an integer that indicates the error type applying to the most recent call to an MPI1 routine. Among the types of error that would be indicated by a call to MPI_ERROR are the use of an invalid PID, GCPID, CCID, or MSGID; the loss of a message on a process due to a system buffer overflow; the use of an invalid block length or stride in one of the message packing routines; and so on. If the integer returned by MPI_ERROR is passed to the routine MPI_ETEXT, a string is returned giving a short description of the error which can then be output by the application. This way of handling errors is essentially the same as that used by PARMACS [13].

4. Discussion and Rationale

In this section we discuss the reasoning behind some of the decisions made in designing MPI1. In the design of this interface, one of the main concerns was to keep both the calling sequences simple and the range of options limited, while at the same time maintaining sufficient functionality. This clearly implies a compromise, and a good decision is vital if MPI1 is to be accepted as a useful standard.

In order to avoid potential programming errors, values of scalar variables are not returned through argument lists. In MPI1, routines are written as function calls rather than subroutine calls, which provides a mechanism for returning scalars. One consequence of this is that in order to determine the source, length, and/or type when a wildcarded message has been received in nonblocking mode, or is known to be pending following a call to MPI_PROBE, it is necessary to call the information routines MPI_INF, MPI_INFOL, and/or MPI_INFOT.

It is not assumed in MPI1 that messages sent from one process to another are received in the order in which they were sent since some systems may use a non-deterministic routing scheme to avoid contention for communication links. Of course, even in such cases the correct order of messages could be recovered by the receiving process if each message was labeled by the sending process with a sequence number. Whether or not messages from one process to another arrive in the correct order has no impact on the definition of a standard, though clearly the assumption is vital to the correct functioning of many parallel algorithms.

MPI1 defines three modes for send and receiving messages, namely the blocking, nonblocking, and synchronized modes. We believe that these are the most widely used types of point-to-point communication operations, and in order to avoid too many varieties of send routine, some potentially useful functionality has been excluded from MPI1. For example, MPI1 does not include forced communication of the sort provided in Intel's NX/2 through the use of "force types." In forced communication, if a message sent in nonblocking mode is delivered to a process, and an application buffer has not already been made available for it by previously posting a receive, then the message is simply discarded, rather than being placed in a system buffer on the destination process. This functionality could be provided in MPI1 by reserving a certain range of types for forced communication, just as in NX/2. The justification for using forced communication is that it avoids some overhead, and thus is often faster. The main disadvantage is that it is the responsibility of the application to ensure that a receive is always posted prior to delivery of a forced message, otherwise the message will be lost.

In handling communication contexts MPI1 uses an approach that is independent of the message type selectivity mechanism. A different approach would be for each phase of an application to initially register the range of types it will use, and for a central message type registry to check for overlaps between the ranges claimed by different phases. An overlap would indicate to the application the potential for communication conflicts. The best approach is unclear. The first option would be more natural to the user, while for the second option communication context control functions would be easier to port onto most current parallel systems without major changes to the runtime systems. Thus, the question is how much MPI1 should be influenced by the presently available systems.

5. Outstanding Issues

In this section we outline a few of the issues that need to be addressed by MPI1, and some features that should be considered for inclusion in future versions of MPI1.

A number of extensions to the support provided by MPI1 for process groups are possible. For example, currently in MPI1 the union of groups cannot be formed, nor is it possible for single processes to join or leave a group. Abstractions for permitting process groups to communicate with each other are another possible extension. MPI1 does not allow processes to be created or destroyed, or for different executable codes to be loaded into different processes, as would be required in order to support the MIMD style of task parallelism. The need for all these possible enhancements must be given careful consideration.

In MPI1 no explicit mechanism is provided to allow a process to inquire about the existence and membership of groups of which it is not a member. In a more general system it would be possible for a process to access information about any group. One way of doing this would be to have some processes dedicated to storing data about the current valid groups, and responding to requests for this information. Whenever, a group is created, discarded, or modified the processes involved must synchronize with one or more of the "group database" processes and inform them of the changes. Clearly, if there are too few such processes bottlenecks may develop in accessing their data; if there are too many then memory and compute power are wasted.

In the current version of MPI1 a process group is formed by a collection of processes without any additional structure. As one consequence, new root groups can only be defined by listing the identifiers of the participating processes. Typical applications, on the other hand, have much more internal structure. For example, the solution of a partial differential equation on a 3D grid is usually performed by processes which are arranged in a corresponding structure. If the programming interface does not provide functions for defining that structure, the user must program the relationship of the logical position of a process and its identifier himself. Also, this information is not available for automatic tools which map neighboring processes onto neighboring hardware processors. Therefore, a mechanism such as that suggested by Hempel [12] for defining, and inquiring about, logical process topologies would be a useful addition to the message passing standard.

Another important consideration when extending MPI1 to handle heterogeneous distributed computing is the fact that different machines not only have different data formats, but also prefer different packet sizes. It would therefore appear that a table is needed that not only maps a PID number to an Internet address and process ID on the destination machine, but which also includes the target machine's preferred packet size.

6. Conclusions

We do not claim to provide the definitive answer to everyone's communication needs. Indeed, our insistence on simplicity precludes that. However, we believe the MPI1 routines proposed here will be useful as a basis for further discussion in the development of a standard for message passing in distributed memory environments. An MPI standards committee was formally instituted in November 1992, with the objective of providing a forum for discussion and of defining a standard message passing interface by July 1993. This committee is similar in structure and organization to that which developed the High Performance Fortran standard. Members of the distributed memory computing community who wish to become involved in the standardization process should send email to walker@msr.epm.ornl.gov by May 1, 1993.

Acknowledgments

This work was partially supported by the ESPRIT programme and the PPPE project. We gratefully acknowledge the participants of the First CRPC Workshop on Standards for Message Passing in a Distributed Memory Environment, and are grateful to the Center for Research on Parallel Computing for sponsoring this workshop. It is also a pleasure to acknowledge the helpful comments and suggestions of Vas Bala, Mark Debbage, Al Geist, William Gropp, Cherri Pancake, Paul Pierce, Peter Rigsbee, Anthony Skjellum, Marc Snir, and Joel Williamson.

7. References

[1] H. E. Bal. *Programming Distributed Systems.* Prentice Hall International, Hemel Hempstead, England, 1991.

[2] 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.

[3] 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.

[4] N. Carriero and D. Gelernter. How to write parallel programs. *ACM Computing Surveys*, 21(3):323, September 1989.

[5] M. Debbage and M. Hill. Draft messaging ideas, revision 0.3. Technical report, Southampton University, UK, 1992. Preprint.

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

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

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

[9] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, October 1990.

[10] D. Gelernter. Generative communication in Linda. *ACM Trans. Prog. Lang. Sys.*, 7(1):80–112, 1985.

[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. A proposal for virtual topologies in MPI1. Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, November 1992.

[13] R. Hempel, H.-C. Hoppe, and A. Supalov. A proposal for a PARMACS library interface. Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, October 1992.

[14] D. Mallon, J. Nash, and P. Dew. Shared objects and its role in standardization. Technical report, Leeds University, UK, 1992. Preprint.

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

[16] S. Otto. MetaMP: a higher level abstraction for message passing programming. Technical report, Oregon Graduate Institute, Department of Computer Science, January 1991.

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

[18] 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.

[19] 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.

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

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

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

[23] D. W. Walker, P. H. Worley, and J. B. Drake. Parallelizing the spectral transform method – part II. *Concurrency: Practice and Experience*, 4:509–531, 1992.

[24] Chih-Po Wen. Timing simulation on a distributed memory multiprocessor. Master's thesis, University of California, Berkeley, CA, 1992.

[25] P. H. Worley and J. B. Drake. Parallelizing the spectral transform method – part I. *Concurrency: Practice and Experience*, 4:269–291, 1992.

Appendix A

In this appendix we give Fortran specifications for the MPI1 routines. The C language specifications are not given explicitly, but are very similar, except for the routines dealing with arbitrary scatter/gather operations (MPI_GSEND, MPI_GRECV, MPI_GPACK and MPI_GUNPACK). In the synopses of the Fortran specifications of some of the routines, message buffers are referred to as integer arrays; however, real arrays can also be passed to these routines.

The appendix is consists of the following five sections.

1. Point-to-point message passing routines,

2. Support for process groups,

3. Support for buffer copying,

4. Support for communication contexts,

5. Utilities.

A.1 Point-to-Point Message Passing Routines

In this section we provide specifications for the following point-to-point message passing and related routines.

- **MPI_CANCEL** Cancel nonblocking send or receive
- **MPI_CRECV** Receive contiguous message
- **MPI_CSEND** Send contiguous message
- **MPI_GRECV** Receive into buffer with arbitrary scatter
- **MPI_GSEND** Send from buffer with arbitrary gather
- **MPI_INFOL** Get length of pending or received message
- **MPI_INFOS** Get source of pending or received message
- **MPI_INFOT** Get type of pending or received message
- **MPI_PROBE** Check pending messages
- **MPI_SRECV** Receive into buffer with constant stride
- **MPI_SSEND** Send from buffer with constant stride
- **MPI_STATS** Check status of nonblocking send or receive
- **MPI_WAIT** Block until send or receive has completed

Message selectivity (within a communication context) is by source process and message type, either of which may have the "wildcard" value of -1, indicating that any source and/or type is acceptable.

Nonblocking sends and receives return a message ID that is unique within the current group context. All other sends and receives return the number of bytes actually sent or received, or -1 if an error occurred.

NAME

MPI_CANCEL Cancel a previously initiated nonblocking send or receive

SYNOPSIS

integer function MPI_CANCEL (msgid)

integer msgid

INPUT ARGUMENTS

msgid	message identifier returned by a call to a nonblocking send or receive

DESCRIPTION

MPI_CANCEL cancels a previously issued nonblocking send or receive specified by the message identifier, `msgid`. Upon return the nonblocking send or receive is no longer active, and may or may not have completed.

RETURN VALUE

MPI_CANCEL returns 0, or -1 if an error occurs.

NAME

MPI_CRECV Receive a message contiguously into a buffer.

SYNOPSIS

integer function MPI_CRECV (mode, buf, source, type, maxlen)

character mode

integer buf(*)

integer source

integer type

integer maxlen

INPUT ARGUMENTS

mode	the mode of the receive ("blocking", "nonblocking", or "synchronized")
source	the ID number of the process sending the message
type	the message type, or type mask
maxlen	the maximum length of the message in bytes

OUTPUT ARGUMENTS

buf	the application buffer into which the message is received.

DESCRIPTION

If `mode` has the value "`blocking`" then the calling process blocks until a message of a specified type is received from a specified source into the application buffer `buf`. Deadlock will occur if no corresponding message is sent loosely synchronously by the source process.

If `mode` has the value "`nonblocking`" then the calling process posts a receive for a message of a specified type from a specified source, and immediately returns.

If `mode` has the value "`synchronized`" then the calling process blocks until the specified message has been received into the application buffer, `buf`, and then sends an acknowledgment to the source process before returning. The receive must be matched by a corresponding send, also done in synchronized mode.

For all modes, if `source` is -1 then selectivity by source is ignored. Similarly, if `type` is -1 then selectivity by type is ignored. Messages longer than `maxlen` bytes are truncated to `maxlen` bytes.

For all modes, the message received is stored contiguously in the buffer `buf`.

RETURN VALUE

Upon successful completion, if `mode` is "`blocking`" or "`synchronized`" then **MPI_CRECV** returns the length of the message received in bytes. This will exceed `maxlen` bytes if the message was truncated. If `mode` is "`nonblocking`" then **MPI_CRECV** returns the message ID number associated with the receive operation. A value of -1 is returned if an error occurs.

NAME

MPI_CSEND Send a message contiguously from a buffer.

SYNOPSIS

integer function MPI_CSEND (mode, buf, dest, type, len)
character mode
integer buf(*)
integer dest
integer type
integer len

INPUT ARGUMENTS

mode	the mode of the send operation
buf	the buffer containing the message to be sent
dest	the ID number of the process to which the message is sent
type	the message type
len	the length of the message in bytes

DESCRIPTION

If `mode` has the value "`blocking`" then **MPI_CSEND** sends a message of type `type` to process `dest`, and blocks until the message buffer, `buf`, is available for reuse.

If `mode` has the value "`nonblocking`" then **MPI_CSEND** initiates transmission of a message of type `type` to process `dest`, and immediately returns. The message buffer, `buf`, should not be changed until the message is guaranteed to have been sent, i.e., to have "cleared the buffer", by a call to **MPI_WAIT**, or by a call to **MPI_STATS** returning a nonnegative integer.

If `mode` has the value "`synchronized`" then **MPI_CSEND** sends a message of type `type` to process `dest`, and blocks until an acknowledgment is received from the destination process to indicate that message receipt has completed. The send must be matched by a corresponding receive, also done in synchronized mode.

For all modes, the message consists of the `len` contiguous bytes in the buffer `buf`.

RETURN VALUE

If `mode` is "`blocking`" or "`synchronized`" then **MPI_CSEND** returns the number of bytes sent. If `mode` is "`nonblocking`" **MPI_CSEND** returns the message ID number associated with the send operation. A value of -1 is returned if an error occurs.

NAME

MPI_GRECV Receive a message and scatter it arbitrarily into a buffer.

SYNOPSIS

integer function MPI_GRECV (mode, buf, source, type, nlist, ilist, nblks)
character mode
integer buf(*)
integer source
integer type
integer nlist(*)
integer ilist(*)
integer nblks

INPUT ARGUMENTS

mode	the mode of the receive ("blocking", "nonblocking", or "synchronized")
source	the ID number of the process sending the message
type	the message type, or type mask
nlist	list of the number of bytes in each data block
ilist	list of the location in `buf` at which each data block starts
nblks	maximum number of data blocks to be scattered

OUTPUT ARGUMENTS

buf	the application buffer into which the message is scattered

DESCRIPTION

If `mode` has the value "`blocking`" then the calling process blocks until a message of a specified type is received from a specified source into the application buffer `buf`. Deadlock will occur if no corresponding message is sent loosely synchronously by the source process.
If `mode` has the value "`nonblocking`" then the calling process posts a receive for a message of a specified type from a specified source, and immediately returns.
If `mode` has the value "`synchronized`" then the calling process blocks until the specified message has been received into the application buffer, `buf`, and then sends an acknowledgment to the source process before returning. The receive must be matched by a corresponding send, also done in synchronized mode.
For all modes, if `source` is -1 then selectivity by source is ignored. Similarly, if `type` is -1 then selectivity by type is ignored. Messages longer than `maxlen` bytes are truncated to `maxlen` bytes.
For all modes, the way in which the data received are stored in the buffer `buf` is controlled by the arrays `nlist` and `ilist`. The data received are treated as a succession of data

blocks, with the ith block being of size `nlist(i)` bytes. This is stored in the buffer **`buf`** so that the start of the block is at **`ilist(i)`** bytes from the start of **`buf`**. The maximum number of data blocks received is **`nblks`**. It is assumed that all indices and numbering of data items begin at 0. It is the responsibility of the user to ensure that **`buf`** is large enough to hold the data scattered into it.

RETURN VALUE

Upon successful completion, if `mode` is "`blocking`" or "`synchronized`" then **MPI_GRECV** returns the total number of bytes received. If `mode` is "`nonblocking`" then **MPI_GRECV** returns the message ID number associated with the receive operation. A value of -1 is returned if an error occurs.

NAME

MPI_GSEND Send a message gathered arbitrarily from a buffer.

SYNOPSIS

integer function MPI_GSEND (mode, buf, dest, type, nlist, ilist, nblks)
character mode
integer buf(∗)
integer dest
integer type
integer nlist(∗)
integer ilist(∗)
integer nblks

INPUT ARGUMENTS

mode	the mode of the send ("blocking", "nonblocking", or "synchronized")
buf	the buffer containing the message to be sent
dest	the ID number of the process to which the message is sent
type	the message type
nlist	list of the number of bytes in each data block
ilist	list of the location in `buf` at which each data block starts
nblks	number of data blocks to be gathered

DESCRIPTION

If `mode` has the value "`blocking`" then **MPI_GSEND** sends a message of type `type` to process `dest`, and blocks until the message buffer, `buf`, is available for reuse.

If `mode` has the value "`nonblocking`" then **MPI_GSEND** initiates transmission of a message of type `type` to process `dest`, and immediately returns. The message buffer, `buf`, should not be changed until the message is guaranteed to have been sent, i.e., to have "cleared the buffer", by a call to **MPI_WAIT**, or by a call to **MPI_STATS** returning a nonnegative integer.

If `mode` has the value "`synchronized`" then **MPI_GSEND** sends a message of type `type` to process `dest`, and blocks until an acknowledgment is received from the destination process to indicate that message receipt has completed.

For all modes, the way in which the message sent is gathered from the buffer `buf` is controlled by the arrays `nlist` and `ilist`. The data are gathered in blocks, with the ith block being of size `nlist(i)` bytes. This is gathered from the buffer `buf` starting at the location `ilist(i)` bytes from the start of `buf`. The total number of data blocks gathered is `nblks`. It is assumed that all indices and numbering of data items begin at 0.

RETURN VALUE

If `mode` is "`blocking`" or "`synchronized`" then **MPI_GSEND** returns the number of bytes sent. If `mode` is "`nonblocking`" then **MPI_GSEND** returns the message ID number associated with the send operation. A value of -1 is returned if an error occurs.

NAME

MPI_INFOL Determine the length of a pending or received message.

SYNOPSIS

integer function MPI_INFOL ()

ARGUMENTS

None

DESCRIPTION

MPI_INFOL determines the length in bytes of a pending or received message. It only returns a valid result if used directly after a call to a receive routine in blocking or synchronized mode, or directly after a call to **MPI_STATS** or **MPI_PROBE** that has returned a nonnegative integer.

RETURN VALUE

Directly after a call to a receive routine in blocking or synchronized mode, a call to **MPI_WAIT**, or a call to to **MPI_STATS** that returns a nonnegative integer, the routine **MPI_INFOL** returns the length in bytes of the message just received. If called directly after **MPI_PROBE** has returned a nonnegative number, **MPI_INFOL** returns the length in bytes of the pending message. If there are no pending messages -1 is returned.

NAME

MPI_INFOS Determine the source process of a pending or received message.

SYNOPSIS

integer function MPI_INFOS ()

ARGUMENTS

None

DESCRIPTION

MPI_INFOS determines the source process of a pending or received message. It only returns a valid result if used directly after a call to a receive routine in blocking or synchronized mode, or directly after a call to **MPI_STATS** or **MPI_PROBE** that has returned a nonnegative integer.

RETURN VALUE

Directly after a call to a receive routine in blocking or synchronized mode, a call to **MPI_WAIT**, or a call to to **MPI_STATS** that returns a nonnegative integer, the routine **MPI_INFOS** returns the ID number of the process that sent the message just received. If called directly after **MPI_PROBE** has returned a nonnegative number, **MPI_INFOS** returns the ID number of the process that sent the pending message. If there are no pending messages -1 is returned.

NAME

MPI_INFOT Determine the type of a pending or received message.

SYNOPSIS

integer function MPI_INFOT ()

ARGUMENTS

None

DESCRIPTION

MPI_INFOT determines the type of a pending or received message. **MPI_INFOT** only returns a valid result if used directly after a call to a receive routine in blocking or synchronized mode, or **MPI_WAIT**, or directly after a call to **MPI_PROBE** or **MPI_STATS** that has returned a nonnegative integer.

RETURN VALUE

Directly after a call to a receive routine in blocking or synchronized mode, **MPI_WAIT**, or a call to **MPI_STATS** that returns a nonnegative integer, **MPI_INFOT** returns the type of the message just received. If called directly after **MPI_PROBE** has returned a nonnegative number, **MPI_INFOT** returns the type of the pending message. If there are no pending messages -1 is returned.

NAME

MPI_PROBE Check for pending messages.

SYNOPSIS

integer function MPI_PROBE (source, type)

integer source

integer type

INPUT ARGUMENTS

source the ID number of the process sending the message.

type the message type, or type mask.

DESCRIPTION

MPI_PROBE checks if there is a message from a specified source and of a specified type awaiting receipt. That is, if there is a such a message stored in a system buffer for which a receive has not yet been posted. If `source` is -1 then this argument is ignored. Similarly, if `type` is -1 then this argument is ignored. Only messages sent using the routines sent in blocking or nonblocking mode may be buffered by the system on the receiving process, so it only makes sense to use **MPI_PROBE** to probe such messages.

RETURN VALUE

If a message satisfying the selectivity criteria is awaiting receipt **MPI_PROBE** returns the length of the message in bytes. Otherwise, -1 is returned.

NAME

MPI_SRECV Receive a message and scatter it with constant stride into a buffer.

SYNOPSIS

integer function MPI_SRECV (mode, buf, source, type, lenblk, stride, nblks)
character mode
integer buf(*)
integer source
integer type
integer lenblk
integer stride
integer nblks

INPUT ARGUMENTS

mode	the mode of the receive ("blocking", "nonblocking", or "synchronized")
source	the ID number of the process sending the message
type	the message type, or type mask
lenblk	the size in bytes of each data block
stride	the number of bytes between the start of each data block
nblks	maximum number of data blocks to be scattered

OUTPUT ARGUMENTS

buf	the application buffer into which the message is scattered

DESCRIPTION

If `mode` has the value "`blocking`" then the calling process blocks until a message of a specified type is received from a specified source into the application buffer `buf`. Deadlock will occur if no corresponding message is sent loosely synchronously by the source process.
If `mode` has the value "`nonblocking`" then the calling process posts a receive for a message of a specified type from a specified source, and immediately returns.
If `mode` has the value "`synchronized`" then the calling process blocks until the specified message has been received into the application buffer, `buf`, and then sends an acknowledgment to the source process before returning. The receive must be matched by a corresponding send, also done in synchronized mode.
For all modes, if `source` is -1 then selectivity by source is ignored. Similarly, if `type` is -1 then selectivity by type is ignored. Messages longer than `maxlen` bytes are truncated to `maxlen` bytes.
For all modes, the data received are treated as a succession of data blocks, each of length `lenblk` bytes. Data blocks are placed in the buffer `buf` so that the start of successive

blocks are separated by `stride` bytes. The maximum number of data blocks received is `nblks`. It is the responsibility of the user to ensure that `buf` is large enough to hold the data scattered into it.

RETURN VALUE

Upon successful completion, if `mode` is "`blocking`" or "`synchronized`" then **MPI_SRECV** returns the length of the message received in bytes. If `mode` is "`nonblocking`" then **MPI_SRECV** returns the message ID number associated with the receive operation. A value of -1 is returned if an error occurs.

NAME

MPI_SSEND Send a message gathered with constant stride from a buffer.

SYNOPSIS

integer function MPI_SSEND (mode, buf, dest, type, lenblk, stride, nblks)

character mode

integer buf(∗)

integer dest

integer type

integer lenblk

integer stride

integer nblks

INPUT ARGUMENTS

mode	the mode of the send ("blocking", "nonblocking", or "synchronized")
buf	the buffer containing the message to be sent
dest	the ID number of the process to which the message is sent
type	the message type
lenblk	the size in bytes of each data block
stride	the number of bytes between the start of each data block
nblks	number of data blocks to be gathered

DESCRIPTION

If `mode` has the value "`blocking`" then **MPI_SSEND** sends a message of type `type` to process `dest`, and blocks until the message buffer, `buf`, is available for reuse.

If `mode` has the value "`nonblocking`" then **MPI_SSEND** initiates transmission of a message of type `type` to process `dest`, and immediately returns. The message buffer, `buf`, should not be changed until the message is guaranteed to have been sent, i.e., to have "cleared the buffer", by a call to **MPI_WAIT**, or by a call to **MPI_STATS** returning a nonnegative integer.

If `mode` has the value "`synchronized`" then **MPI_SSEND** sends a message of type `type` to process `dest`, and blocks until an acknowledgment is received from the destination process to indicate that message receipt has completed.

For all modes, the data sent are gathered from the buffer `buf` in blocks, each of length `lenblk` bytes. The start of successive data blocks are separated by `stride` bytes in the buffer `buf`. The total number of data blocks gathered is `nblks`.

RETURN VALUE

If `mode` is "`blocking`" or "`synchronized`" then **MPI_SSEND** returns the number of bytes sent. If `mode` is "`nonblocking`" **MPI_SSEND** returns the message ID number associated with the send operation. A value of -1 is returned if an error occurs.

NAME

MPI_STATS Check the status of a nonblocking send or receive operation.

SYNOPSIS

integer function MPI_STATS (msgid)

integer msgid

INPUT ARGUMENTS

msgid message identifier returned by a call to a nonblocking send or receive

DESCRIPTION

If the message identifier, `msgid`, refers to a message being sent in nonblocking mode, then **MPI_STATS** checks if the message has cleared the message buffer yet. If it has, then the message buffer is available for reuse. If the message identifier, `msgid`, refers to a message being received in nonblocking mode, then **MPI_STATS** checks if message receipt has been completed yet, i.e., if the incoming message has been placed in an application buffer. If it has, then the data received into the buffer is available for use.

RETURN VALUE

MPI_STATS returns the number of bytes sent or received if the nonblocking send or receive operation has completed. Otherwise, -1 is returned.

NAME

MPI_WAIT Block until a nonblocking send or receive operation has completed.

SYNOPSIS

integer function MPI_WAIT (msgid)

integer msgid

INPUT ARGUMENTS

msgid message identifier returned by a call to a nonblocking send or receive

DESCRIPTION

If the message identifier, `msgid`, refers to a message being sent in nonblocking mode, then **MPI_WAIT** blocks until the message has cleared the message buffer. Upon return from such a call to **MPI_WAIT** the message buffer is available for reuse, but receipt of the message by the destination process is not guaranteed. If the message identifier, `msgid`, refers to a message being received in nonblocking mode, then **MPI_WAIT** blocks until message receipt has been completed. The data received into the message buffer is then available for use.

RETURN VALUE

On successful completion **MPI_WAIT** returns the number of bytes sent or received. Otherwise, -1 is returned.

A.2 Support for Process Groups

In this section specifications for the following routines for supporting process groups are given.

- **MPI_DEFNG** Create a group from a list of processes
- **MPI_FREEG** Discard a group
- **MPI_GETID** Determine GCPID of calling process in a group
- **MPI_INFOG** Determine processes in a group
- **MPI_PARTG** Partition a group
- **MPI_POPG** Restore previous group context
- **MPI_PUSHG** Establish new group context
- **MPI_SYNCG** Synchronize a group of processes

NAME

MPI_DEFNG Define a group of processes.

SYNOPSIS

integer function MPI_DEFNG (nprocs, plist)

integer nprocs

integer plist($*$)

INPUT ARGUMENTS

nprocs	the number of processes in the new group
plist	a list of `nprocs` process ID numbers

DESCRIPTION

MPI_DEFNG creates a new group consisting of the `nprocs` processes whose ID numbers are listed in the array `plist`. The new group can subsequently be partitioned by calls to **MPI_PARTG**. **MPI_DEFNG** must be called synchronously by all the processes listed in `plist`.

RETURN VALUE

On successful completion **MPI_DEFNG** returns the unique group ID number of the newly formed group. If an error occurs a value of -1 is returned.

NAME

MPI_FREEG Discard a specified group.

SYNOPSIS

integer function MPI_FREEG (gid)

integer gid

INPUT ARGUMENTS

gid the group ID number of the group to be discarded

DESCRIPTION

MPI_FREEG may be used to free memory that stores information about groups that are no longer needed. The group `gid` is discarded, and may not be referred to subsequently. **MPI_FREEG** must be called synchronously by all processes in the group `gid`.

RETURN VALUE

On successful completion **MPI_FREEG** returns 0. Otherwise -1 is returned.

NAME

MPI_GETID Determine the group context PID of the calling process for a specified group ID number.

SYNOPSIS

integer function MPI_GETID (gid)
integer gid

INPUT ARGUMENTS

gid the group ID for which the group context PID is required

DESCRIPTION

MPI_GETID determines the group context PID of the calling process within the group `gid`. A value of -1 is returned if the calling process is not in the group `gid`.

RETURN VALUE

MPI_GETID returns the group context PID of the calling process within the group `gid`. A value of -1 is returned if the calling process is not in the group `gid`.

NAME

MPI_INFOG Determine the number of processes in a group, and return a list of the PID numbers of the group members.

SYNOPSIS

integer function MPI_INFOG (gid, maxlis, plist)
integer gid
integer maxlis
integer plist(*)

INPUT ARGUMENTS

gid	a group ID number
maxlis	the maximum size of the array `plist`

OUTPUT ARGUMENTS

plist	a list of the PID numbers of the processes in group `gid`

DESCRIPTION

MPI_INFOG determines the number of processes the group `gid`, and returns a list of the PID numbers of the group members in the array `plist`. The calling process must be a member of the group `gid`. If there are more than `maxlis` processes in group `gid`, only the PID numbers of `maxlis` of them are returned in `plist`.

RETURN VALUE

On successful completion **MPI_INFOG** returns the number of processes in the group `gid`, or -1 if an error occurs.

NAME

MPI_PARTG Partition a group into subgroups.

SYNOPSIS

integer function MPI_PARTG (gid, key)

integer gid

integer key

INPUT ARGUMENTS

gid	the ID number of the group to be partitioned
key	the key whose value determines the partitioning

DESCRIPTION

MPI_PARTG partitions the group `gid` into subgroups according to the value of `key`. All processes for which `key` has the same value form a distinct subgroup. **MPI_PARTG** must be called synchronously by all processes in the group `gid`.

RETURN VALUE

On successful completion **MPI_PARTG** returns the unique GID number of the subgroup to which the calling process belongs. Otherwise, -1 is returned.

NAME

MPI_POPG Re-establish former process group context.

SYNOPSIS

integer function MPI_POPG ( )

ARGUMENTS

None

DESCRIPTION

MPI_POPG re-establishes the process group context that was in effect before the preceding call to **MPI_PUSHG**. **MPI_POPG** must be called synchronously by all processes in the group whose context was established by the preceding call to **MPI_PUSHG**. The calling process must not be involved in any nonblocking communication within the current communication context when calling **MPI_POPG**.

RETURN VALUE

On successful completion **MPI_POPG** returns the process group ID number of the group whose context is re-established. Otherwise, -1 is returned.

NAME

MPI_PUSHG Establish a new group context.

SYNOPSIS

integer function MPI_PUSHG (gid)

integer gid

INPUT ARGUMENTS

gid the group ID number of the context to be established

DESCRIPTION

A call to **MPI_PUSHG** establishes an environment in which it appears to the processes in the group `gid` that they are the only processes in use by the application. This environment is called the process group context of `gid`. The effect of a call to **MPI_PUSHG** is nullified by the next subsequent call to **MPI_POPG**, which re-establishes the process group context that was in effect before the call to **MPI_PUSHG**. If the group `gid` contains n processes, then within the group context of `gid` each process is labeled by a unique integer between 0 and $n-1$, referred to as its group context PID. Processes may only be referenced by their group context PIDs, which are automatically mapped to the corresponding process ID numbers by the system. It is an error to refer to any process ID number outside the range 0 to $n-1$, and the processes in group `gid` may not communicate with processes outside the group. Groups created outside the current group context by calls to **MPI_DEFNG**, or **MPI_PARTG** may not be referenced. Groups created within the current group context may not be referenced after exiting the context by calling **MPI_POPG**. Within a group context the group ALL refers to just the processes in the current group context. Group contexts may be nested. **MPI_PUSHG** must be called synchronously by all processes in the group `gid`. The calling process must not be involved in any nonblocking communication within the current communication context when calling **MPI_PUSHG**.

RETURN VALUE

On successful completion **MPI_PUSHG** returns the number of processes in the group `gid`. Otherwise -1 is returned.

NAME

MPI_SYNCG Synchronize processes.

SYNOPSIS

integer function MPI_SYNCG (gid)

integer gid

ARGUMENTS

gid a process group ID

DESCRIPTION

MPI_SYNCG performs a barrier synchronization involving all processes in the group `gid`, of which the calling process must be a member.

RETURN VALUE

On successful completion **MPI_SYNCG** returns 0. Otherwise, -1 is returned.

A.3 Support for Buffer Copying

In this section specifications for the following routines for packing data into and out of message buffers are given.

- **MPI_SPACK** Gather data with constant stride
- **MPI_SUNPACK** Scatter data with constant stride
- **MPI_GPACK** General-purpose gather routine
- **MPI_GUNPACK** General-purpose scatter routine

NAME

MPI_SPACK Pack data blocks into a buffer with constant stride.

SYNOPSIS

integer function MPI_SPACK (buf, lenblk, stride, nblk, msg)

integer buf(*)

integer lenblk

integer stride

integer nblk

integer msg(*)

INPUT ARGUMENTS

buf	buffer from which data are to be gathered
lenblk	size of each data block in bytes
stride	number of bytes between successive blocks in buffer `buf`
nblk	number of data blocks to be gathered

OUTPUT ARGUMENTS

msg	buffer in which the gathered data is packed

DESCRIPTION

MPI_SPACK gathers data from the buffer `buf` and packs it contiguously into the buffer `msg`. In `buf` the data blocks consist of `lenblk` bytes, with the starts of successive blocks being separated by a constant `stride` bytes. The number of blocks gathered in `nblk`. The most common use of **MPI_SPACK** is to fill a message buffer for subsequent communication.

RETURN VALUE

Upon successful completion **MPI_SPACK** returns the total length of the message in bytes. Otherwise, -1 is returned.

NAME

MPI_SUNPACK Unpack data blocks from a buffer with constant stride.

SYNOPSIS

integer function MPI_SUNPACK (buf, lenblk, stride, nblk, msg)

integer buf(∗)

integer lenblk

integer stride

integer nblk

integer msg(∗)

INPUT ARGUMENTS

buf	buffer to which data are to be scattered
lenblk	size of each data block in bytes
stride	number of bytes between successive blocks in buffer `buf`
nblk	number of data blocks to be scattered

OUTPUT ARGUMENTS

msg	buffer in which the data to be scattered are packed

DESCRIPTION

MPI_SUNPACK unpacks contiguous data from the buffer `msg` and scatters it with constant stride into the buffer `buf`. Successive contiguous blocks of `lenblk` bytes are extracted from `msg` and copied to `buf` so that the first such block is aligned with the start of `buf`, and the start of successive blocks is separated by `stride` bytes. A total of `nblk` data blocks are unpacked. The most common use of **MPI_SUNPACK** is to unpack data received from another process. It is the responsibility of the user to ensure that `buf` is large enough to hold the data unpacked into it.

RETURN VALUE

Upon successful completion **MPI_SUNPACK** returns the total length of the message in bytes. Otherwise, -1 is returned.

NAME

MPI_GPACK General routine for packing data blocks into a buffer.

SYNOPSIS

integer function MPI_GPACK (buf, nlist, ilist, nblk, msg)

integer buf(∗)

integer nlist(∗)

integer ilist(∗)

integer nblk

integer msg(∗)

INPUT ARGUMENTS

buf	buffer from which data are to be gathered
nlist	list of the number of bytes in each block
ilist	list of the location in `buf` at which each data block starts
nblk	number of data blocks to be gathered

OUTPUT ARGUMENTS

msg	buffer into which the gathered data are packed

DESCRIPTION

MPI_GPACK extracts `nblk` data blocks from the buffer `buf` and packs them contiguously into the buffer `msg` according to the information in the arrays `nlist` and `ilist`. The ith data block extracted consists of the contiguous `nlist(i)` bytes starting at the location `ilist(i)` bytes from the start of `buf`. It is assumed that all indices and numbering of data items begin at 0. The most common use of **MPI_GPACK** is to fill a message buffer for subsequent communication.

RETURN VALUE

Upon successful completion **MPI_GPACK** returns the total length of the message in bytes. Otherwise, -1 is returned.

NAME

MPI_GUNPACK General routine for unpacking data blocks from a buffer.

SYNOPSIS

integer function MPI_GUNPACK (buf, nlist, ilist, nblk, msg)
integer buf(*)
integer nlist(*)
integer ilist(*)
integer nblk
integer msg(*)

INPUT ARGUMENTS

buf	buffer into which data are to be scattered
nlist	list of the number of bytes in each block
ilist	list of the location in `buf` at which each data block starts
nblk	number of data blocks to be scattered

OUTPUT ARGUMENTS

msg	buffer from which the data to be scattered are unpacked

DESCRIPTION

MPI_GUNPACK takes `nblk` successive contiguous data blocks from the buffer `msg` and unpacks them into the buffer `buf` according to the information in the arrays `nlist` and `ilist`. The ith data block unpacked consists of `nlist(i)` contiguous bytes, and is copied to the `buf` so that the start of the block is aligned with the location `ilist(i)` bytes from the start of `buf`. It is assumed that all indices and numbering of data items begin at 0. The most common use of **MPI_GUNPACK** is to unpack a message received from another process. It is the responsibility of the user to ensure that `buf` is large enough to hold the data unpacked into it.

RETURN VALUE

Upon successful completion **MPI_GUNPACK** returns the total length of the message in bytes. Otherwise, -1 is returned.

A.4 Support for Communication Contexts

In this section specifications for the following routines for managing communication contexts are given,

- **MPI_INFOC** Get information on valid communication contexts
- **MPI_NEWC** Create a new communication context
- **MPI_POPC** Restore a communication context
- **MPI_PUSHC** Establish a new communication context

NAME

MPI_INFOC Get information about valid communication contexts

SYNOPSIS

integer function MPI_INFOC (maxlis, ilist)

integer maxlis

integer ilist(*)

INPUT ARGUMENTS

maxlis maximum number of communication context ID numbers in the array `ilist`

OUTPUT ARGUMENTS

ilist a list of communication context ID numbers

DESCRIPTION

MPI_INFOC determines the number of communication contexts that have been created for the current process group context, and returns a list of the corresponding communication context ID numbers in the array `ilist`. The first entry in `ilist` is always the ID number of the default communication context. If the number of ID numbers exceeds `maxlis`, then only `maxlis` are returned in the array `ilist`.

RETURN VALUE

On successful completion **MPI_INFOC** returns the number of communication contexts. Otherwise, -1 is returned.

NAME

MPI_NEWC Create a new communication context.

SYNOPSIS

integer function MPI_NEWC ( )

ARGUMENTS

None

DESCRIPTION

MPI_NEWC creates a new communication context within the scope of the current process group context.

RETURN VALUE

On successful completion **MPI_NEWC** returns the unique ID number of the new communication context. Otherwise -1 is returned.

NAME

MPI_POPC Re-establish former communication context.

SYNOPSIS

integer function MPI_POPC ( )

ARGUMENTS

None

DESCRIPTION

MPI_POPC re-establishes the communication context that was in effect before the preceding call to **MPI_PUSHC**.

RETURN VALUE

On successful completion **MPI_POPC** returns the ID number of the communication context that is re-established. Otherwise, -1 is returned.

NAME

MPI_PUSHC Establish a new communication context.

SYNOPSIS

integer function MPI_PUSHC (ccid)

integer ccid

INPUT ARGUMENTS

ccid	the ID number of the communication context to be established

DESCRIPTION

MPI_PUSHC sets the current communication context to that given by the communication context ID number, `ccid`. This communication context stays in effect until the subsequent corresponding call to **MPI_POPC**, or until the next call to **MPI_POPG**, which destroys all the communication contexts of the process group context being exited. **MPI_PUSHC** must be called by all processes in the current process group context.

RETURN VALUE

On successful completion **MPI_PUSHC** returns 0. Otherwise -1 is returned.

A.5 Utilities

In this section specifications for the following utility routines are given,

- **MPI_CPU** Get user CPU time in seconds
- **MPI_DATE** Get today's date as a character string
- **MPI_ERROR** Determine the current MPI error status
- **MPI_ETEXT** Get text string corresponding to error status
- **MPI_MACHINE** Get text string describing machine
- **MPI_INFOMN** Get process and machine characteristics
- **MPI_TIME** Get current time as a character string
- **MPI_WALL** Get elapsed wallclock time in seconds

NAME

MPI_CPU Determine CPU time used.

SYNOPSIS

double precision function MPI_CPU ( )

ARGUMENTS

None

DESCRIPTION

MPI_CPU determines the CPU time in seconds used by the calling process since the process was created.

RETURN VALUE

MPI_CPU returns the CPU time used in seconds by the calling process since the process was created.

NAME

MPI_DATE Determine today's date.

SYNOPSIS

character*8 function MPI_DATE ( )

ARGUMENTS

None

DESCRIPTION

MPI_DATE gives the date as an eight character string of the form "MM/DD/YY." If the two-digit integer, `YY`, giving the year is $\geq$ 90, then the actual year is obtained by adding 1900 to `YY`. Otherwise, 2000 is added to `YY` to calculate the actual year. Thus, for example, "06/30/90" signifies June 30, 1990, and "01/01/01" signifies January 1, 2001.

RETURN VALUE

MPI_DATE gives the date as an eight character string of the form "MM/DD/YY."

NAME

MPI_ERROR Determine error status following a call to MPI1

SYNOPSIS

integer function MPI_ERROR ( )

ARGUMENTS

None

DESCRIPTION

MPI_ERROR returns an integer giving the error status of the preceding call to an MPI1 routine.

RETURN VALUE

The meaning of the error status returned by **MPI_ERROR** is given in the table below. Additional entries may be added later.

Error status	Meaning
0	No error
1	Invalid PID used
2	Invalid GID used
3	Invalid MSGID used
4	Invalid CCID used
5	Invalid GCPID used
6	Invalid message buffer size
7	Invalid stride in MPI_SPACK/MPI_SUNPACK
8	Invalid block size in pack/unpack routine
9	Invalid data item size in pack/unpack routine
10	System buffer overflow
11	Too many communication contexts
12	Too many group contexts

NAME

MPI_ETEXT Give string describing the error status

SYNOPSIS

character*80 function MPI_ETEXT (ierrno)

integer ierrno

INPUT ARGUMENTS

ierrno The error status

DESCRIPTION

MPI_ETEXT gives a brief description of the error corresponding to the value of the error status integer `ierrno`.

RETURN VALUE

MPI_ETEXT returns a string describing the error status.

NAME

MPI_MACHINE Get machine name, version, and related information.

SYNOPSIS

character*80 function MPI_MACHINE ( )

ARGUMENTS

None

DESCRIPTION

MPI_MACHINE returns a character string giving the name of the machine that the calling process is running on, together with other information that may include the location of the machine, the type of machine, and similar site-specific details.

RETURN VALUE

MPI_MACHINE returns a character string giving details about the machine on which the calling process is running.

NAME

MPI_INFOMN Get information on the machine configuration.

SYNOPSIS

integer function MPI_INFOMN (maxlis, ilist)
integer maxlis
integer ilist(*)

INPUT ARGUMENTS

maxlis the size of the array `ilist`.

OUTPUT ARGUMENTS

ilist an integer array

DESCRIPTION

MPI_INFOMN returns in the array `ilist` a list of integers that characterize the machine that the calling process is running on. The first integer in `ilist` is the number of physical processors of the machine in use by the application, the second in the total number of processors in the machine. Other entries characterize the memory, I/O, and performance of the machine. The meaning of the entries in `ilist` is still under review.

RETURN VALUE

On successful completion **MPI_INFOMN** returns the number of entries in `ilist` that have been assigned a valid value. Otherwise, -1 is returned.

NAME

MPI_TIME Determine the current time.

SYNOPSIS

character*8 function MPI_TIME ( )

ARGUMENTS

None

DESCRIPTION

MPI_TIME gives the time as an eight character string of the form "HH:MM:SS" to the nearest second. A twenty-four hour clock is assumed for which "00:00:00" is midnight. Thus, for example, "15:30:00" signifies 3:30pm.

RETURN VALUE

MPI_TIME gives the time as an eight character string of the form "HH:MM:SS."

NAME

MPI_WALL Determine elapsed wallclock time in seconds.

SYNOPSIS

double precision function MPI_WALL ( )

ARGUMENTS

None

DESCRIPTION

MPI_WALL determines the wallclock time in seconds that has elapsed on the calling process since the process was created.

RETURN VALUE

MPI_WALL returns the wallclock time in seconds that has elapsed on the calling process since the process was created.