MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES

3 4456 0377879 4

ornl

ORNL/TM-12512

OAK RIDGE
NATIONAL
LABORATORY

MARTIN MARIETTA

The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers

David W. Walker

OAK RIDGE NATIONAL LABORATORY
CENTRAL RESEARCH LIBRARY
CIRCULATION SECTION
4500N ROOM 175
LIBRARY LOAN COPY
DO NOT TRANSFER TO ANOTHER PERSON
If you wish someone else to see this report, send in name with report and the library will arrange a loan.

MANAGED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831; prices available from (615) 576-8401, FTS 626-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-12512

465
22

Engineering Physics and Mathematics Division

Mathematical Sciences Section

THE DESIGN OF A STANDARD MESSAGE PASSING INTERFACE FOR DISTRIBUTED MEMORY CONCURRENT COMPUTERS

David W. Walker

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

Date Published: October 1993

Research was supported by the Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office.

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

MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES

3 4456 0377879 4

Contents

1 Introduction 1
2 An Overview of MPI 2
3 Details of MPI 3
3.1 Groups, Contexts, and Communicators 4
3.1.1 Process Groups 4
3.1.2 Communication Contexts 5
3.1.3 Communicator Objects 7
3.2 Application Topologies 7
3.3 Point-to-Point Communication 8
3.3.1 Message Selectivity 8
3.3.2 General Datatypes 9
3.3.3 Communication Completion 12
3.3.4 Persistent Communication Objects 13
3.4 Collective Communication 13
3.4.1 Collective Data Movement Routines 14
3.4.2 Global Computation Routines 16
4 Summary 16
5 References 17

THE DESIGN OF A STANDARD MESSAGE PASSING INTERFACE FOR DISTRIBUTED MEMORY CONCURRENT COMPUTERS

David W. Walker

Abstract

This paper presents an overview of MPI, a proposed standard message passing interface for MIMD distributed memory concurrent computers. The design of MPI has been a collective effort involving researchers in the United States and Europe from many organizations and institutions. MPI includes point-to-point and collective communication routines, as well as support for process groups, communication contexts, and application topologies. While making use of new ideas where appropriate, the MPI standard is based largely on current practice.

1. Introduction

This paper gives an overview of MPI, a proposed standard message passing interface for distributed memory concurrent computers. The main advantages of establishing a message passing interface for such machines are portability and ease-of-use, and a standard message passing interface is a key component in building a concurrent computing environment in which applications, software libraries, and tools can be transparently ported between different machines. Furthermore, the definition of a message passing standard provides vendors with a clearly defined set of routines that they can implement efficiently, or in some cases provide hardware or low-level system support for, thereby enhancing scalability.

The functionality that MPI is designed to provide is based on current common practice, and is similar to that provided by widely-used message passing systems such as Express [12], NX/2 [13], Vertex, [11], PARMACS [8,9], and P4 [10]. In addition, the flexibility and usefulness of MPI has been broadened by incorporating ideas from more recent and innovative message passing systems such as CHIMP [4,5], Zipcode [14,15], and the IBM External User Interface [7]. The general design philosophy followed by MPI is that while it would be imprudent to include new and untested features in the standard, concepts that have been tested in a research environment should be considered for inclusion. Many of the features in MPI related to process groups and communication contexts have been investigated within research groups for several years, but not in commercial or production environments. However, their incorporation into MPI is justified by the expressive power they bring to the standard.

The MPI standardization effort involves about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers are involved in MPI, along with researchers from universities, government laboratories, and industry. The standardization process began with the Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia [16]. At this workshop the basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process.

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [3]. MPI1 embodies the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. This proposal was intended to initiate discussion of standardization issues within the distributed memory concurrent computing community, and has served as a basis for the subsequent MPI standardization process. Since MPI1 was primarily intended to promote discussion and "get the ball rolling," it focuses mainly on point-to-point communications. MPI1 does not include any collective communication routines. MPI1 brought

to the forefront a number of important standardization issues, and has served as a catalyst for subsequent progress, however, its major deficiency is that the management of resources is not thread-safe. Although MPI1 and the MPI draft standard described in this paper have many features in common, they are distinct proposals, with MPI1 now being largely superseded by the MPI draft standard.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at which it was decided to place the standardization process on a more formal footing, and to generally adopt the procedures and organization of the High Performance Fortran forum. Subcommittees were formed for the major component areas of the standard, and an email discussion service established for each. In addition, the goal of producing a draft MPI standard by the Fall of 1993 was set. To achieve this goal the MPI working group has met every 6 weeks for two days throughout the first 9 months of 1993, and it is intended to present the draft MPI standard at the Supercomputing 93 conference in November 1993. These meetings and the email discussion together constitute the MPI forum, membership of which has been open to all members of the high performance computing community.

This paper is being written at a time when MPI is still in the process of being defined, but when the main features have been agreed upon. The only major exception concerns communication between processes in different groups. Some syntactical details, and the language bindings for Fortran-77 and C, have not yet been considered in depth, and so will not be discussed here. This paper is not intended to give a definitive, or even a complete, description of MPI. While the main design features of MPI will be described, limitations on space prevent detailed justifications for why these features were adopted. For these details the reader is referred to the MPI specification document, and the archived email discussions, which are available electronically as described in Section 4.

2. An Overview of MPI

MPI is intended to be a standard message passing interface for applications running on MIMD distributed memory concurrent computers. We expect MPI also to be useful in building libraries of mathematical software for such machines. MPI is not specifically designed for use by parallelizing compilers. MPI does not contain any support for fault tolerance, and assumes reliable communications. MPI is a message passing interface, not a complete parallel computing programming environment. Thus, issues such as parallel I/O, parallel program composition, and debugging are not addressed by MPI. In addition, MPI does not provide explicit support for active messages or virtual communication channels, although extensions for such features are not precluded, and may be made in the future. Finally, MPI provides no explicit support for multithreading, although one of the design goals of MPI was to ensure that it can be

implemented efficiently in a multithreaded environment.

The MPI standard does not mandate that an implementation should be interoperable with other MPI implementations. However, MPI does provide all the datatype information needed to allow a single MPI implementation to operate in a heterogeneous environment.

A set of routines that support point-to-point communication between pairs of processes forms the core of MPI. Routines for sending and receiving blocking and nonblocking messages are provided. A blocking send does not return until it is safe for the application to alter the message buffer on the sending process without corrupting or changing the message sent. A nonblocking send may return while the message buffer on the sending process is still volatile, and it should not be changed until it is guaranteed that this will not corrupt the message. This may be done by either calling a routine that blocks until the message buffer may be safely reused, or by calling a routine that performs a nonblocking check on the message status. A blocking receive suspends execution on the receiving process until the incoming message has been placed in the specified application buffer. A nonblocking receive may return before the message has been received into the specified application buffer, and a subsequent call must be made to ensure that this has occurred before the application uses the data in the message.

In MPI a message may be sent in one of three communication modes. The communication mode specifies the conditions under which the sending of a message may be initiated, or when it completes. In *ready* mode a message may be sent only if a corresponding receive has been initiated. In *standard* mode a message may be sent regardless of whether a corresponding receive has been initiated. Finally, MPI includes a *synchronous* mode which is the same as the standard mode, except that the send operation will not complete until a corresponding receive has been initiated on the destination process.

There are, therefore, 6 types of send operation and 2 types of receive, as shown in Figure 1. In addition, routines are provided that send to one process while receiving from another. Different versions are provided for when the send and receive buffers are distinct, and for when they are the same. The send/receive operation is blocking, so does not return until the send buffer is ready for reuse, and the incoming message has been received. The two send/receive routines bring the total number of point-to-point message passing routines up to 10.

3. Details of MPI

In this section we discuss the MPI routines in more detail. Since the point-to-point and collective communication routines depend heavily on the approach taken to groups and contexts, and to a lesser extent on process topologies, we shall discuss groups, contexts, and topologies first. These three related areas have generated much discussion within the MPI forum, and a consensus has emerged only in the last few weeks. To some extent this difficulty in arriv-

SEND	Blocking	Nonblocking
Standard	`mpi_send`	`mpi_isend`
Ready	`mpi_rsend`	`mpi_irsend`
Synchronous	`mpi_ssend`	`mpi_issend`

RECEIVE	Blocking	Nonblocking
Standard	`mpi_recv`	`mpi_irecv`

Figure 1: Classification and names of the point-to-point send and receive routines.

ing at a consensus arises because different commonly-used message passing interfaces generally handle groups, contexts, and topologies differently, and offer varying levels of support. The differing requirements in these three areas within the parallel computing community have also contributed to the diversity of views.

3.1. Groups, Contexts, and Communicators

Although it is now agreed within the MPI forum that groups and contexts should be bound together into abstract communicator objects, as described in Section 3.1.3, the precise details have yet to be worked out, particularly in the case of communicators for communication between groups. Thus, in this subsection we will give an overview of groups, contexts, and communicators, without going into specific details that may subsequently change. In particular, we will not discuss communication between processes in different groups as at the time of writing the precise details are still under discussion.

3.1.1. Process Groups

The prevailing view within the MPI forum is that a process group is an ordered collection of processes, and each process is uniquely identified by its rank within the ordering. For a group of n processes the ranks run from 0 to $n-1$. This definition of groups closely conforms to current practice.

Process groups can be used in two important ways. First, they can be used to specify which processes are involved in a collective communication operation, such as a broadcast. Second, they can be used to introduce task parallelism into an application, so that different groups perform different tasks. If this is done by loading different executable codes into each group, then we refer to this as MIMD task parallelism. Alternatively, if each group executes a different conditional branch within the same executable code, then we refer to this as SPMD task parallelism (also known as control parallelism). Although MPI does not provide mechanisms for loading executable codes onto processors, nor for creating processes and assigning them to

processors, each process may execute its own distinct code. However, it is expected that many initial MPI implementations will adopt a static process model, so that, as far as the application is concerned, a fixed number of processes exist from program initiation to completion, each running the same SPMD code.

Although the MPI process model is static, process groups are dynamic in the sense that they can be created and destroyed, and each process can belong to several groups simultaneously. However, the membership of a group cannot be changed asynchronously. For one or more processes to join or leave a group, a new group must be created which requires the synchronization of all processes in the group so formed. In MPI a group is an opaque object referenced by means of a handle. MPI provides routines for creating new groups by listing the ranks (within a specified parent group) of the processes making up the new group, or by partitioning an existing group using a key. The group partitioning routine is also passed an index, the size of which determines the rank of the process in the new group. This also provides a way of permuting the ranks within a group, if all processes in the group use the same value for the key, and set the index equal to the desired new rank. Additional routines give the rank of the calling process within a given group, test whether the calling process is in a given group, perform a barrier synchronization with a group, and inquire about the size and membership of a group. Other routines concerned with groups may be included in the final MPI draft.

3.1.2. Communication Contexts

Communication contexts, first used in the Zipcode communication system [14,15], promote software modularity by allowing the construction of independent communication streams between processes, thereby ensuring that messages sent in one phase of an application are not incorrectly intercepted by another phase. Communication contexts are particularly important in allowing libraries that make message passing calls to be used safely within an application. The point here is that the application developer has no way of knowing if the tag, group, and rank completely disambiguate the message traffic of different libraries and the rest of the application. Context provides an additional criterion for message selection, and hence permits the construction of independent tag spaces.

If communication contexts are not used there are two ways in which a call to a library routine can lead to unintended behavior. In the first case the processes enter a library routine synchronously when a send has been initiated for which the matching receive is not posted until after the library call. In this case the message may be incorrectly received in the library routine. The second possibility arises when different processes enter a library routine asynchronously, as shown in the example in Figure 2, resulting in nondeterministic behavior. If the program behaves correctly processes 0 and 1 each receive a message from process 2, using a wildcarded

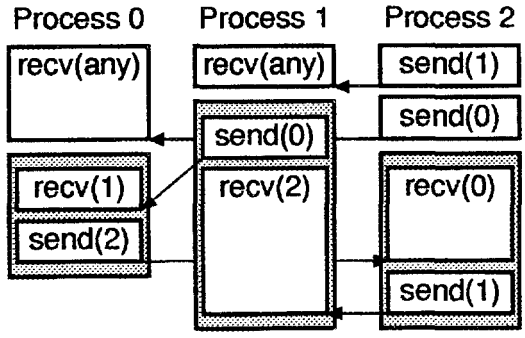

Figure 2: Use of contexts. Time increases down the page. Numbers in parentheses indicate the process to which data are being sent or received. The gray shaded area represents the library routine call. In this case the program behaves as intended. Note that the second message sent by process 2 is received by process 0, and that the message sent by process 0 is received by process 2.

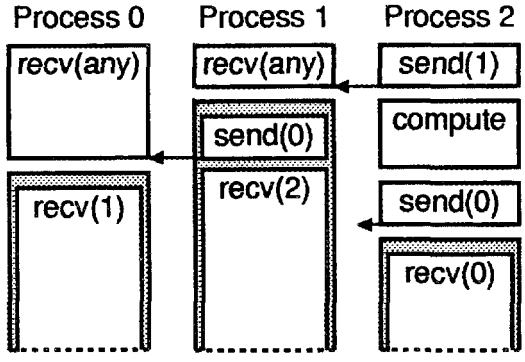

Figure 3: Unintended behavior of program. In this case the message from process 2 to process 0 is never received, and deadlock results.

selection criterion to indicate that they are prepared to receive a message from any process. The three processes then pass data around in a ring within the library routine. If communication contexts are not used this program may intermittently fail. Suppose we delay the sending of the second message sent by process 2, for example, by inserting some computation, as shown in Figure 3. In this case the wildcarded receive in process 0 is satisfied by a message sent from process 1, rather than from process 2, and deadlock results. By supplying a different communication context to the library routine we can ensure that the program is executed correctly, regardless of when the processes enter the library routine.

3.1.3. Communicator Objects

The "scope" of a communication operation is specified by the communication context used, and the group, or groups, involved. In a collective communication, or in a point-to-point communication between members of the same group, only one group needs to be specified, and the source and destination processes are given by their rank within this group. In a point-to-point communication between processes in different groups, two groups must be specified to define the scope. In this case the source and destination processes are given by their ranks within their respective groups. In MPI abstract opaque objects called "communicators" are used to define the scope of a communication operation. In intragroup communication involving members of the same group a communicator can be regarded as binding together a context and a group. The creation of intergroup communicators for communicating between processes in different groups is still under discussion within the MPI Forum, and so will not be discussed here.

3.2. Application Topologies

In many applications the processes are arranged with a particular topology, such as a two- or three-dimensional grid. MPI provides support for general application topologies that are specified by a graph in which processes that communicate a significant amount are connected by an arc. If the application topology is an n-dimensional Cartesian grid then this generality is not needed, so as a convenience MPI provides explicit support for such topologies. For a Cartesian grid periodic or nonperiodic boundary conditions may apply in any specified grid dimension. In MPI a group either has a Cartesian or graph topology, or no topology.

In MPI, application topologies are supported by an initialization routine, MPI_GRAPH or MPI_CART, that specifies the topology of a given group, a function MPI_INQRANK that determines the rank given a location in the topology associated with a group, and the inverse function MPI_INQLOC that determines where a process is in the topology. In addition, the routine MPI_INQMAP returns the topology associated with a given group, and for a group with a Cartesian topology, the routine MPI_INQCART gives the size and periodicity of the topology.

In addition to removing from the user the burden of having to write code to translate between process identifier, as specified by group and rank, and location in the topology, MPI also:

1. allows knowledge of the application topology to be exploited in order to efficiently assign processes to physical processors,

2. provides a routine MPI_PARTC for partitioning a Cartesian grid into hyperplane groups by removing a specified set of dimensions,

3. provides support for shifting data along a specified dimension of a Cartesian grid, and

By dividing a Cartesian grid into hyperplane groups it is possible to perform collective communication operations within these groups. In particular, if all but one dimension is removed a set of one-dimensional subgroups is formed, and it is possible, for example, to perform a multicast in the corresponding direction.

Support for shift operations is provided by a routine, MPI_SHIFT_ID, that returns the ranks of the processes that a process must send data to, and receive data from, when participating in the shift. Once the source and destination process are known for each process, the shift is performed by calling the routine MPI_SENDRECV that allows each process to send to one process while receiving from another. In a circular shift each process sends data to the process whose location in the given dimension is obtained by adding a specified integer (which may be negative) to its own location, modulo the number of processes in that dimension. In an end-off shift each process determines the rank of its destination process by adding a specified integer to its own rank, but if this exceeds the number of processes in the given dimension, or is less than zero, then no data are sent. If the Cartesian grid is periodic in the dimension in which the shift is done, then MPI_SHIFT_ID returns source and destination processes appropriate for a circular shift. Otherwise MPI_SHIFT_ID returns source and destination processes appropriate for an end-off shift.

3.3. Point-to-Point Communication

3.3.1. Message Selectivity

In MPI a process involved in a communication operation is identified by group and rank with that group. Thus,

$$\text{Process ID} \equiv (\text{group, rank})$$

In point-to-point communication, messages may be considered labeled by communication context and message tag within that context. Thus,

$$\text{Message ID} \equiv (\text{context, tag})$$

When sending or receiving a message the process and message identifiers must be specified. The group and context, which define the scope of the communication operation, are specified by means of a communicator object in the argument list of the send and receive routines. The rank and tag also appear in the argument list. A message sent in one scope can only be received in a different scope, so the communicator objects specified by the send and receive routines must match. The group and context components of a communicator may not be wildcarded. Within a given scope, message selectivity is by rank and tag. Either, or both, of these may be wildcarded by a receiving process to indicate that the corresponding selection criterion is to be

```
MPI_SEND (
   IN start_of_buffer
   IN number_of_items
   IN datatype_of_items
   IN destination_rank
   IN tag
   IN communicator)
```

```
MPI_RECV (
   OUT start_of_buffer
   IN max_number_of_items
   IN datatype_of_items
   IN source_rank
   IN tag
   IN communicator
   OUT return_status_object)
```

Figure 4: Argument lists for the blocking send and receive routines.

ignored. The argument lists for the block send and receive routines are shown in Figure 4.

In Figure 4, the last argument to MPI_RECV is a handle to a return status object. This object may passed to an inquiry routine to determine the length of the message, or the actual source rank and/or message tag if wildcards have been used. The argument lists for the nonblocking send and receives are very similar, except that each returns a handle to an object that identifies the communication operation. This object is used subsequently to check for completion of the operation. In addition, the nonblocking receive does not return a return status object. Instead the return status object is returned by the routine that confirms completion of the receive operation.

3.3.2. General Datatypes

All point-to-point message passing routines in MPI take as an argument the datatype of the data communicated. In the simplest case this will be a primitive datatype, such as an integer or floating point number. However, MPI also supports more general datatypes, and thereby supports the communication of *array sections and structures involving combinations of primitive* datatypes.

A general datatype is a sequence of pairs of primitive datatypes and integer byte displacements. Thus,

$$\text{Datatype} = \{ (\text{type}_0, \text{disp}_0), (\text{type}_1, \text{disp}_1), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1}) \}$$

Together with a base address, a datatype specifies a communication buffer. General datatypes are built up hierarchically from simpler components. There are four basic constructors for datatypes, namely the contiguous, vector, indexed, and structure constructors. We will now discuss each of these in turn.

The *contiguous* constructor creates a new datatype from repetitions of a specified old datatype. This requires us to specify the old datatype and the number of repetitions, n. For example, if the old datatype is **oldtype** $= \{$ (**double**, 0), (**char**, 8) $\}$ and $n = 3$, then the new datatype would be,

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

It should be noted how each repeated unit in the new datatype is aligned with a double word boundary. This alignment is dictated by the appearance of a **double** in the old datatype, so that the extent of the old datatype is taken as 16 bytes, rather than 9 bytes.

The *vector* constructor builds a new datatype by replicating an old datatype in blocks at fixed offsets. The new datatype consists of **count** blocks, each of which is a repetition of **blocklen** items of some specified old datatype. The starts of successive blocks are offset by **stride** items of the old datatype. Thus, if **count** $= 2$, **blocklen** $= 3$, and **stride** $= 4$ then the new datatype would be,

$$\{ (\texttt{double}, 0), (\texttt{char}, 8), (\texttt{double}, 16), (\texttt{char}, 24), (\texttt{double}, 32), (\texttt{char}, 40), \\ (\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104) \}$$

Here the offset between the two blocks is 64 bytes, which is the stride multiplied by the extent of the old datatype.

The *indexed* constructor is a generalization of the vector constructor in which each block has a different size and offset. The sizes and offsets are given by the entries in two integer arrays, `B` and `I`. The new datatype consists of **count** blocks, and the ith block is of length `B[i]` items of the specified old datatype. The offset of the start of the ith block is `I[i]` items of the old datatype. Thus, if **count** $= 2$, `B` $= \{3, 1\}$, and `I` $= \{64, 0\}$, then the new datatype would be,

$$\{ (\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104), \\ (\texttt{double}, 0), (\texttt{char}, 8) \}$$

The *structure* constructor is the most general of the datatype constructors. This constructor generalizes the indexed constructor by allowing each block to be of a different datatype. Thus,

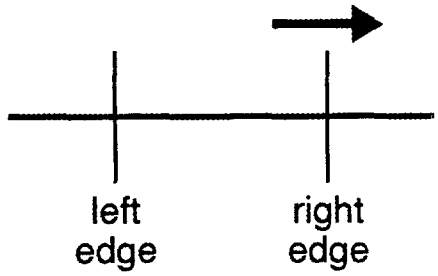

Figure 5: Particle migration in a one-dimensional code. The left and right edges of a process domain are shown. We shall consider just the migration of particles across the righthand boundary.

in addition to specifying the number of blocks, `count`, and the block length and offset arrays, `B` and `I`, we must also give the datatype of the replicated unit in each block. Let us assume this is specified in an array `T`. The length of the ith block is `B[i]` items of type `T[i]`, and the offset of the start of the ith block is `I[i]` bytes. Thus, if `count=3`, `T` = {`MPI_FLOAT`, `oldtype`, `MPI_CHAR`}, `I` = $\{0, 16, 26\}$, and `B` = $\{2, 1, 3\}$, then the new datatype would be,

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

In addition to the constructors described above, there is a variant of the vector constructor in which the stride is given in bytes instead of the number of items. There is also a variant of the indexed constructor in which the block offsets are given in bytes.

To better understand the use of general data structures consider the example of an application in which particles move on a one-dimensional domain. We assume that each process is responsible for a different section of this domain. In each time step particles may move from the subdomain of one process to that of another, and so the data for such particles must be communicated between processes. We shall just consider here the task of migrating particles across the righthand boundary of a process, as shown in Figure 5. The particle data are stored in an array of structures, with each entry in this structure consisting of the particle position, `x`, velocity, `v`, and type, `k`:

```
struct Pstruct { double x; double v; int k; };
```

The C code for migrating particles across the righthand boundary is shown in Figure 6.

In Figure 6 the code in the first box creates a datatype, `Ptype`, that represents the `Pstruct` structure for a single particle. This datatype is,

```
Ptype = {(double,0), (double,8), (int,16)}
```

In the second code box the particles that have crossed the righthand boundary are identified,

```
struct Pstruct particle[1000];
MPI_datatype Ptype, Ztype;
MPI_datatype Stype[3]={MPI_double, MPI_double, MPI_int};
int Sblock[3]={1, 1, 1};
int Sindex[3];
int Pindex[100];
int Pblock[100];
Sindex[0] = 0;
Sindex[1] = sizeof(double);
Sindex[2] = 2*sizeof(double);
MPI_type_struct (3, Stype, Sindex, Sblock, &Ptype);
j=0;
for (i=0;i<1000;i++)
  if (x[i] > right_edge) {
    Pindex[j] = i;
    Pblock[j] = 1;
    j++;}
MPI_type_indexed (j, Ptype, Pindex, Pblock, &Ztype);
MPI_type_commit (Ztype);
MPI_send (particle, 1, Ztype, dest, tag, comm);
```

Figure 6: Fragment of C code for migrating particles across the righthand process boundary

and their index in the **particle** array is stored in **Pindex**. It is assumed that no more than 100 particles cross the boundary. The call to **MPI_type_indexed** uses an indexed constructor to create a new datatype, **Ztype**, that references all the migrating particles. Before sending the data, the **Ztype** datatype must be committed. This is done to allow the system to use a different internal representation for **Ztype**, and to optimize the communication operation. Committing a datatype is most likely to be advantageous when reusing a datatype many times, which is not the case in this example. Finally, the migrating particles are sent to their destination process, **dest**, by a call to **MPI_send**. The offsets in the **Ztype** datatype are interpreted relative to the address of the start of the **particle** array.

3.3.3. Communication Completion

Following a call to a nonblocking send or receive routine there are a number of ways in which the handle returned by the call can be used to check the completion status of the communication operation, or to suspend further execution until the operation is complete. MPI_WAIT does not return until the communication operation referred to by the input handle is complete. MPI_TEST does not wait until the operation identified by the input handle is complete, but instead returns a logical variable that is TRUE if the operation is complete, and FALSE otherwise. If the input handle refers to a receive operation, then MPI_WAIT and MPI_TEST both return a handle to a

return status object. This handle can subsequently be passed to a query routine to determine the actual source, tag, and length of the message received.

An additional two routines exist for waiting for the completion of any or all of the handles in a list of handles. Similarly, there are variants of the test routine that check if all, or at least one, of the communication operations identified by a list of handles is complete.

3.3.4. Persistent Communication Objects

MPI also provides a set of routines for creating communication objects that completely describe a send or receive operation by binding together all the parameters of the operation. A handle to the communication object so formed is returned, and may subsequently be passed to the routine MPI_START to actually initiate the communication. The MPI_WAIT routine, or a similar completion routine, must be called to ensure completion of the operation, as discussed in Section 3.3.3.

Persistent communication objects may be used to optimize communication performance, particularly when the same communication pattern is repeated many times in an application. For example, if a send routine is called within a loop, performance may be improved by creating a communication object that describes the parameters of the send prior to entering the loop, and then calling MPI_START inside the loop to send the data on each pass through the loop.

There are four routines for creating communication objects: three for send operations, corresponding to the standard, ready, and synchronous modes, and one for receive operations. A persistent communication object must be deallocated when no longer needed.

3.4. Collective Communication

Collective communication routines provide for coordinated communication among a group of processes [1,2]. The process group is given by the communicator object that is input to the routine. The MPI collective communication routines have been designed so that their syntax and semantics are consistent with those of the point-to-point routines. The collective communication routines may, but do not have to be, implemented using the MPI point-to-point routines. Collective communication routines do not have message tag arguments, though an implementation in terms of the point-to-point routines may need to make use of tags. A collective communication routine must be called by all members of the group with consistent arguments. As soon as a process has completed its role in the collective communication it may continue with other tasks. Thus, a collective communication is not necessarily a barrier synchronization for the group. MPI does not include nonblocking forms of the collective communication routines. MPI collective communication routines are divided into two broad classes: data movement routines, and global computation routines.

3.4.1. Collective Data Movement Routines

There are 3 basic types of collective data movement routine: broadcast, scatter, and gather. There are two versions of each of these three routines: in the one-all case data are communicated between one process and all others; in the all-all case data are communicated between each process and all others. Figure 7 shows the one-all and all-all versions of the broadcast, scatter, and gather routines for a group of six processors.

The all-all broadcast, and both varieties of the scatter and gather routines, involve each process sending distinct data to each process, and/or receiving distinct data from each process. In these routines each process may send to and/or receive from each other process a different number of data items, but the send and receive datatypes must be consistent. To illustrate this point consider the following example in which process 0 gathers data from processes 1 and 2. Suppose the receive datatype in process 0, and the send datatypes in processes 1 and 2 are as follows,

In process 0: **recvtype**={(**int**, 0), (**float**, 4)}

In process 1: **sendtype**={(**int**, 0), (**float**, 4), (**int**, 96), (**float**, 100), (**int**, 32), (**float**, 36)}

In process 2: **sendtype**={(**int**, 16), (**float**, 20), (**int**, 48), (**float**, 52)}

Such a situation could arise in a C program in which an indexed datatype constructor has been applied to an array of structures, each element of which consists of an integer and a floating-point number. Although the datatypes are different in each process, they are *type consistent*, since each consists of repetitions of an integer followed by a float.

The one-all broadcast routine broadcasts data from one process to all other processes in the group. The all-all broadcast broadcasts data from each process to all others, and on completion each has received the same data. Thus, for the all-all broadcast each process ends up with the same output data, which is the concatenation of the input data of all processes, in rank order.

The one-all scatter routine sends distinct data from one process to all processes in the group. This is also known as "one-to-all personalized communication". In the all-all scatter routine each process scatters distinct data to all processes in the group, so the processes receive different data from each process. This is also known as "all-to-all personalized communication".

The communication patterns in the gather routines are the same as in the scatter routines, except that the direction of flow of data is reversed. In the one-all gather routine one process (the root) receives data from every process in the group. In the root process receives the concatenation of the input buffers of all processes, in rank order. There is no separate all-all gather routine since this would just be identical to the all-all scatter routine, so there are 5 basic data movement routines.

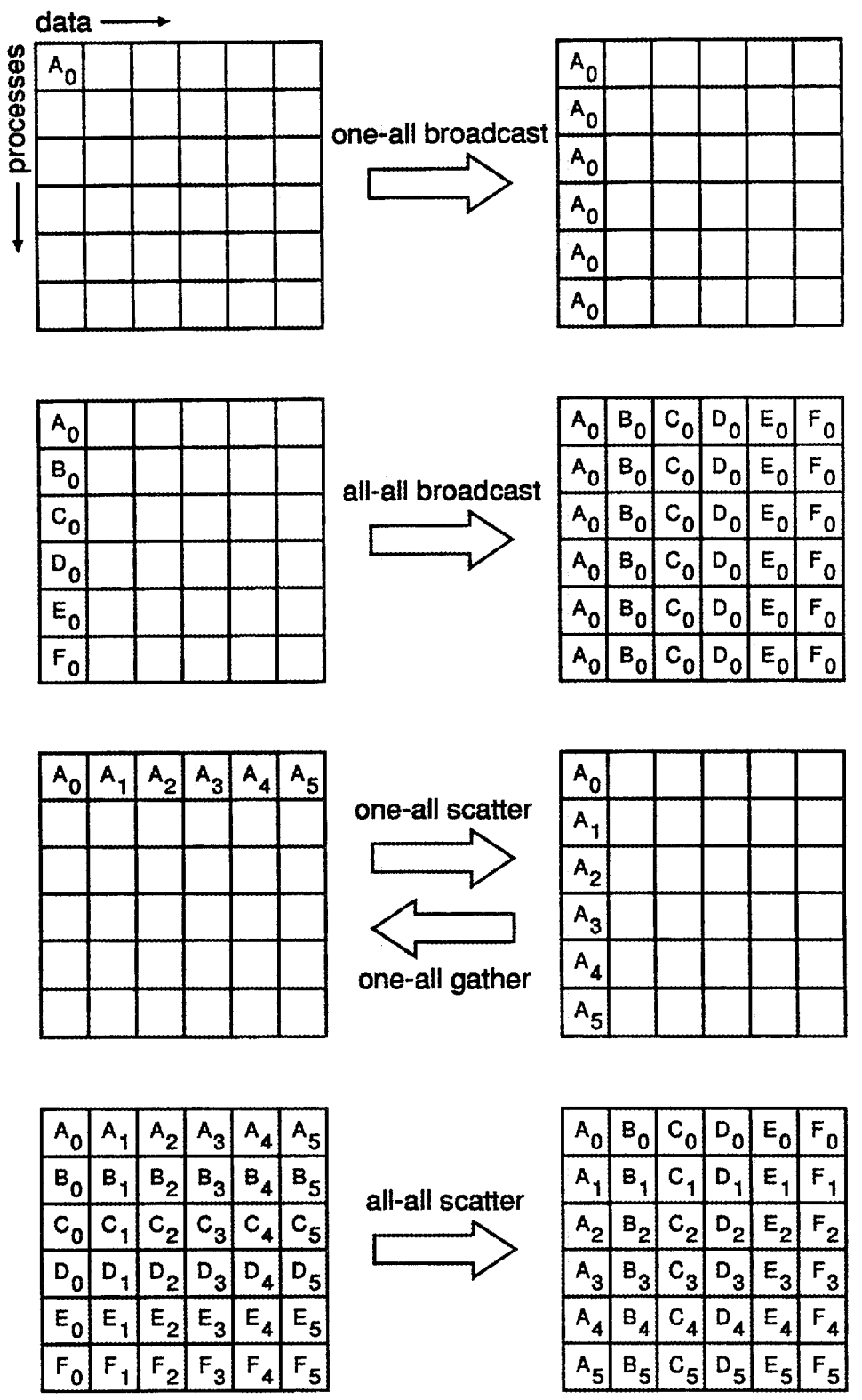

Figure 7: One-all and all-all versions of the broadcast, scatter, and gather routines for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the one-all broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

In addition, MPI provides versions of all these 5 routines, except the one-all broadcast, in which the send and receive datatypes are type consistent as discussed above, but in which each process is allocated a *fixed* size portion of the communication buffer. These bring the total number of data movement routines to 9.

3.4.2. Global Computation Routines

There are two basic global computation routines in MPI: reduce and scan. The reduce and scan routines both require the specification of an input function. One version is provided in which the user selects the function from a predefined list; in the second version the user supplies (a pointer to) a function that is associative and commutative; in the third version the user supplies (a pointer to) a function that is associative, but not necessarily commutative. In addition, there are three variants of the reduction routines. In one variant the reduced results are returned to a single specified process; in the second variant the reduced results are returned to all processes involved; and, in the third variant the reduced results are scattered across the processes involved. This latter variant is a generalization of the `fold` routine described in Chapter 21 of [6]. Thus, there are 12 global computation routines, and a total of 21 collective communication routines (or 22 if we include the routine for performing a barrier synchronization over a process group).

The reduce routines combine the values provided in the input buffer of each process using a specified function. Thus, if D_i is the data in the process with rank i in the group, and $\oplus$ is the combining function, then the following quantity is evaluated,

$$\mathcal{D} = D_0 \oplus D_1 \oplus D_2 \oplus \cdots \oplus D_{n-1} \tag{1}$$

where n is the size of the group. Common reduction operations are the evaluation of the maximum, minimum, or sum of a set of values distributed across a group of processes.

The scan routines perform a parallel prefix with respect to an associative reduction operation on data distributed across a specified group. On completion the output buffer of the process with rank i contains the result of combining the values from the processes with rank $0, 1, \ldots, i$, i. e.,

$$\mathcal{D}_i = D_0 \oplus D_1 \oplus D_2 \oplus \cdots \oplus D_i \tag{2}$$

It should be noted that segmented scans can be performed by first creating distinct subgroups for each segment.

4. Summary

This paper has given an overview of the main features of MPI, but has not described the detailed syntax of the MPI routines, or discussed language binding issues. These will be fully

discussed in the MPI specification document, a draft of which is expected to be available by the Supercomputing 93 conference in November 1993.

The design of MPI has been a cooperative effort involving about 60 people. Much of the discussion has been by electronic mail, and has been archived, along with copies of the MPI draft and other key documents. Copies of the archives and documents may be obtained by netlib. For details of what is available, and how to get it, please send the message "send index from mpi" to `netlib@ornl.gov`.

Acknowledgments

Many people have contributed to MPI , so it is not possible to acknowledge them all individually. However, many of the ideas presented in this paper are due to the MPI subcommittee chairs: Scott Berryman, James Cownie, Jack Dongarra, Al Geist, William Gropp, Rolf Hempel, Steve Huss-Lederman, Anthony Skjellum, Marc Snir, and Steven Zenith. Lyndon Clarke, Bob Knighten, Rik Littlefield, and Rusty Lusk have also made important contributions, as has also Steve Otto, the editor of the MPI specification document.

5. References

[1] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and Marc Snir. Ccl: A portable and tunable collective communication library for scalable parallel computers. Technical report, IBM T. J. Watson Research Center, 1993. Preprint.

[2] J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and Marc Snir. Ccl: A portable and tunable collective communication library for scalable parallel computers. Technical report, IBM Almaden Research Center. 1993. Preprint.

[3] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

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

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

[6] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, N.J., 1988.

[7] D. Frye, R. Bryant, H. Ho, R. Lawrence, and M. Snir. An external user interface for scalable parallel systems. Technical report, IBM, May 1992.

[8] 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.

[9] 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.

[10] Ewing Lusk, Ross Overbeek, et al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

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

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

[13] 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.

[14] 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.

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

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

INTERNAL DISTRIBUTION

1. B. R. Appleton
2. J. Choi
3–4. T. S. Darland
5. E. F. D'Azevedo
6. J. J. Dongarra
7. G. A. Geist
8. L. J. Gray
9. M. R. Leuze
10. E. G. Ng
11. C. E. Oliver
12. B. W. Peyton
13–17. S. A. Raby
18. C. H. Romine
19. T. H. Rowan
20–24. R. F. Sincovec
25–29. D. W. Walker
30–34. R. C. Ward
35. P. H. Worley
36. Central Research Library
37. ORNL Patent Office
38. K-25 Applied Technology Library
39. Y-12 Technical Library
40. Laboratory Records - RC
41–42. Laboratory Records Department

EXTERNAL DISTRIBUTION

43. Thomas A. Adams, NCC and OSC / NRaD, Code 733, 271 Catalina Blvd., San Diego, CA 92152-5000

44. Robert J. Allen, Daresbury Laboratory, S.E.R.C., Daresbury, Warrington WA4 4AD, United Kingdom

45. Giovanni Aloisio, Dipt. di Elettrotecnica ed Elettronica, Universita di Bari, Via Re David 200, 70125 Bari, Italy

46. Ed Anderson, Mathematical Software Group, Cray Research Incorporated, 655F Lone Oak Drive, Eagan, MN 55121

47. Mark Anderson, Rice University, Department of Computer Science, P. O. Box 1892, Houston, TX 77251

48. Ian G. Angus, Boeing Computer Services, M/S 7L-22, P. O. Box 24346, Seattle, WA 98124-0346

49. Marco Annaratone, Digital Equipment Corporation, 146 Main Street MLO1-5/U46, Maynard, MA 01754

50. Vasanth Bala, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598

51. Donald M. Austin, 6196 EECS Bldg ., University of Minnesota, 200 Union Street, S.E., Minneapolis, MN 55455

52. Joseph G. Baron, IBM Corporation, AWS Advanced Product Development, 11400 Burnet Road, Austin, TX 78758-3493

53. Edward H. Barsis, Computer Science and Mathematics, P. O. Box 5800, Sandia National Laboratories, Albuquerque, NM 87185

54. Eric Barszcz, Mail Stop T-045, NASA Ames Research Center, Moffet Field, CA 94035

55. Eric Barton, Meiko Limited, 650 Aztec West, Bristol BS12 4SD, United Kingdom

56. A. Basu C-DAC 2/1 Brunton Road Bangalore 560 025 India

57. Adam Beguelin, Carnegie Mellon University, School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213-3890

58. Siegfried Benker, Institute for Statistics and Computer Science, University of Vienna, A-1210 Vienna, Austria

59. Ed Benson, Digital Equipment Corp., 146 Main Street, ML01-5/U46, Maynard, MA 01754

60. Roger Berry, NCUBE Corporation, 4313 Prince Road, Rockville, MD 20853

61. Scott Berryman, Yale University, Computer Science Department, 51 Prospect Street, New Haven, CT 06520

62. Biondo Biondi, Stanford University, Department of Geophysics, Stanford, CA 94305

63. Robert Bjornson, Department of Computer Science, Box 2158 Yale Station, New Haven, CT 06520

64. Peter Brezany, Institute for Statistics and Computer Science, University of Vienna, A-1210 Vienna, Austria

65. Roger W. Brockett, Wang Professor of EE and CS, Division of Applied Sciences, Harvard University, Cambridge, MA 02138

66. Eric Browne, University of Cambridge, Department of Earth Sciences, Downing Street, Cambridge CB2 3EQ, United Kingdom

67. Clemens H. Cap, University of Zurich, Department of Computer Science, Winterthurerstr. 190, CH-8057 Zurich, Switzerland

68. Trevor Carden, Parsys Ltd., Boundary House, Boston Road, London W7 2QE, United Kingdom

69. Siddhartha Chatterjee, RIACS, Mail Stop T045-1, NASA Ames Research Center, Moffett Field, CA 94035-1000

70. Hsing-bung Chen, University of Texas-Arlington, CSE Department, Box 19015, Arlington, TX 76019

71. Doreen Y. Cheng, Computer Science Corporation, NASA Ames Research Center, Mail Stop 258-6, Moffett Field, CA 94035

72. Kuo-Ning Chiang, National Center for High-Performance Computing, P.O. Box 19-136, Hsinchu, Taiwan R.O.C

73. Lyndon Clarke, Edinburgh Parallel Computing Centre, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom

74. Robert Cohen, Department of Computer Science, Australian National University, GPO Box 4, Canberra 2601, Australia

75. Michele Colajanni, Dip. di Ingegneria Elettronica, Universita' di Roma "Tor Vergata" Via della Ricerca Scientifica, 00133 - Roma, Italy

76. Jeremy Cook, Parallel Processing Laboratory, Dept. of Informatics, University of Bergen, High Technology Centre, N-5020 Bergen, Norway

77. Manuel Eduardo C. D. Correia, Centro de Informatica, Universidade do Porto (CIUP), Rua do Campo Alegre 823, 4100 Porto, Portugal

78. Jim Cownie, Meiko Limited, 650 Aztec West, Bristol BS12 4SD, United Kingdom

79. Michel Dayde, CERFACS, 42 Avenue G. Coriolis, 31057 Toulouse Cedex, France

80. David DiNucci, Computer Sciences Corporation, NASA Ames Research Center, M/S 258-6, Moffet Field, CA 94035

81. Mark Debbage, University of Southampton, Dept. of Electronics and Computer Science, Highfield, Southampton SO9 5NH, United Kingdom

82. Dominique Duval, Telmat Informatique, BP 12, Rue de l'Industrie, 68360 Soultz, France

83. Tom Eidson, Theoretical Flow Physics Branch, M/S 156, NASA Langley Research Center, Hampton, VA 23665

84. Victor Eijkhout, University of Tennessee, 107 Ayres Hall, Department of Computer Science, Knoxville, TN 37996-1301

85. Anne Elster, Cornell University, Xerox DRI, 502 Engineering and Theory Center, Ithaca, NY 14853

86. Rob Falgout, Lawrence Livermore National Lab, L-419, P. O. Box 808, Livermore, CA 94551

87. Jim Feeney, IBM Endicott, R. D. 3, Box 224, Endicott, NY 13760

88. Edward Felten, Department of Computer Science, University of Washington, Seattle, WA 98195

89. Vince Fernando, NAG Limited, Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, United Kingdom

90. Sam Fineberg, NASA Ames Research Center, M/S 258-6, Moffett Field, CA 94035-1000

91. Randy Fischer, 615 NW 32st Place, Gainesville, FL 32607

92. Jon Flower, Parasoft Corporation, 2500 E. Foothill Blvd., Suite 205, Pasadena, CA91107

93. David Forslund, Los Alamos National Laboratory, Advanced Computing Laboratory, MS B287, Los Alamos, NM 87545

94. Geoffrey C. Fox, Syracuse University, Northeast Parallel Architectures Center, 111 College Place, Syracuse, NY 13244-4100

95. Lars Frellesen, Math-Tech Aps, Kildeskovsvej 67, 2820 Gentofte, DK - Denmark

96. Josef Fritscher, Computing Center, Technical University of Vienna, Wiedner Hauptstrasse 8-10, A-1040 Vienna, Austria

97. Daniel D. Frye, IBM Corporation, Dept. 49NA / MS 614, Neighborhood Road, Kingston, NY 12401

98. Kyle Gallivan, University of Illinois, CSRD, 465 CSRL, 1308 West Main Street, Urbana, IL 61801-2307

99. J. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

100. Mike Gerndt, Zentralinstitut fuer Angewandte Mathematik, Forschungszentrum Juelich GmbH, Postfach 1913, D-5170 Juelich, Germany

101. Ian Glendinning, University of Southampton, Dept. of Electronics and Comp. Sci., Southampton SO9 5NH, United Kingdom

102. Gene H. Golub, Department of Computer Science, Stanford University, Stanford, CA 94305

103. Adam Greenberg, Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214

104. Robert Greimel, AVL List Gmbh., Department TSS, Kleiststrasse 48, A-8020 Graz, Austria,

105. William Gropp, Argonne National Laboratory, Mathematics and Computer Science, 9700 South Cass Avenue, MCS 221, Argonne, IL 60439-4844

106. Sanjay Gupta, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665-5225

107. John Gustafson, 236 Wilhelm, Ames Laboratory, Iowa State University, Ames, IA 50011

108. Fred Gustavson, IBM T. J. Watson Research Center, Room 33-260, P. O. Box 218, Yorktown Heights, NY 10598

109. Robert Halstead, Digital Equipment Corporation, Cambridge Research Lab., One Kendall Sq. Bldg. 700, Cambridge, MA 02139

110. Robert J. Harrison, Battelle Pacific Northwest Laboratory, Mail Stop K1-90, P. O. Box 999, Richland, WA 99352

111. Leslie Hart, NOAA/FSL, R/E/FS5, 325 Broadway, Boulder, CO 80303

112. Tom Haupt, Syracuse University, Northeast Parallel Architectures Center, 111 College Place, Syracuse, NY 13244-4100

113. Michael Heath, University of Illinois, NCSA, 4157 Beckman Institute, 405 North Mathews Avenue, Urbana, IL 61801-2300

114. Don Heller, Center for Research on Parallel Computation, Rice University, P.O. Box 1892, Houston, TX 77251

115. Rolf Hempel, GMD, Schloss Birlinghoven, Postfach 13 16, D-W-5205 Sankt Augustin 1, Germany

116. Tom Henderson, NOAA/FSL, R/E/FS5, 325 Broadway, Boulder, CO 80303

117. Anthony J. G. Hey, University of Southampton, Dept. of Electronics and Comp. Sci., Southampton SO9 5NH, United Kingdom

118. Mark Hill, University of Southampton, Dept. of Electronics and Comp. Sci., Southampton SO9 5NH, United Kingdom

119. C. T. Howard Ho, IBM Almaden Research Center, K54/802, 650 Harry Road, San Jose, CA 95120

120. Randy Holmes, IBM Corporation, High Performance Computing Services, 1507 LBJ Freeway, Dallas, TX 75234

121. Gary W. Howell, Florida Institute of Technology, Department of Applied Mathematics, 150 W. Univeristy Blvd., Melbourne, FL 32901

122. Chengchang Huang, 2814 Beau Jardin, Apt. 301, Lansing, MI 48910

123. Steve Huss-Lederman, Supercomputing Research Center, 17100 Science Drive, Bowie, MD 20715-4300

124. Joefon Jann, IBM T.J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598

125. S. Lennart Johnsson, Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214

126. Charles Jung, IBM Kingston, 67LB/MS 614, Neighborhood Road, Kingston, NY 12401

127. Edgar T. Kalns, Michigan State University, Advanced Computing Systems Lab, Department of Computer Science, East Lansing, MI 48824

128. Malvyn H. Kalos, Cornell Theory Center, Engineering and Theory Center Bldg., Cornell University, Ithaca, NY 14853-3901

129. John Kapenga, Department of Computer Science, Western Michigan Universuty, Kalamazoo, MI 49008

130. Hans Kaper, Mathematics and Computer Science Division, Argonne National Laboratory, Bldg. 221, 9700 South Cass Avenue, Argonne, IL 60439

131. Udo Keller, PALLAS GmbH, Hermuelheimer Strasse 10, D-W5040 Bruehl, Germany

132. Ken Kennedy, Rice University, Department of Computer Science, P. O. Box 1892, Houston, TX 77251

133. Ronan Keryell, Ecole Nationale Superieure des Mines de Paris, Centre de Recherche en Informatique, 35, Rue Saint-Honore, 77305 Fontainebleau Cedex, France

134. Shlomo Kipnis, IBM T. J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598

135. Robert L. Knighten, Intel Corporation, Supercomputer Systems Division, 15201 NW Greenbrier Parkway, Beaverton, OR 97006

136. Charles Koelbel, Rice University, CITI/CRPC, P. O. Box 1892, Houston, TX 77251

137. Edward Kushner, Intel Corporation, 15201 NW Greenbrier Parkway, Beaverton, OR 97006

138. Pierre Lagier, 24, Avenue de l'Europe, 78141 Velizy Villacoublay, France

139. Derryck Lamptey, National Transputer Support Centre, University of Sheffield, Sheffield, United Kingdom

140. Falk Langhammer, Parsytec Computer GmbH, Juelicher Strasse 338, D-5100 Aachen, Germany

141. Randolph Langley, Florida State University, 400 SCL, B-186, Tallahassee, FL 32306

142. Bob Leary, San Diego Supercomputer Center, P. O. Box 85608, San Diego, CA 92186-9784

143. Bruce Leasure, Kuck and Associates, Inc., 1906 Fox Drive, Champaign, IL 61820

144. James E. Leiss, Rt. 2, Box 142C, Broadway, VA 22815

145. Eric Leu, IBM Almaden Research Center, 650 Harry Road K54/802, San Jose, CA 95123

146. David Levine, Argonne National Laboratory, MCS 221 C-216, Argonne, IL 60439

147. John Lewis, Boeing Computer Services , Mail Stop 7L-21, P. O. Box 24346, Seattle, WA 98124-0346

148. David Linden, Digital Equipment Corp., 146 Main Street, ML01-5/U46, Maynard, MA 01754

149. Rik Littlefield, Battelle Pacific Northwest Laboratory, Mail Stop K1-87, P. O. Box 999, Richland, WA 99352

150. Miron Livny, University of Wisconsin, Department of Computer Science, 1210 West Dayton Street, Madison, WI 53706

151. Rusty Lusk, Argonne National Laboratory, Mathematics and Computer Science, 9700 South Cass Avenue, MCS 221, Argonne, IL 60439-4844

152. Arthur B. Maccabe, Sandia National Labs, Dept. 1424, Albuquerque, NM 87185-5800

153. Neil MacDonald, Edinburgh Parallel Computing Centre, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom

154. Peter Madams, nCUBE Corporation, 919 East Hillsdale Blvd., Foster City, CA 94404

155. Amitava Majumdar, University of Michigan, Department of Nuclear Engineering, Ann Arbor, MI 48109

156. David P. Mallon, Leeds University, School of Computer Studies, Leeds LS2 9JT, United Kingdom

157. Dan Cristian Marinescu, Computer Sciences Department, Purdue University, West Lafayette, IN 47907

158. Tim Mattson, Scientific Computing Associates, Inc., 265 Church Street, New Haven, CT 06510-7010

159. Oliver McBryan, University of Colorado at Boulder, Department of Computer Science, Campus Box 425, Boulder, CO 80309-0425

160. Robert McLay, University of Texas at Austin, Dept ASE-EM 60600, Austin, TX 78712

161. James McGraw, Lawrence Livermore National Laboratory, L-306, P. O. Box 808, Livermore, CA 94550

162. Phil McKinley, A714 Wells Hall, Michigan State University, East Lansing, MI 48824

163. Piyush Mehrotra, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665

164. Paul Messina, California Institute of Technology, Mail Stop 158-79, 1201 E. California Boulevard, Pasadena, CA 91125

165. Moataz Mohamed, University of Oregon, Department of Computer Science, Eugene, OR 97403

166. Neville Moray, Department of Mechanical and Industrial Engineering, University of Illinois, 1206 West Green Street, Urbana, IL 61801

167. Charles Mosher, ARCO Exploration and Production Technology, 2300 West Plano Parkway, Plano, TX 75075-8499

168. Harish Nag, Intel Corporation, M/S CO4-02, 5200 Elam Young Parkway, Hillsboro, OR 97124

169. Jonathan Nash, Leeds University, School of Computer Studies, Leeds LS2 9JT, United Kingdom

170. Dan Nessett, Lawrence Livermore National Laboratory, L-60, Livermore, CA 94550

171. Lionel M. Ni, Michigan State University, Dept. of Computer Science, A714 Wells Hall, East Lansing, MI 48824-1027

172. Mike Norman, Edinburgh Parallel Computing Centre, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom

173. James M. Ortega, Department of Applied Mathematics, Thornton Hall, University of Virginia, Charlottesville, VA 22901

174. Steve Otto, Oregon Graduate Institute, Department of Computer Sci. and Eng., 19600 NW von Neumann Drive, Beaverton, OR 97006-1999

175. Andrea Overman, NASA Langley Research Center, MS 125, Hampton, VA 23665

176. Peter S. Pacheco, University of San Francisco, Department of Mathematics, San Francisco, CA 94117

177. Cherri M. Pancake, Department of Computer Science, Oregon State University, Corvallis, OR 97331-3202

178. Raj Panda, IBM Corporation, Mail Code E39/4305, 11400 Burnet Rd. , Austin, TX 78758

179. David Payne, Intel Corporation, Supercomputer Systems Division, 15201 NW Greenbrier Parkway, Beaverton, OR 97006

180. Arnulfo Perez, Centro de Intelligencia Artifical, ITESM, Suc. De Correos "J" C.P. 64849, Monterrey N.L., Mexico

181. K. S. Perienayagam, Centre for Development of Advanced Computing, Pune University Campus, Pune 411 007, India

182. Matthew Peters, Parallel and Distributed Processing, IBM UK Scientific Centre, Winchester, United Kingdom

183. Garry Petrie, Intel MS CO5-01, 5200 NE Elam Young Parkway, Hillsboro, OR 97124-6497

184. Greg Pfister, IBM Corporation, Mail Stop 9462, 11400 Burnet Road, Austin, TX 78758-3493

185. Jean-Laurent Philippe, ARCHIPEL S.A., PAE des Glaisins, 1 rue du Bulloz, F-74940 Annecy-le-Vieux, France

186. Paul Pierce, Intel Corporation, Supercomputer Systems Division, 15201 NW Greenbrier Parkway, Beaverton, OR 97006

187. Robert J. Plemmons, Departments of Mathematics and Computer Science, Box 7311, Wake Forest University Winston-Salem, NC 27109

188. James C. T. Pool, Deputy Director, Caltech Concurrent Supercomputing Facility, MS 158-79, California Institute of Technology, Pasadena, CA 91125

189. Steve Poole, 11631 Lima, Houston, TX 77099

190. Roldan Pozo, University of Tennessee, 107 Ayres Hall, Department of Computer Science, Knoxville, TN 37996-1301

191. Angela Quealy, Sverdrup Technology, Inc., NASA Lewis Research Center Group, 2001 Aerospace Pkwy, Brook Park, OH 44142

192. Padma Raghavan, University of Illinois, NCSA, 4151 Beckman Institute, 405 North Matthews Avenue, Urbana, IL 61801

193. Sanjay Ranka, Syracuse University, Northeast Parallel Architectures Center, 111 College Place, Syracuse, NY 13244-4100

194. Robbert van Renesse, Dept. of Computer Science, 4118 Upson Hall, Cornell University, Ithaca, NY 14853

195. Werner C. Rheinboldt, Department of Mathematics and Statistics, University of Pittsburgh, Pittsburgh, PA 15260

196. Peter Rigsbee, Cray Research Incorporated, 655 Lone Oak Drive, Eagan MN 55121

197. Guy Robinson, European Centre for Medium Range Weather Forecasting, Reading RG3 9AX, Berkshire, United Kingdom

198. Matt Rosing, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665-5225

199. Joel Saltz, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665-5225

200. Ahmed H. Sameh, CSRD, University of Illinois 1308 West Main Street Urbana, IL 61801-2307

201. Erich Schikuta, CITI/CRPC Rice University, 6100 South Main, Houston, TX 77005

202. Rob Schreiber, RIACS, Mail Stop T045-1, NASA Ames Research Center, Moffett Field, CA 94022

203. David S. Scott, Intel Scientific Computers, 15201 N. W. Greenbrier Parkway, Beaverton, OR 97006

204. Eugen Schenfeld, NEC Research Institute, 4 Independence Way, Princeton, NJ 08540

205. Ricardo A. Schmutzer, Pontificia Universidad Catolica de Chile, Department of Computer Science, Las Torcazas 212, Las Condes - Santiago, Chile

206. Mark Sears, Division 1424, Sandia National Laboratories, P O Box 5800, Albuquerque, NM 87185-5800

207. Ambuj Singh, UC Santa Barbara, Department of Computer Science, Santa Barbara, CA 93106

208. Chuck Simmons, 500 Oracle Parkway, Box 659414, Redwood Shores, CA 94065

209. Anthony Skjellum, Mississippi State University, Department of Computer Science, Drawer CS, Mississippi State, MS 39762-5623

210. Steven G. Smith, Lawrence Livermore National Lab, L-419, P. O. Box 808, Livermore, CA 94550

211. Marc Snir, IBM T. J. Watson Research Center, PO Box 218, Room 28-226, Yorktown Heights, NY 10598

212. Karl Solchenbach, PALLAS GmbH, Hermuelheimer Strasse 10 D-5040 Bruehl Germany

213. Charles H. Still, Lawrence Livermore National Lab, L-416, P. O. Box 808, Livermore, CA 94550

214. Alain Stroessel, Institut Francais du Petrole, Parallel Processing Group, BP 311 - 92506 Rueil Malmaison, France

215. Vaidy Sunderam, Emory University, Dept. of Math and Computer Science, Atlanta, GA 30322

216. Mike Surridge, Univ. of Southampton Parallel Applications Centre, 2 Venture Road, Chilworth Research Centre, Southampton SO1 7NP, United Kingdom

217. Alan Sussman, University of Maryland, Computer Science Department, A. V. Williams Building, College Park, MD 20742

218. Paul N. Swartztrauber, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307

219. Clemens-August Thole, GMD-I1.T, Schloss Birlinghoven, D-5205 Sankt Augustin 1, Germany

220. Bob Tomlinson, Los Alamos National Laboratory, Group C-8, MS B-272, Los Alamos, NM 87545

221. Anne Trefethen, Engineering and Theory Center, Cornell University, Ithaca, NY 14853

222. Christian Tricot, ARCHIPEL S.A., PAE des Glaisins, 1 rue du Bulloz, F-74940 Annecy-le-Vieux, France

223. Anna Tsao, Supercomputing Research Center, 17100 Science Drive, Bowie, MD 20715-4300

224. Lew Tucker, Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214

225. Robert van de Geijn, University of Texas, Department of Computer Sciences , TAI 2.124, Austin, TX 78712

226. Robert G. Voigt, National Science Foundation, Room 417, 1800 G Street, N.W., Washington, DC 20550

227. Linton Ward, 11400 Burnet Rd, Austin, TX 78758

228. Dick Weaver, IBM M77/E365, 555 Bailey Ave, P. O. Box 49023, San Jose, CA 95161-9023

229. Tammy Welcome, Lawrence Livermore National Lab, Massively Parallel Computing Initiative, L-416, P. O. Box 808, Livermore, CA 94550

230. Jim West, IBM Corporation, MC 5600, 3700 Bay Area Blvd., Houston, TX 77058

231. Stephen R. Wheat, Dept. 1424, Sandia National Labs, Albuquerque, NM 87185-5800

232. Mary F. Wheeler, Rice University, Department of Mathematical Sciences, P. O. Box 1892, Houston, TX 77251

233. Andrew B. White, Computing Division, Los Alamos National Laboratory, P. O. Box 1663, MS-265, Los Alamos, NM 87545

234. Joel Williamson, Convex Computer Corporation, 3000 Waterview Parkway, Richardson, TX 75083-3851

235. Steve Zenith, Kuck and Associates, Inc., 1906 Fox Drive, Champaign, IL 61820-7334

236. Mohammad Zubair, NASA Langley Research Center, Mail Stop 132C, Hampton, VA 23665

237. Office of Assistant Manager for Energy Research and Development, U.S. Department of Energy, Oak Ridge Operations Office, P. O. Box 2001 Oak Ridge, TN 37831-8600

238–239. Office of Scientific & Technical Information, P. O. Box 62, Oak Ridge, TN 37831