MPI: A Message Passing Interface

The MPI Forum*

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 and networks of workstations. 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 [15], PVM [2], NX/2 [16], Vertex, [14], PARMACS [10, 11], and P4 [4, 13]. 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 [6, 7], Zipcode [17, 18], and the IBM External User Interface [8]. 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 [19]. 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. Following this a preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker [5]. This proposal was intended as a discussion document, and embodies the main features that were identified in the earlier workshop as being necessary in a message passing standard. A meeting of the MPI working group was held at Supercomputing '92, at which it was decided to place the standardization process on a more formal footing, and generally to follow the format 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 July 1993 was set. To achieve this goal the MPI working group has met every 6 weeks for two days, and is presenting 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 is open to all members of the high performance computing community.

This paper is being written at a time when MPI

[1] See the Acknowledgements section for a list of members of the MPI Forum.

is still in the process of being defined, but when the main features have been agreed upon. The only major exception is the rôle played by communicator objects in handling process groups and communication contexts. This is discussed in Section 3.1, and at the time of writing (August 1993) is still an area of active discussion. The details of the syntax, and the language bindings for Fortran-77, Fortran-90, C, 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 and workstation networks. 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 provides reliable communications (or fails the program). 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. (Though MPI does provide a portable mechanism which will allow its intrumentation and the collection of tracefiles for tools such as `ParaGraph`[9] or `Upshot`[12]). MPI does not provide support for active messages. MPI was designed easily to allow heterogeneous implementations and virtual communication channels. 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 for a multithreaded environment.

3 Details of MPI

In this section we discuss the MPI routines in more detail, and indicate some of the alternate suggestions that have been made for different aspects of the interface. 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 at the time of writing a consensus is only just beginning to emerge.

3.1 Groups, Contexts, and Communicators

This section explains the concepts of group and context, which are, in turn, bound together into abstract communicator objects.

3.1.1 Process Groups

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). The initial MPI specification 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. Since MPI says nothing about the way in which a program is started, it takes no stance on whether these processes are multiple instances of the same executable (the SPMD model), or instances of many executables (loose MIMD model), or something in between. However, the MPI draft will not preclude the subsequent addition or adoption of a more sophisticated, dynamic process model.

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. To make a group with different membership, a new group must be created. This operation can be performed either locally (without synchronisation), or by a collective partitioning operation in the group to be split. In MPI a group is an opaque object referenced by means of a handle[1]. 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.

3.1.2 Communication Contexts

Communication contexts were initially proposed to allow the creation of distinct, separable message streams between processes, with each stream having a unique context. A common use of contexts is to ensure that messages sent in one phase of an application are not incorrectly intercepted by another phase. The point here is that the two phases may actually be calls to two different third-party library routines, and the application developer has no way of knowing if the message tag, group, and rank completely disambiguate the message traffic of the different libraries from one another and from the rest of the application. Context provides an additional criterion for message selection, and hence permits the construction of independent message tag spaces (see Section 3.3.1).

The user never performs explicit operations on contexts (there is no user visible context data type), however contexts are maintained within communicators on the user's behalf, so that messages sent through a given communicator can only be received through the correctly matching communicator. MPI provides a collective routine on a communicator to pre-allocate a number of contexts for use within the scope of that communicator, these can then be used by the MPI system, without a further synchronisation, when the user creates duplicates or sub-groups using the communicator. The program is correct, provided that these operations occur in the same order on all the processes which own the communicator. (This is the same criterion as for the other collective operations on a communicator.)

[1] In Fortran, a handle is an index into a table, while in C, a handle will be a provided typedef.

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. In this case the source and destination processes are given by their ranks within their respective groups. In MPI abstract objects called "communicators" are used to define the scope of a communication operation. Communicators used in intra-group and inter-group communication are referred to as intra- and inter-communicators, respectively. An intra-communicator can be regarded as binding together a context and a group, while an inter-communicator binds together a context and two groups, one of which contains the source and the other the destination. Communicator objects are passed to all point-to-point and collective communication routines to specify the context and the group, or groups, involved in the communication operation.

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 are connected by an arc. As a convenience, MPI provides explicit support for n-dimensional Cartesian grids. 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.

3.3 Point-to-Point Communication

3.3.1 Message Selectivity

MPI provides for point-to-point communication, with message selectivity explicitly based on source process, message tag, and communication context. The source and tag may be wild-carded, so that in effect they are ignored in message selection. The context may not be wild-carded. The source and destination processes are specified by means of a group and a rank. For intra-group communication the group and context are bound together in an intra-communicator, as discussed in Section 3.1.3. For inter-group communication the groups containing the source and destination processes are bound together with the context in an inter-communicator. Thus, a send routine is passed a handle to a communicator object, the rank of the destination process, and the message type to fully specify the context and destination of a message. A receive routine uses the same three things to determine message selectivity.

3.3.2 Communication Modes

A send operation can take place in one of three communication modes. A message sent in **standard** mode does not require a corresponding receive to have been previously posted on the destination process. A message sent in standard mode will still be delivered when the receive is posted sometime later. A message sent in **ready** mode requires that a receive have been previously posted on the destination process. If the receive has not been previously posted the outcome is indeterminate. In standard mode, the send can return before the matching receive has been posted. For a message sent in **synchronous** mode the send operation does not return until a matching receive has been posted on the destination process.

For each of the three communication modes, a send operation can either be locally blocking or nonblocking, so there are a total of six different types of send routine. A blocking send routine will not return until the data locations specified in the send can be safely reused without corrupting the message. A nonblocking send does not wait for any particular event to occur before returning. Instead it returns a handle to a communication object that can subsequently be used when calling routines that check for completion of the send operation.

A receive operation may also be locally blocking or nonblocking and either of these two types of receive may be used to match any of the six types of send. A blocking receive will not return until the message has been stored at the locations indicated by the receive. A nonblocking receive returns a handle to a communication object, and does not wait for any particular event to occur. The handle can be used subsequently to check the status of the receive operation, or to block until it completes. A nonblocking receive also returns a handle to a "return status object" which is used to store the length, source, and tag of the message. When the receive has completed this information can then be queried by calling an appropriate routine.

The 6 send and 2 receive routines described above form the core of the MPI standard for point-to-point communication.

3.3.3 User-defined Datatypes

MPI provides mechanisms to specify general, mixed (of different types), non-contiguous message buffers. This is done by allowing the user to define the datatype (which consists of a set of types and memory offsets) using MPI datatype-definition routines. Once the datatype is defined, it can be passed into any of the point-to-point or collective communication routines. The effect of this will be for data to be collected out of possibly non-contiguous memory locations, transmitted, and then placed into possibly non-contiguous memory locations at the receiving end. It is up to the implementation to decide whether the data of a general datatype should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

User-defined datatypes as supported by MPI allow the convenient and (potentially) efficient transmittal of general array sections (in Fortran 90 terminology), and arrays of (sub-portions of) records or structures.

since all send and receive routines specify numbers of data items of a particular type, whether built-in or user-defined, implementations have enough information to provide transaltions that allow an MPI program to run on heterogeneous networks.

3.4 Collective Communication

Collective communication routines provide for coordinated communication among a group of processes [1, 3]. The process group and context is given by the intra-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. In addition, the collective communication routines may be , but do not have to be, implemented using the MPI point-to-point routines. Collective communication routines do not have a tag argument. A collective communication routine must be called by all members of the group with consistent arguments. As soon as a process has completed its rôle in the collective communication it may continue with other tasks. Thus, a collective communication is not necessarily a barrier synchronization for the group. On the other hand, an MPI implementation is free to have barriers inside collective communication functions. In short, the user must program as if the collective communication routines do have barriers, but cannot depend on any synchronization from them. MPI does not include nonblocking forms of the collective communication routines. In 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 three basic types of collective data movement routine: broadcast, scatter, and gather. There are two versions of each of these. 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. 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. All processes must send and/or receive buffers of the same type and length.

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, each process ends up with the same output buffer, which is the concatenation of the input buffers 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. The root process receives the concatenation of the input buffers of all processes, in rank order. The all-all gather routine is identical to the all-all scatter routine.

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, and in the second version the user supplies (a pointer to) a function. Thus, MPI contains four reduce and four scan routines.

4 Summary and Conclusions

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`.

Acknowledgements

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 the result of hours of deliberation with members of the MPI Forum, which consists of: Scott Berryman, Lyndon Clarke, Doreen Cheng, James Cownie, Jack Dongarra, Anne C. Elster, Jim Feeney, Sam Fineberg, Jon Flower, Al Geist, Ian Glendinning, Adam Greenberg, William Gropp, Leslie Hart, Tom Haupt, Don Heller, Tom Henderson, Rolf Hempel, Tony Hey, C. T. Howard Ho, Steve Huss-Lederman, John Kapenga, Bob Knighten, Rik Littlefield, Ewing Lusk, Arthur B. Maccabe, Peter Madams, Oliver McBryan, Dan Nessett, Steve Otto, Peter Pacheco, Paul Pierce, Sanjay Ranka, Peter Rigsbee, Mark Sears, Ambuj Singh, Anthony Skjellum, Marc Snir, Alan Sussman, Eric Van de Velde, David Walker, Stephen Wheat, and Steven Zenith.

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] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. A users' guide

to PVM parallel virtual machine. Technical Report TM-11826, Oak Ridge National Laboratory, July 1991.

[3] 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.

[4] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

[5] 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.

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

[9] Heath, M. T. and J. A. Etheridge. 1991. "Visualizing the performance of parallel programs." Technical Report ORNL TM-11813. Oak Ridge National Laboratory.

[10] 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.

[11] 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.

[12] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with `upshot`. Technical Report ANL–91/15, Argonne National Laboratory, 1991.

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

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

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

[16] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[17] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990.

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

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