Programming Environments for Massively Parallel Distributed Systems, Monte Verità, Switzerland

The MPI Message Passing Interface Standard

Lyndon Clarke*
Edinburgh Parallel Computing Centre,
The University of Edinburgh,
Edinburgh EH9 3JZ, UK
lyndon@epcc.ed.ac.uk

Ian Glendinning†
Dep. of Elec. and Comp. Science,
University of Southampton,
Southampton, SO9 5NH, UK
igl@ecs.soton.ac.uk

Rolf Hempel†
GMD (German National Research Centre for Computer Science),
53731 Sankt Augustin, Germany
Rolf.Hempel@gmd.de

Abstract

The diverse message passing interfaces provided on parallel and distributed computing systems have caused difficulty in movement of application software from one system to another and have inhibited the commercial development of tools and libraries for these systems. The Message Passing Interface (MPI) Forum has developed a *de facto* interface standard which was finalised in Q1 of 1994. Major parallel system vendors and software developers were involved in the definition process, and the first implementations of MPI are already appearing. This article presents an overview of the MPI initiative and the standard interface, in particular those aspects which merge demonstrated research with common practice.

1 Introduction

The message passing paradigm is the most generally applicable and efficient programming model for parallel machines with distributed memory and has been used widely in parallel and distributed computing systems for some years. The development of parallel computing has been hindered by the absence of a standard message passing interface.

During 1992 the international Message Passing Interface (MPI) initiative was founded by Oak Ridge National Laboraory and the Center for Parallel Computing

*Work supported by the Science and Engineering Research Council NACC grant B/28667
†Work supported in part by CEC ESPRIT project 6643 (PPPE)

at Rice University [8]. The goal of this effort was to define a message passing interface wihch would be efficiently implemented on a wide range of parallel and distributed computing systems, this establishing a *de facto* standard and avoiding the overhead and delays associated with an official standardization process.

The procedures of the MPI Forum were modelled on those of the HPF Forum, in particular the process was open to all interested parties. Much of the technical discussion was conducted via electronic mail, and the forum met every six weeks in Dallas where formal decisions were made. The first complete draft [3] was presented in November 1993 at the Supercomputing Conference '93 in Portland, Oregon. Two further meetings were held early in 1994, the first at INRIA Sophia Antipolis, France and the second at Knoxville, Tennesee. These meetings considered public comment on the November draft which resulted in final changes and approval of the standard by the Forum. The finished interface document is planned to be released in April 1994.

2 Overview of MPI

MPI is intended to be the standard message passing interface for parallel application and library programming. The basic content of MPI is point–to–point communication between pairs of processes and collective communication within groups of processes. MPI also contains more advanced message passing features which allow the user to manipulate process groups, provide topological structure for process groups, and support the development and utilisation of parallel libraries.

The computing platforms for MPI comprise homogeneous and heterogeneous parallel and distributed systems. Every message whether in point–to–point or collective communication has an associated data type. The primitive data types of the host language, for example `INTEGER` and `REAL` in Fortran, are supported. MPI also provides very general facilities which can be used to describe `struct` types in C, and non-contiguous data in either C or Fortran, as "derived" data types. The data types of MPI provide all the information required for data conversion in heterogeneous environments, and do not preclude efficient implementation of MPI in homogeneous environments. They also allow the user to send and receive messages with complicated storage patterns without the need to copy data in to and out of message buffers, and allow an implementation to optimise communications with such storage patterns.

MPI does not make provision for process creation aside from requiring at least a basic SPMD process model from implementations. Other important issues such as parallel input/output and remote read/write were not included in MPI because the committee felt that research into these features is not yet mature enough for standardisation, or that there was insufficient time to establish concensus during this phase of MPI. The MPI Forum intends to cover further topics in a second phase which may commence as early as the second half of 1994.

3 Groups, contexts and communicators

Point–to–point and collective communications within MPI are performed within process groups. MPI defines a group as an ordered set of process identifiers, each of which is assigned a numerical rank within the group, between zero and the size of the group. Communications within MPI are also performed within a communication context which insulates messages in different parts of the program from one another. The defining property of a context is that a message sent in one context can only be received in that same context. The communication context is the primary mechanism for isolation of messages in different libraries and the user program from one another.

Process groups are user level objects in MPI but communication contexts are not directly visible. MPI bundles the process group and communication context concepts into a user level object called a *communicator* which provides communication services within a unique scope, as in the Zipcode [7] and CHIMP [5, 1] interfaces. MPI defines an initial communicator MPI_COMM_WORLD which has a group containing all processes of the program and a unique communication context.

MPI provides routines which allow the user to dynamically create new communicators, similar to the group routines of EUI [4]. MPI_COMM_DUP creates a duplicate of an existing communicator, i.e. a new communicator with the same group of processes and a different communication context. This routine is key to the construction of robust communicative parallel libraries. MPI_COMM_SPLIT creates one or more new communicators which contain distinct subgroups of an existing communicator and of course a different context. This routine is key to clear expression of task and control parallel programs. In the simplest use of MPI, which corresponds to a number of current communication libraries, MPI_COMM_WORLD is the only communicator used in the program.

4 Point-to-point communication

The point-to-point message-passing routines form the core of the MPI standard, the basic operations being *send* and *receive*. They allow messages to be sent between pairs of processes, with message selectivity based explicitly on message tag and source process, and implicitly on communication context. Each process can execute its own code, in MIMD style, and can be sequential or multithreaded. There is no explicit support for threads, but care has been taken to make MPI "thread safe", by avoiding the use of global state.

The send and receive primitives are provided in a blocking form in which the sender buffer can be reused immediately on return from send and the receiver buffer contains the complete message on return from receive. There is one blocking receive primitive MPI_RECV. There are four blocking send primitives corrspeonding to the four communication modes in MPI.

Standard The sender is blocked until the send buffer can be reused without altering the message. The receiver is blocked until the message has been copied into the receive buffer. Since the system is expected to copy the message subject to buffer resources, the send-recv pair does not guarantee synchronisation. This mode seems to best represent common practice. The send primitive is MPI_SEND.

Synchronous The sender is blocked until the receiver issues the corresponding receive. No system buffer is required and the message can be transferred without intermediate copies, at the expense of synchronisation. The send primitive is MPI_SSEND.

Ready-receive The program is in error if the send is issued before the matching receive has been issued. This allows a simple protocol where the message is sent "in hope" and dropped if there is no ready receive, but use demands special care. The send primitive is MPI_RSEND.

Buffered This mode allows the user to control the space available for buffering within a defined buffer model, providing guaranteed portability for programs that demand message buffering. The send primitive blocks until the message is copied into the buffer space or is in error if insufficient buffer space was available. The send primitive is MPI_BSEND.

MPI also provides primitives of the non-blocking, or immediate return, form, MPI_I?SEND and MPI_IRECV, in which the message buffer must not be used until the communication has completed, similar to the immediate routines in NX/2 [6]. There is a small but comprehensive set of routines to test and wait for completion of non-blocking functions. This functionality, which is semantically orthogonal to the four communication modes, allows the system to overlap communication with computation and allows the user to write programs which do not incur the overhead of copying message data into intermediate buffers.

5 Collective communication

Collective communications are provided where all processes in a process group are involved in a collective operation. A collective function is called as if it contained a group synchronisation, although this property is not mandated since efficient implementations may not synchronise. We now decribe a familiar selection of the collective routines.

MPI_BARRIER Synchronisation of every processes within a group.

MPI_BCAST Every process within a group receives data broadcast by a "root" process.

MPI_GATHER Every process within a group sends data to a "root" which stores the data in rank order.

MPI_SCATTER The inverse of MPI_GATHER, where a "root" process sends sections of data to every process within a group in rank order.

MPI_REDUCE Performs a parallel reduction over every group process within a group. The operation is selected from a set of defined arithmetic and logical operators or is described as a user function. The output is available to a "root" process, every process, or scattered over the processes.

MPI also contains collective routines for all-to-all global communication, all-to-all personal communication otherwise known as complete exchange, and inclusive parallel prefix otherwise known as scan.

6 Process topologies

Many numerical applications have a geometrical background. For example, the parallelization of a PDE solver on a three–dimensional grid leads to a corresponding arrangement of the processes. The most natural way of addressing those processes is to specify their coordinates in the grid, as opposed to their linear ranks in the group. MPI supports the setup of general Cartesian process structures, as well as arbitrary process graphs, similar to the PARMACS interface [2], Process topologies in MPI are assigned to process groups within communicators, and process ranks in the group are ordered by topological location. Topologies are created and deleted at run-time, and a process can exist within many topologies simultaneously.

Cartesian topologies are created by calling MPI_CART and can also be derived from higher dimension Cartesian topologies by calling MPI_CART_SUB. For example these functions can be used to create a two dimensional process grid group and the corresponding one dimensional process row and column groups. Since group boundaries limit the scope of collective operations the process topologies can easily be used for operations like broadcast and reduction in matrix columns.

Graph topologies are created by calling MPI_GRAPH which accepts an adjacency list as the description of the graph. This complements the Cartesian topology and is applicable in problems which are parallelised by the block structured domain decomposition approach where adjacent blocks are mapped to adjacent processors in the graph. Due to the locality principle of PDE methods, a process in a topology tends to exchange most messages with adjacent processes. Thus, the topology information can be used by an MPI implementation to minimize network congestion by mapping adjacent processes onto adjacent resources.

7 Conclusion

We have briefly described features of the MPI standard including the core point to point and collective communications, communication contexts and process groups, and process topologies. We have not described more advanced features such as the communicator cache facility, which allows the user to extend the collective communication and process topology capabilities of MPI, or provision for communication between processes in different groups, which makes MPI attractive for applications which contain internal parallel client-server or pipeline structures. The interested reader is referred to the interface document [3].

At the time of writing two implementations of MPI are known to be available in the public domain. The first of these has been authored by Argonne National Laboratory and Mississipi State University, and is based on a device interface which has been designed to allow rapid and reasonably efficient ports of MPI to parallel systems. The device interface has been implemented using Chameleon providing a range of platforms, and has also been ported directly to a small number of parallel systems. The second implementation has been authored by Edinburgh Parallel Computing Centre as a library running atop CHIMP, which also provides a range of parallel and distributed computing systems.

References

[1] R. Alasdair A. Bruce, James G. Mills, and A. Gordon Smith. Chimp version 2.0 interface. Technical Report EPCC-KTP-CHIMP-V2-IFACE, Edinburgh Parallel Computing Centre, University of Edinburgh, January 1993.

[2] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message–passing library. *Parallel Computing, special issue on message–passing interfaces*, to appear.

[3] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993.

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

[5] James G. Mills, Lyndon J. Clarke, and Arthur S. Trew. Chimp concepts. Technical Report EPCC-KTP-CHIMP-CONC, Edinburgh Parallel Computing Centre, University of Edinburgh, April 1991.

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

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

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