A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard

William Gropp
Ewing Lusk
Mathematics and Computer Science Division*
Argonne National Laboratory

Nathan Doss
Anthony Skjellum
Department of Computer Science &
NSF Engineering Research Center for CFS
Mississippi State University†

Abstract

MPI (Message Passing Interface) is a specification for a standard library for message passing that was defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and applications specialists. Multiple implementations of MPI have been developed. In this paper, we describe MPICH, unique among existing implementations in its design goal of combining portability with high performance. We document its portability and performance and describe the architecture by which these features are simultaneously achieved. We also discuss the set of tools that accompany the free distribution of MPICH, which constitute the beginnings of a portable parallel programming environment. A project of this scope inevitably imparts lessons about parallel computing, the specification being followed, the current hardware and software environment for parallel computing, and project management; we describe those we have learned. Finally, we discuss future developments for MPICH, including those necessary to accommodate extensions to the MPI Standard now being contemplated by the MPI Forum.

1 Introduction

The message-passing model of parallel computation has emerged as an expressive, efficient, and well-understood paradigm for parallel programming. Until recently, the syntax and precise semantics of each message-passing library implementation were different from

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†This work was supported in part by the NSF Engineering Research Center for Computational Field Simulation, Mississippi State University. Additional support came from the NSF Career Program under grant ASC-95-01917, and from ARPA under Order D350.

the others, although many of the general semantics were similar. The proliferation of message-passing library designs from both vendors and users was appropriate for a while, but eventually it was seen that enough consensus on requirements and general semantics for message-passing had been reached that an attempt at standardization might usefully be undertaken.

The process of creating a standard to enable portability of message-passing applications codes began at a workshop on Message Passing Standardization in April 1992, and the Message Passing Interface (MPI) Forum organized itself at the Supercomputing '92 Conference. During the next eighteen months the MPI Forum met regularly, and Version 1.0 of the MPI Standard was completed in May 1994 [16, 36]. Some clarifications and refinements were made in the spring of 1995, and Version 1.1 of the MPI Standard is now available [17]. For a detailed presentation of the Standard itself, see [42]; for a tutorial approach to MPI, see [29]. In this paper we assume that the reader is relatively familiar with the MPI specification, but we provide a brief overview in Section 2.2.

The project to provide a portable implementation of MPI began at the same time as the MPI definition process itself. The idea was to provide early feedback on decisions being made by the MPI Forum and provide an early implementation to allow users to experiment with the definitions even as they were being developed. Targets for the implementation were to include all systems capable of supporting the message-passing model. MPICH is a freely available, complete implementation of the MPI specification, designed to be both portable and efficient. The "CH" in MPICH stands for "Chameleon," symbol of adaptability to one's environment and thus of portability. Chameleons are fast, and from the beginning a secondary goal was to give up as little efficiency as possible for the portability.

MPICH is thus both a research project and a software development project. As a *research project*, its goal is to explore methods for narrowing the gap between the programmer of a parallel computer and the performance deliverable by its hardware. In MPICH, we adopt the constraint that the programming interface will be MPI, reject constraints on the architecture of the target machine, and retain high performance (measured in terms of bandwidth and latency for message-passing operations) as a goal. As a *software project*, MPICH's goal is to promote the adoption of the MPI Standard by providing users with a free, high-performance implementation on a diversity of platforms, while aiding vendors in providing their own customized implementations. The extent to which these goals have been achieved is the main thrust of this paper.

The rest of this paper is organized as follows. Section 2 gives a short overview of MPI and briefly describes the precursor systems that influenced MPICH and enabled it to come into existence so quickly. In Section 3 we document the extent of MPICH's portability and present results of a number of performance measurements. In Section 4 we describe in some detail the software architecture of MPICH, which comprises the results of our research in combining portability and performance. In Section 5 we present several specific aspects of the implementation that merit more detailed analysis. Section 6 describes a family of supporting programs that surround the core MPI implementation and turn MPICH into a portable environment for developing parallel applications. In Section 7 we describe how we as a small, distributed group have combined a number of freely available tools in the Unix environment to enable us to develop, distribute, and maintain MPICH with a minimum of resources. In the course of developing MPICH, we have learned a number of lessons from the challenges posed (both accidentally and deliberately) for MPI implementors by the MPI specification; these lessons are discussed in Section 8. Finally, Section 9 describes the current status of MPICH (Version 1.0.12 as of February 1996) and outlines our plans for future development.

2 Background

In this section we give an overview of MPI itself, describe briefly the systems on which the first versions of MPICH were built, and review the history of the development of the project.

2.1 Precursor Systems

MPICH came into being quickly because it could build on stable code from existing systems. These systems prefigured in various ways the portability, performance, and some of the other features of MPICH. Although most of that original code has been extensively reworked, MPICH still owes some of its design to those earlier systems, which we briefly describe here.

P4 [8] is a third-generation parallel programming library, including both message-passing and shared-memory components, portable to a great many parallel computing environments, including heterogeneous networks. Although p4 contributed much of the code for TCP/IP networks and shared-memory multiprocessors for the early versions of MPICH, most of that has been rewritten. P4 remains one of the "devices" on which MPICH can be built (see Section 4), but in most cases more customized alternatives are available.

Chameleon [31] is a high-performance portability package for message passing on parallel supercomputers. It is implemented as a thin layer (mostly C macros) over vendor message-passing systems (Intel's NX, TMC's CMMD, IBM's MPL) for performance and over publicly available systems (p4 and PVM) for portability. A substantial amount of Chameleon technology is incorporated into MPICH(as detailed in Section 4).

Zipcode [41] is a portable system for writing scalable libraries. It contributed several concepts to the design of the MPI Standard—in particular contexts, groups, and mailers (the equivalent of MPI communicators). Zipcode also contains extensive collective operations with group scope as well as virtual topologies, and this code was heavily borrowed from in the first version of MPICH.

2.2 Brief Overview of MPI

MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation (such as a message buffering and message delivery progress requirement). MPI includes point-to-point message passing and collective (global) operations, all scoped to a user-specified group of processes. Furthermore, MPI provides *abstractions* for processes at two levels. First, processes are named according to the rank of the group in which the communication is being performed. Second, virtual topologies allow for graph or Cartesian naming of processes that help relate the application semantics to the message passing semantics in a convenient, efficient way. Communicators, which house groups and communication context (scoping) information, provide an important measure of safety that is necessary and useful for building up library-oriented parallel code.

MPI also provides three additional classes of services: environmental inquiry, basic timing information for application performance measurement, and a profiling interface for external performance monitoring. MPI makes heterogeneous data conversion a transparent part of its services by requiring datatype specification for all communication operations. Both built-in and user-defined datatypes are provided.

MPI accomplishes its functionality with opaque objects, with well-defined constructors and destructors, giving MPI an object-based look and feel. Opaque objects include groups (the fundamental container for processes), communicators (which contain groups and are used as arguments to communication calls), and request objects for asynchronous operations. User-defined and predefined datatypes allow for heterogeneous communication and elegant description of gather/scatter semantics in send/receive operations as well as in collective operations.

MPI provides support for both the SPMD and MPMD modes of parallel programming. Furthermore, MPI can support interapplication computations through intercommunicator operations, which support communication between groups rather than within a single group. Dataflow-style computations also can be constructed from intercommunicators. MPI provides a thread-safe application programming interface (API), which will be useful in multithreaded environments as implementations mature and support thread safety themselves.

2.3 Development History of MPICH

At the organizational meeting of the MPI Forum at the Supercomputing '92 conference, Gropp and Lusk volunteered to develop an immediate implementation that would track the Standard definition as it evolved. The purpose was to quickly expose problems that the specification might pose for implementors and to provide early experimenters with an opportunity to try ideas being proposed for MPI before they became fixed. The first version of MPICH, in fact, implemented the prespecification described in [46] within a few days. The speed with which this version was completed was due to the existing portable systems p4 and Chameleon. This first MPICH, which offered quite reasonable performance and portability, is described in [30].

Starting in spring 1993, this implementation was gradually modified to provide increased performance and portability. At the same time the system was greatly expanded to include all of the MPI specification. Algorithms for the collective operations and topologies, together with code for attribute management, were borrowed from Zipcode and tuned as the months went by.

What made this project unique was that we had committed to following the MPI specification as it developed—and it changed at every MPI Forum meeting. Most system implementors wait for a stable specification. The goals of this project dictated that, in the short term, we deliberately choose a constantly changing specification. The payoff came, of course, when the MPI Standard was released in May 1994: the MPICH implementation was complete, portable, fast, and available immediately. It is worthwhile to contrast this situation with what happened in the case of the High-Performance Fortran (HPF) Standard. The HPF Forum (which started and finished a year before the MPI Forum) produced their standard specification in much the same way that the MPI Forum did. However, since implementation was left entirely to the vendors, who naturally waited until the specification was complete before beginning to invest implementation effort, HPF implementations are only now (February 1996) becoming available, whereas a large community has been using MPI for over a year.

For the past year, with the MPI Standard stable, MPICH has continued to evolve in several directions. First, the Abstract Device Interface (ADI) architecture, described in Section 4 and central to the performance, has developed and stabilized. Second, individual vendors and others have begun taking advantage of this interface to develop their own highly specialized implementations of it; as a result, extremely efficient implementations of MPI exist on a greater variety of machines than we would have been able to tune MPICH for ourselves. In particular, Convex, Intel, SGI, and Meiko have produced implementations of the ADI that produce excellent performance on their own machines, while taking advantage of the portability of the great majority of the code in MPICH above the ADI layer. Third, the set of tools that form part of the MPICH parallel programming environment has been extended; these are described in Section 6.

2.4 Related Work

The publication of the MPI Standard provided many implementation groups with a clear specification; and several freely available, partially portable implementations have appeared. Like MPICH, their initial versions were built on existing portable message-passing systems. They differ from MPICH in that they focus on the workstation environment, where software performance is necessarily limited by Unix socket functionality. Some of these systems are as follows:

- LAM [7] is available from the Ohio Supercomputer Center and runs on heterogeneous networks of Sun, DEC, SGI, IBM, and HP workstations.
- CHIMP-MPI [5] is available from the Edinburgh Parallel Computing Center and runs on Sun, SGI, DEC, IBM, and HP workstations, the Meiko Computing Surface machines, and the Fujitsu AP-1000. It is based on CHIMP [9].
- At the Technical University of Munich, research has been done on a system for checkpointing message-passing jobs, including MPI. See [43] and [44].
- Unify [45], available from Mississippi State University, layers MPI on a version of PVM [20] that has been modified to support contexts and static groups. Unify allows mixed MPI and PVM calls in the same program.

Proprietary and platform-specific implementations provided by vendors are described in Section 9.

3 Portability and Performance

The challenge of the MPICH project is to combine both portability and performance. In this section we first survey the range of environments in which MPICH can be used, and then present performance data for a representative sample of those environments.

3.1 Portability of MPICH

The MPI standard itself addresses the message-passing model of parallel computation. In this model, processes with separate address spaces (like Unix processes) communicate with one another by sending and receiving messages. A number of different hardware platforms support such a model.

3.1.1 Exploiting High-Performance Switches

The most obvious hardware platform for MPI is a distributed-memory parallel supercomputer, in which each process can be run on a separate node of the machine, and communication occurs over a high-performance switch of some kind. In this category are the Intel Paragon, IBM SP2, Meiko CS-2, Thinking Machines CM-5, NCube-2, and Cray T3D. (Although the Cray T3D provides some hardware that allows one to treat it as a shared-memory machine, it falls primarily into this category; see [4].) Details of how MPICH is implemented on each of these machines are given in Section 4, and performance results for the Paragon and SP2 are given in Section 3.2.

3.1.2 Exploiting Shared-Memory Architectures

A number of architectures support a shared-memory programming model, in which a memory location can be both read and written to by multiple processes. Although this is not part of MPI's computational model, an MPI implementation may take advantage of capabilities in this area offered by the hardware/software combination to provide particularly efficient message-passing operations. Current machines offering this model include the SGI Onyx, Challenge, Power Challenge, and Power Challenge Array machines, IBM SMP's (symmetric multiprocessors), the Convex Exemplar, and the Sequent Symmetry. MPICH is implemented using shared memory for efficiency on all of these machines (details in Section 4). Performance measurements for the SGI are given in Section 3.2.6.

3.1.3 Exploiting Networks of Workstations

One of the most common parallel computing environments is a network of workstations. Many institutions use Ethernet-connected personal workstations as a "free" computational resource, and at many universities laboratories equipped with Unix workstations provide both shared Unix services for students and an inexpensive parallel computing environment for instruction. In many cases, the workstation collection includes machines from multiple vendors. Interoperability is provided by the TCP/IP standard. MPICH runs on workstations from Sun (both SunOS and Solaris), DEC, Hewlett-Packard, SGI, and IBM. Recently, the Intel 486 and Pentium compatible machines have been able to join the Unix workstation family by running one of the common free implementations of Unix, such as FreeBSD, NetBSD, or Linux. MPICH runs on all of these workstations and on heterogeneous collections of them. Details of how heterogeneity is handled are presented in Section 4, and some performance figures for Ethernet-connected workstations are given in Section 3.2.

An important family of non-Unix operating systems is supported by Microsoft. MPICH has been ported to Windows 3.1 (where it simulates multiprocessing on a single processor); the system is called WinMPI [37, 38].

3.2 Performance of MPICH

The MPI specification was designed to allow high performance in the sense that semantic restrictions on optimization were avoided wherever user convenience would not be severely impacted. Furthermore, a number of features were added to enable users to take advantage of optimizations that some systems offered, without affecting portability to other systems that did not have such optimizations available. In MPICH we have tried to take advantage of those features in the Standard that allow for extra optimization, but we have not done so in every possible case.

Performance on one's own application is, of course, what counts most. Nonetheless, useful predictions of application performance can be made, based on the results of specially constructed benchmark programs. In this section, we first describe some of the difficulties that arise in benchmarking message-passing systems, then discuss the programs we have developed to address these difficulties and finally present results from running the benchmarks on a representative sample of the environments supported by MPICH.

The MPICH implementation includes two MPI programs, `mpptest` and `goptest`, that provide reliable tests of the performance of an MPI implementation. The program `mpptest` provides testing of both point-to-point and collective operations on a specified number of processors; the program `goptest` can be used to study the scalability of collective routines as a function of number of processors.

3.2.1 Performance Measurement Problems and Pitfalls

One common problem with simple performance measurement programs is that the results are different each time the program is run, even on the same system. A number of factors are responsible, ranging from assuming that the clock calls have no cost and infinite resolution to the effects of other jobs running on the same machine. A good performance test will give the same (to the clock's precision) answer each time. The `mpptest` and `goptest` programs distributed with MPICH compute the average time for a number of iterations of an operation (thus handling the cost and granularity of the clock) and then run the same test over several times and take the minimum of those times (thus reducing the effects of other jobs). The programs can also provide information about the mean and worst-case performance.

More subtle are issues of which test to run. The simplest "ping-pong" test, which sends the same data (using the same data buffer) between two processes, allows data to reside entirely in the memory cache. In many real applications, however, neither buffer will already be mapped into cache, and this situation can affect the performance of the operation. Similarly, data transfers that are not properly aligned on word boundaries can be more expensive than those that are. MPI also has noncontiguous datatypes; the performance of an implementation with these datatypes may be significantly slower than for contiguous data. Another parameter is the number of processors used, even if only two are communicating. Certain implementations will include a latency cost proportional to the number of processors. This gives the best performance on the two-processor ping-pong test at the cost of (possibly) lower performance on real applications. `Mpptest` and `goptest` include tests to measure these effects.

3.2.2 Benchmarks for Point-to-Point Operations

In this section we present some of the simplest benchmarks for performance of MPICH on various platforms. The performance test programs `mpptest` and `goptest` can produce a wealth of information; the script `basetest`, provided with the MPICH implementation, can be used to get a more complete picture of the behavior of a particular system. Here, we present only the most basic data: short- and long-message performance.

For the short-message graphs, the only options used with `mpptest` are `-auto` and `-size 0 1000 40`. The option `-auto` tells `mpptest` to choose the sizes of the messages so as to reveal the exact message size where there is any sudden change in behavior (for example, at an internal packet-size boundary). The `-size` option selects messages with sizes from 0 to 1000 bytes in increments of 40 bytes. The short-message graphs give a good picture of the latency of message passing.

For the long-message graphs, a few more options are used. Some make the test runs more efficient. The size range of message is set with `-size 1000 77000 4000`, which selects messages of sizes between about 1K and 80K, sampled every 4000 bytes.

These tests provide a picture of the best achievable bandwidth performance. More realistic tests can be performed by using `-cachesize` (to force the use of different data areas), `-overlap` (for communication and computation overlap), `-async` (for nonblocking communications), and `-vector` (for noncontiguous communication). Using `-givedy` gives information on the range of performance, displaying both the mean and worst-case performance.

3.2.3 Performance of MPICH Compared with Native Vendor Systems

One question that can be asked about MPI is how its performance compares with proprietary vendor systems. Fortunately, the `mpptest` program was designed to work with many message-passing systems and can be built to call a vendor's system directly. In Figure 1, we compare MPI and Intel's NX message-passing. The MPICH implementation for the Intel Paragon, while implemented with a special ADI, still relies on message-passing services provided by NX. Despite this fact, the MPI performance is quite good and can probably be improved with the second-generation ADI, planned for a later release of MPICH. We use this as a representative example to demonstrate that the apparently elaborate structure shown in Figures 7 and 8 does not impose serious performance overheads beyond those of

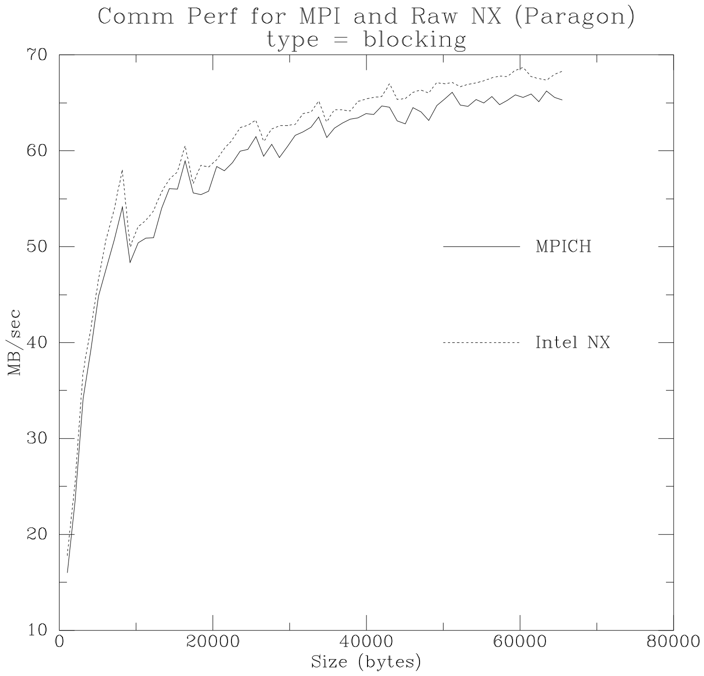

Figure 1: MPICH vs. NX on the Paragon

the underlying, vendor-specific message-passing layer.

3.2.4 Paragon Measurements

The Intel Paragon has a classic distributed-memory architecture with a (cut-through routed) 2-D mesh topology. Latency and bandwidth performance are shown in Figure 2. The Paragon performance measurements shown in Figure 2 were taken while other users were on the system. This explains why the right side of Figure 2 is "rougher" than the curve in Figure 1, although the peak bandwidth shown is similar.

3.2.5 IBM SP2 measurements

The IBM SP2 at Argonne National Laboratory has Power-1 nodes (the same as in the IBM SP1) and the SP2 high-performance switch. Measurements on IBM SP2 with Power-2 nodes (thin or wide) will be different. The latencies shown in Figure 3 reflect the slower speed of the Power-1 nodes. Note the obvious packet boundaries in the short-message plot.

3.2.6 SGI Power Challenge Measurements

The SGI Power Challenge is a symmetric multiprocessor. The latency and bandwidth performance as shown in Figure 4 indicate the performance for the `ch_shmem` device, a

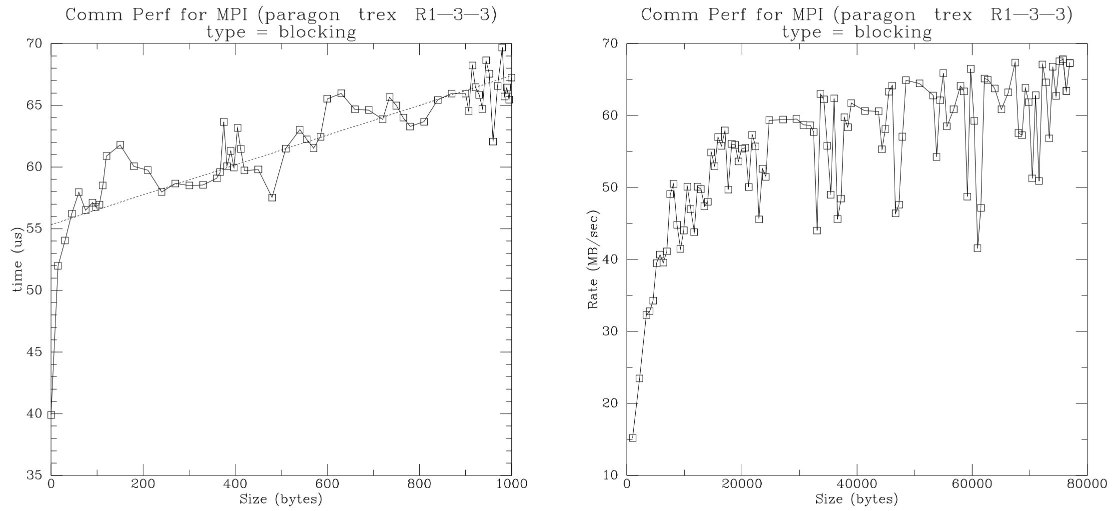

Figure 2: Short and long messages on the Paragon

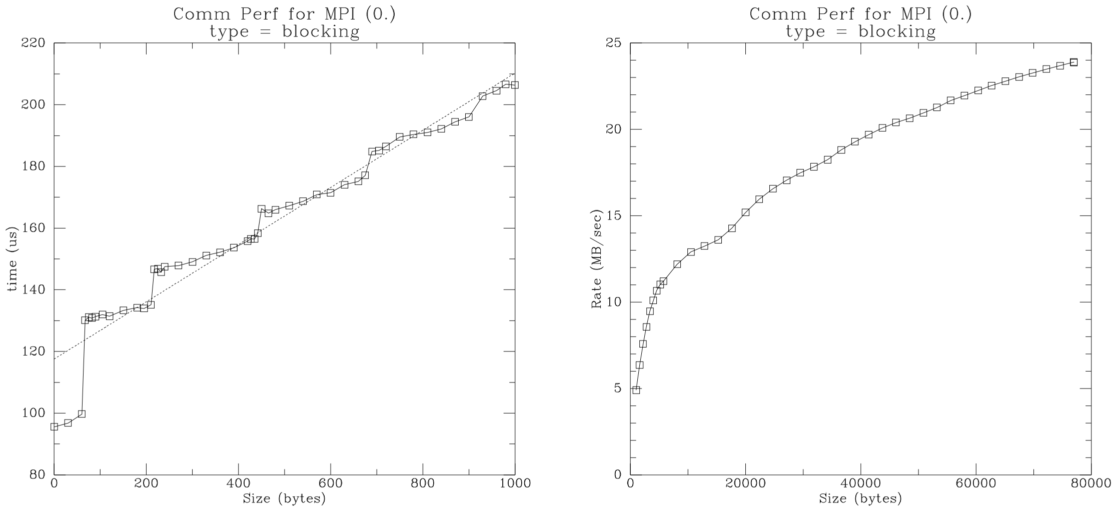

Figure 3: Short and long messages on the IBM SP2

generic shared-memory device supplied with the MPICH implementation.

3.2.7 Cray T3D Measurements

The Cray T3D supports a shared memory interface (the `shmem` library). For MPICH, this library is used to support MPI message-passing semantics. The latency and bandwidth performance are shown in Figure 5.

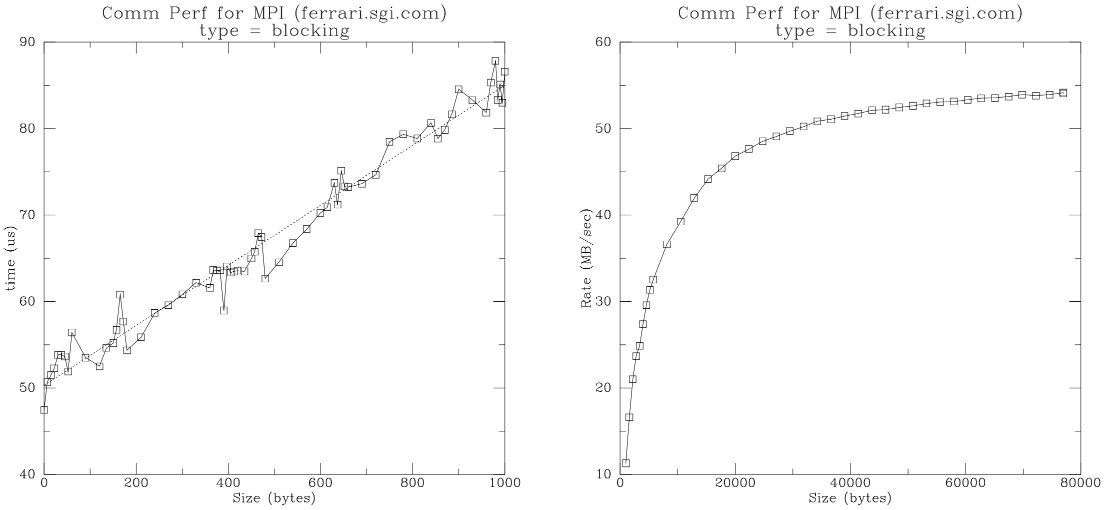

Figure 4: Short and long messages on the SGI Power Challenge

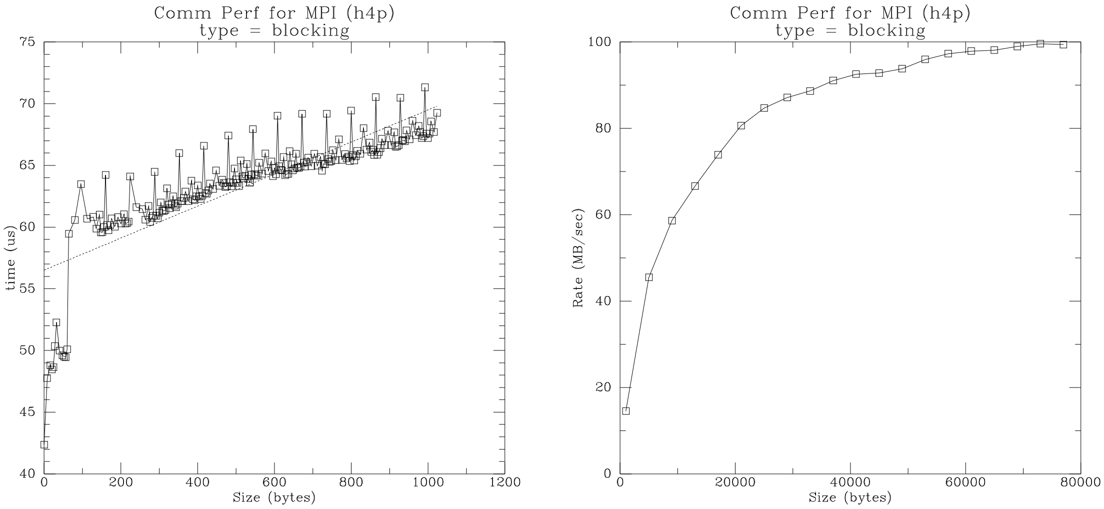

Figure 5: Short and long messages on the Cray T3D

3.2.8 Workstation Network Measurements

Workstation networks connected by simple Ethernet are common. The performance of MPICH for two Sun SPARCStations, on a shared Ethernet, are shown in Figure 6.

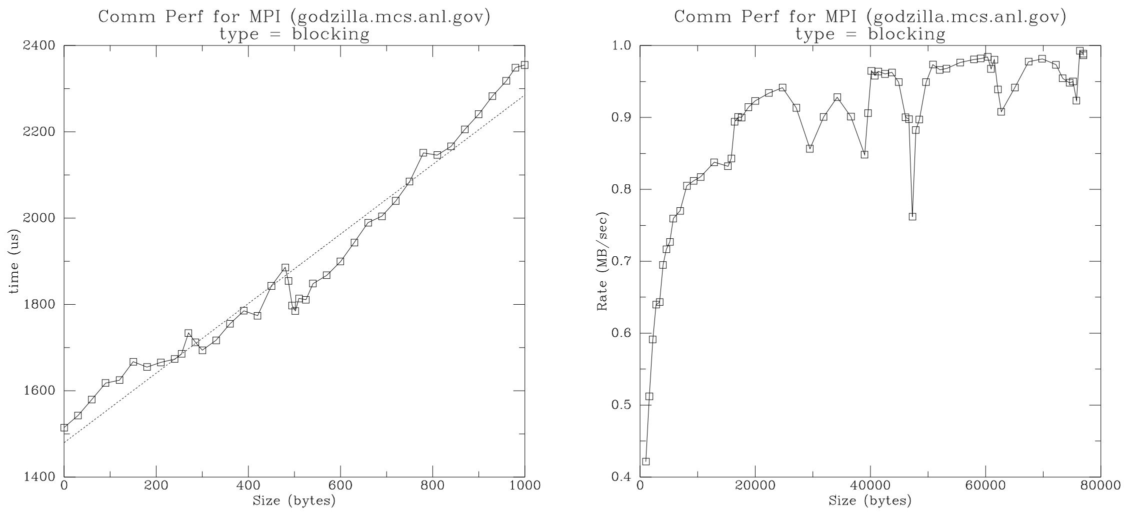

Figure 6: Short and Long Messages on a workstation network

4 Architecture of MPICH

In this section we describe in detail how the software architecture of MPICH supports the conflicting goals of portability and high performance. The design was guided by two principles. First, we wished to maximize the amount of code that can be shared without compromising performance. A large amount of the code in any implementation is system independent. Implementation of most of the MPI opaque objects, including datatypes, groups, attributes, and even communicators, is platform-independent. Many of the complex communication operations can be expressed portably in terms of lower-level ones. Second, we wished to provide a structure whereby MPICH could be ported to a new platform quickly, and then gradually tuned for that platform by replacing parts of the shared code by platform-specific code. As an example, we present in Section 4.3 a case study showing how MPICH was quickly ported and then incrementally tuned for peak performance on SGI shared-memory systems.

The central mechanism for achieving the goals of portability and performance is a specification we call the abstract device interface (ADI) [24]. All MPI functions are implemented in terms of the macros and functions that make up the ADI. All such code is portable. Hence, MPICH contains many *implementations* of the ADI, which provide portability, ease of implementation, and an incremental approach to trading portability for performance. One implementation of the ADI is in terms of a lower level (yet still portable) interface we call the *channel interface* [28]. The channel interface can be extremely small (five functions at minimum) and provides the quickest way to port MPICH to a new environment. Such a port can then be expanded gradually to include specialized implementation of more of the ADI functionality. The architectural decisions in MPICH are those that relegate the implementation of various functions to the channel interface, the ADI, or the application programmer interface (API), which in our case is MPI.

4.1 The Abstract Device Interface

The design of the ADI is complex because we wish to allow for, but not require, a range of possible functions of the device. For example, the device may implement its own message-queuing and data-transfer functions. In addition, the specific environment in which the device operates can strongly affect the choice of implementation, particularly with regard to how data is transferred to and from the user's memory space. For example, if the device code runs in the user's address space, then it can easily copy data to and from the user's space. If it runs as part of the user's process (for example, as library routines on top of a simple hardware device), then the "device" and the API can easily communicate, calling each other to perform services. If, on the other hand, the device is operating as a separate process and requires a context switch to exchange data or requests, then switching between processes can be very expensive, and it becomes important to minimize the number of such exchanges by providing all information needed with a single call.

Although MPI is a relatively large specification, the device-dependent parts are small. By implementing MPI using the ADI, we were able to provide code that can be shared among many implementations. Efficiency could be obtained by vendor-specific proprietary implementations of the abstract device. For this approach to be successful, the semantics of the ADI must not preclude maximally efficient instantiations using modern message-passing hardware. While the ADI has been designed to provide a portable MPI implementation, nothing about this part of the design is specific to the MPI library; our definition of an abstract device can be used to implement any high-level message-passing library.

To help in understanding the design, it is useful to look at some abstract devices for other operations, for example, for graphical display or for printing. Most graphical displays provide for drawing a single pixel at an arbitrary location; any other graphical function can be built by using this single, elegant primitive. However, high-performance graphical displays offer a wide variety of additional functions, ranging from block copy and line drawing to 3-D surface shading. One approach for allowing an API (application programmer interface) to access the full power of the most sophisticated graphics devices, without sacrificing portability to less capable devices, is to define an abstract device with a rich set of functions, and then provide software emulations of any functions not implemented by the graphics device. We use the same approach in defining our message-passing ADI.

A message-passing ADI must provide four sets of functions: specifying a message to be sent or received, moving data between the API and the message-passing hardware, managing lists of pending messages (both sent and received), and providing basic information about the execution environment (e.g., how many tasks are there). The MPICH ADI provides all of these functions; however, many message-passing hardware systems may not provide list management or elaborate data-transfer abilities. These functions are emulated through the use of auxiliary routines, described in [24].

The abstract device interface is a set of function definitions (which may be realized as either C functions or macro definitions) in terms of which the user-callable standard MPI functions may be expressed. As such, it provides the message-passing protocols that distinguish MPICH from other implementations of MPI. In particular, the ADI layer contains the code for packetizing messages and attaching header information, managing multiple buffering policies, matching posted receives with incoming messages or queuing them if necessary, and handling heterogeneous communications. For details of the exact interface and the algorithms used, see [24].

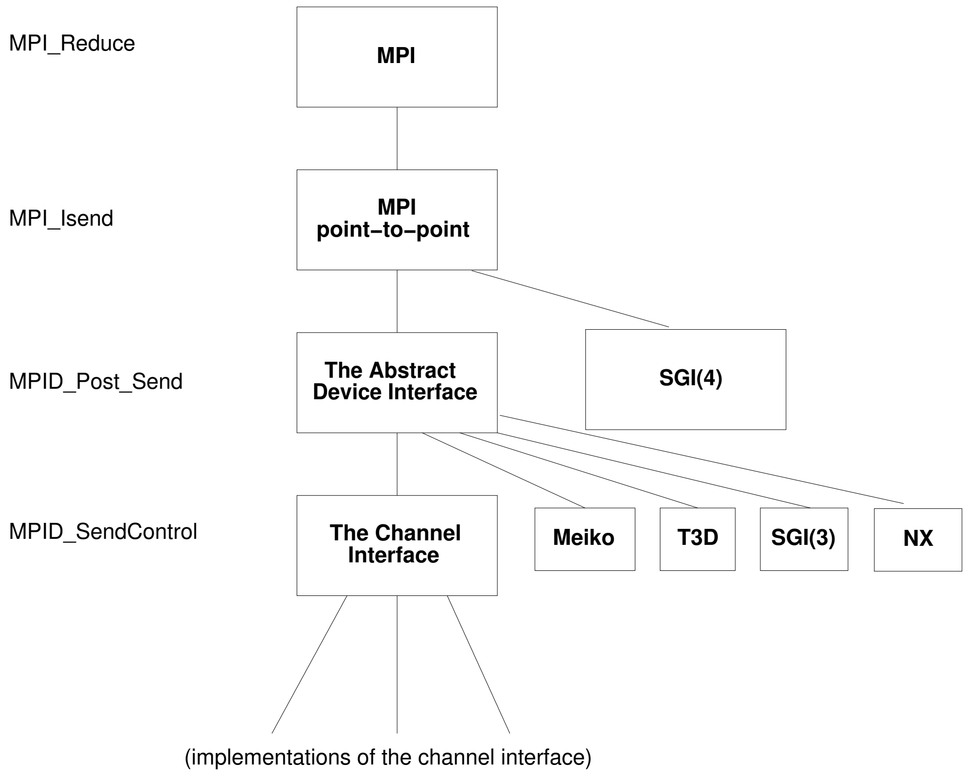

Figure 7: Upper layers of MPICH

A diagram of the upper layers of MPICH, showing the ADI, is shown in Figure 7. Sample functions at each layer are shown on the left. Without going into details on the algorithms present in the ADI, one can expect the existence of a routine like `MPID_SendControl`, which communicates control information. The implementation of such a routine can be in terms of a vendor's own existing message-passing system or new code for the purpose or can be expressed in terms of a further portable layer, the *channel interface*.

4.2 The Channel Interface

At the lowest level, what is really needed is just a way to transfer data, possibly in small amounts, from one process's address space to another's. Although many implementations are possible, the specification can be done with a small number of definitions. The channel interface, described in more detail in [28], consists of only five required functions. Three routines send and receive envelope (or control) information: `MPID_SendControl`,[1] `MPID_RecvAnyControl`, and `MPID_ControlMsgAvail`; two routines send and receive data: `MPID_SendChannel` and `MPID_RecvFromChannel`. Others, which might be available in specially optimized implementations, are defined and used when certain macros are defined

[1] One can use `MPID_SendControlBlock` instead of or along with `MPID_SendControl`. It can be more efficient to use the blocking version for implementing blocking calls.

that signal that they are available. These include various forms of blocking and nonblocking operations for both envelopes and data.

These operations are based on a simple capability to send data from one process to another process. No more functionality is required than what is provided by Unix in the `select`, `read`, and `write` operations. The ADI code uses these simple operations to provide the operations, such as `MPID_Post_recv`, that are used by the MPI implementation.

The issue of buffering is a difficult one. We could have defined an interface that assumed no buffering, requiring the ADI that calls this interface to perform the necessary buffer management and flow control. The rationale for not making this choice is that many of the systems used for implementing the interface defined here do maintain their own internal buffers and flow controls, and implementing another layer of buffer management would impose an unnecessary performance penalty.

The channel interface implements three different data exchange mechanisms.

Eager In the *eager* protocol, data is sent to the destination immediately. If the destination is not expecting the data (e.g., no `MPI_Recv` has yet been issued for it), the receiver must allocate some space to store the data locally.

This choice often offers the highest performance, particularly when the underlying implementation provides suitable buffering and handshakes. However, it can cause problems when large amounts of data are sent before their matching receives are posted, causing memory to be exhausted on the receiving processors.

This is the default choice in MPICH.

Rendezvous In the *rendezvous* protocol, data is sent to the destination only when requested (the control information describing the message is always sent). When a receive is posted that matches the message, the destination sends the source a request for the data. In addition, it provides a way for the sender to return the data.

This choice is the most robust but, depending on the underlying system software, may be less efficient than the eager protocol. Some legacy programs may fail when run using a rendezvous protocol if an algorithm is unsafely expressed in terms of `MPI_Send`. Such a program can be safely expressed in terms of `MPI_Bsend`, but at a possible cost in efficiency. That is, the user may desire the semantics of an eager protocol (messages are buffered on the receiver) with the performance of the rendezvous protocol (no copying) but since buffer space is exhaustible and `MPI_Bsend` may have to copy, the user may not always be satisfied.

MPICH can be configured to use this protocol by specifying `-use_rndv` during configuration.

Get In the *get* protocol, data is read directly by the receiver. This choice requires a method to directly transfer data from one process's memory to another. A typical implementation might use `memcpy`.

This choice offers the highest performance but requires special hardware support such as shared memory or remote memory operations. In many ways, it functions like the rendezvous protocol, but uses a different set of routines to transfer the data.

To implement this protocol, special routines must be provided to prepare the address for remote access and to perform the transfer. The implementation of this protocol allows data to be transferred in several pieces, for example, allowing arbitrarily sized messages to be transferred using a limited amount of shared memory. The routine `MPID_SetupGetAddress` is called by the *sender* to determine the *address* to send to the destination. In shared-memory systems, this may simply be the address of the data (if all memory is visible to all processes) or the address in shared-memory where all (or some) of the data has been copied. In systems with special hardware for moving data between processors, it may be the appropriate handle or object.

MPICH includes multiple implementations of the channel interface (see Figure 8).

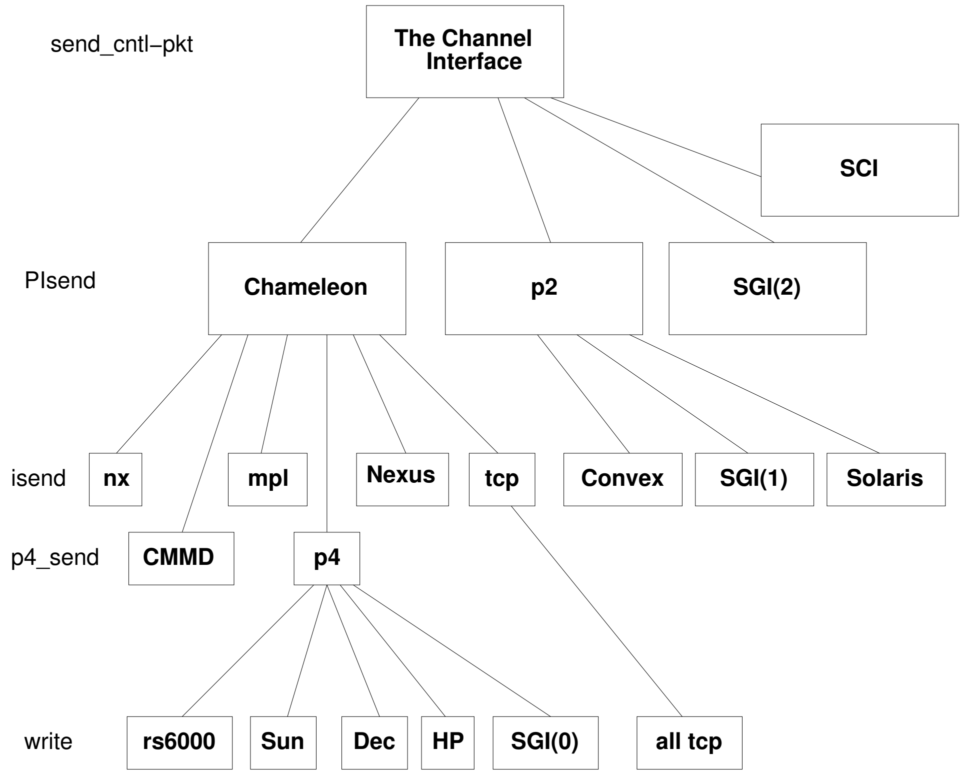

Figure 8: Lower layers of MPICH

Chameleon Perhaps the most significant implementation is the Chameleon version, which was particularly important during the initial phase of MPICH implementation. By implementing the channel interface in terms of Chameleon [31] macros, we provide portability to a number of systems at one stroke, with no additional overhead, since Chameleon macros are resolved at compile time. Chameleon macros exist for most vendor message-passing systems, and also for p4, which in turn is portable to very many systems. A newer implementation of the channel interface is a direct TCP/IP interface, not involving p4.

Shared memory A completely different implementation of the channel interface has been done (portably) for a shared-memory abstraction, in terms of a shared-memory `malloc` and locks. There are, in turn, multiple (macro) implementations of the shared-memory implementation of the channel interface. This is represented as the `p2` box in Figure 8.

Specialized Some vendors (SGI and HP-Convex, at present) have implemented the channel interface directly, without going through the shared-memory portability layer. This approach takes advantage of particular memory models and operating system features that the shared-memory implementation of the channel interface does not assume are present.

SCI A specialized implementation of the channel interface has been developed for an implementation of the Scalable Coherent Interface [40] from Dolphin Interconnect Solutions, which provides portability to a number of systems that use it [39].

Contrary to some descriptions of MPICH that have appeared elsewhere, MPICH has never relied on the p4 version of the channel interface for portability to massively parallel processors. From the beginning, the MPP (IBM SP, Intel Paragon, TMC CM-5) versions used the macros provided by Chameleon. We rely on the p4 implementation only for the workstation networks, and a p4-independent version for TCP/IP will be available soon.

4.3 A Case Study

One of the benefits of a system architecture like that shown in Figures 7 and 8 is the flexibility provided in choosing where to insert vendor-specific optimizations. One illustration of how this flexibility was used is given by the evolution of the Silicon Graphics version of MPICH.

Since Chameleon had been ported to p4 and p4 had been ported to SGI workstations long before the MPICH project began, MPICH ran on SGI machines from the very beginning. This is the box shown as SGI(0) in Figure 8. This implementation used TCP/IP sockets between workstations and standard Unix System V shared memory operations for message passing within a multiprocessor like the SGI Onyx.

The SGI(1) box in Figure 8 illustrates an enhanced version achieved by using a simple, portable shared-memory interface we call p2 (half of p4). In this version, shared memory operations use special SGI operations for shared-memory functions instead of the less robust System V operations.

SGI(2) in Figure 8 is a direct implementation of the channel interface that we did in collaboration with SGI. It uses SGI-specific mechanisms for memory sharing that allow single-copy data movement between processes (as opposed to copying into and out of an intermediate shared buffer), and it uses lock-free shared queue management routines that take advantage of special assembler language instructions of the MIPS microprocessor.

SGI next developed a direct implementation of the ADI that did not use the channel interface model (SGI(3) in Figure 7), and then bypassed the ADI altogether to produce a very high-performance MPI implementation for the Power Challenge Array product, combining both shared-memory operations and message-passing over the HiPPI connections between shared-memory clusters. Even at this specialized level, it retains much of the upper levels of MPICH that are implemented either independently of, or completely on top of, the message-passing layer, such as the collective operations and topology functions.

At all times, SGI users had access to a complete MPI implementation, and their programs did not need to change in any way as the implementation improved.

5 Selected Subsystems

A detailed description of all the design decisions that went into MPICH would be tedious. Here we focus on several of the salient features of this implementation that distinguish it from other implementations of MPI.

5.1 Groups

The basis of an MPICH process group is an ordered list of process identifiers, stored as an integer array. A process's rank in a group refers to its index in this list. Stored in the list is an address in a format the underlying device can use and understand. This is often the rank in `MPI_COMM_WORLD`, but need not be.

5.2 Communicators

The Standard describes two types of communicators, intracommunicators and intercommunicators, which consist of two basic components, namely process groups and communication contexts. MPICH intracommunicators and intercommunicators use this same structure.

The Standard describes how intracommunicators and intercommunicators are related (see Section 5.6 of the Standard). We take advantage of this similarity to reduce the complexity of functions that operate on both intracommunicators and intercommunicators (e.g., communicator accessors, point-to-point operations). Most functions in the portable layer of MPICH do not need to distinguish between an intracommunicator and an intercommunicator. For example, each communicator has a local group (`local_group`) and a remote group (`group`) as described in the definition of an intercommunicator. For intracommunicators, these two groups are identical (reference counting is used to reduce the amount of overhead associated with keeping two copies of a group; see Section 5.1); however, for intercommunicators, these two groups are disjoint.

Another similarity between intracommunicators and intercommunicators is the use of contexts. Each communicator has a send context (`send_context`) and a receive context (`recv_context`). For intracommunicators, these two contexts are equal; for intercommunicators, these contexts may be different. Regardless of the type of communicator, MPI point-to-point operations attach the `send_context` to all outgoing messages and use the `recv_context` when matching contexts upon receipt of a message.

For most MPICH devices, contexts are integers. Contexts for new communicators are allocated through a collective operation over the group of processes involved in the communicator construction. Through this collective operation, all processes involved agree on a context that is currently not in use by any of the processes. One of the algorithms used to allocate contexts involves passing the highest context currently used by a process to an `MPI_Allreduce` with the `MPI_MAX` operation to find the smallest context (an integer) unused by any of the participants.

In order to provide safe point-to-point communications within a collective operation, an additional "collective" context is allocated for each communicator. This collective context is used during communicator construction to create a "hidden" communicator (`comm_coll`) that cannot be accessed directly by the user. This is necessary so that point-to-point operations used to implement a collective operation do not interfere with user-initiated point-to-point operations.

Other important elements of the communicator data structure include the following:

`np, local_rank, lrank_to_grank` Used to provide more convenient access to local group information.

`collops` Array of pointers to functions implementing the collective operations for the communicator (see Section 5.3).

5.3 Collective Operations

As noted in the preceding section, MPICH collective operations are implemented on top of MPICH point-to-point operations. MPICH collective operations retrieve the hidden communicator from the communicator passed in the argument list and then use standard MPI point-to-point calls with this hidden communicator. We use straightforward "power-of-two"-based algorithms to provide scalability; however, considerable opportunities for further optimization remain.

Although the basic implementation of MPICH collective operations uses point-to-point operations, special versions of MPICH collective operations exist. These special versions include both vendor-supplied and shared-memory versions. In order to allow the use of these special versions on a communicator-by-communicator basis, each communicator contains a list of function pointers that point to the functions that implement the collectives for that particular communicator. Each communicator structure contains a reference count so that communicators can share the same list of pointers.

```
typedef struct MPIR_COLLOPS {
    int (*Barrier) (MPI_Comm comm );
    int (*Bcast) (void* buffer, int count, MPI_Datatype datatype,
                        int root, MPI_Comm comm );

    ... other function pointers ...

    int ref_count;     /* So we can share it */
} MPIR_COLLOPS;
```

Each MPI collective operation checks the validity of the input arguments, then forwards the function arguments to the dereferenced function for the particular communicator. This approach allows vendors to substitute system-specific implementations for all or some of the collective routines. Currently, Meiko, Intel, and Convex have provided vendor-specific collective implementations. These implementations follow system-specific strategies; for example, the Convex SPP collective routines makes use both of shared memory and of the memory hierarchies in the SPP.

5.4 Attributes

Attribute caching on communicators is implemented by using a height-balanced tree (HBT or AVL tree) [35]. Each communicator has an HBT associated with it, although initially the HBT may be an empty or null tree. Caching an attribute on a communicator is simply an insertion into the HBT; retrieving an attribute is simply searching the tree and returning the cached attribute.

MPI `keyval`s are created by passing the attribute's copy function and destructor as well as any `extra_state` needed to the `keyval` constructor. Pointers to these are kept in the `keyval` structure that is passed to attribute functions.

Additional elements of a `keyval` include a flag denoting whether C or Fortran calling conventions are to be used for the copy function (the attribute input argument to the copy function is passed by value in C and passed by reference in Fortran).

Caching on other types of MPI handles is being considered for inclusion in the MPI-2 standard. The MPICH HBT implementation of caching can be used almost exactly as is for implementing caching on other types of MPI handles by simply adding an HBT to the other types of handles.

5.5 Topologies

Support for topologies is layered on the communicator attribute mechanism. Because of this configuration, the code implementing topologies is almost entirely portable even to other MPI implementations. For communicators with associated topology information, the communicator's cache contains a structure describing the topology (either a Cartesian topology or a graph topology). The MPI topology functions access the cached topology information as needed (using standard MPI calls), then use this information to perform the requested operation.

5.6 The Profiling Interface

The MPI Forum wished to promote the development of tools for understanding program behavior, but considered it premature to standardize any specific tool interface. The MPI specification provides instead a general mechanism for intercepting calls to MPI functions. Thus both end users and tool developers can develop portable performance analyzers and other tools without access to the MPI implementation source code. The only requirement is that every MPI function be callable (in both C and Fortran) by an alternate name (`PMPI_Xxxx` as well as the usual `MPI_Xxxx`.). In some environments (those supporting "weak symbols") the additional entry points can be supplied in the source code. In MPICH we take the less elegant but more portable approach of building a duplicate MPI library in which all functions are known by their `PMPI_` names. Of course, only one copy of the source code is maintained. Users can interpose their own "profiling wrappers" for MPI functions by linking with their own wrappers, the standard version of the MPI library, and the profiling version of the MPI library in the proper order. MPICH also supplies a number of prebuilt profiling libraries; these are described in Section 6.3.1.

5.7 The Fortran Interface

MPI is a language-independent specification with separate language bindings. The MPI-1.1 standard specifies a C and a Fortran 77 binding. Since these bindings are quite similar, we decided to implement MPI in C, with the Fortran implementation simply calling the C routines. This strategy requires some care, however, because some C routines take arguments by value while all Fortran routines take arguments by reference. In addition, the MPICH implementation uses pointers for the MPI opaque objects (such as `MPI_Request` and `MPI_Comm`); Fortran has no native pointer datatype, and the MPI standard uses the Fortran `INTEGER` type for these objects. Rather than manually create each interface routine, we used a program that had been developed at Argonne for just this purpose.

The program, `bfort` [21], reads the C source file and uses structured comments to identify routines for which to generate interfaces. Special options allow it to handle opaque types, choose how to handle C pointers, and provide name mapping. In many cases, this was all that was necessary to create the Fortran interfaces. In cases where routine-specific code was needed (for example, in `MPI_Waitsome` where zero-origin indexing is used in C and one-origin is used in Fortran), the automatically generated code was a good base to use for the custom code. Using the automatic tool also simplifies updating all of the interfaces when a system with a previously unknown Fortran-C interface is encountered. This situation arose the first time we ported MPICH to a system that used the program `f2c` [14] as a way to provide a Fortran compiler; `f2c` generates unusual external names for Fortran routine names. We needed only to rerun `bfort` to update the Fortran interfaces. This interface handles the issues of pointer conversions between C and Fortran (see Section 8.5) as well as the mapping of Fortran external names to C external names. The determination of the name format (e.g., whether Fortran externals are upper or lower case and whether they have underscore characters appended to them) is handled by our `configure` program, which compiles a test program with the user's selected Fortran compiler and extracts the external name from the generated object file. This allows us to handle different Fortran compilers and options on the same platform.

5.8 Job Startup

The MPI Forum did not standardize the mechanism for starting jobs. This decision was entirely appropriate; by way of comparison, the Fortran standard does not specify how to start Fortran programs. Nonetheless, the extreme diversity of the environments in which MPICH runs and the diversity of job-starting mechanisms in those environments (special commands like `prun`, `poe`, or `mexec`, settings of various environment variables, or special command-line arguments to the program being started) suggested to us that we should encapsulate the knowledge of how to run a job on various machines in a single command. We named it `mpirun`. In all environments, an MPI program, say `myprog`, can be run with, say, 12 processes by issuing the command

```
mpirun -np 12 myprog
```

Note that this might not be the only way to start a program, and additional arguments might usefully be passed to both `mpirun` and `myprog` (see Section 6.4), but the `mpirun` command will always work, even if the starting of a job requires complex interaction with a resource manager. For example, at Argonne we use a home-grown scheduler called EASY instead of IBM's LoadLeveler to start jobs on our IBM SP; interaction with EASY is encapsulated in `mpirun`.

A number of other MPI implementations and environments have also decided to use the name `mpirun` to start MPI jobs. The MPI Forum is discussing whether this command can be at least partially standardized for MPI-2 (see Section 9.4).

5.9 Building MPICH

An important component of MPICH's portability is the ability to build it in the same way in many different environments. We rely on the existence of a Bourne shell `sh` (or superset) and Unix-style `make` on the user's machine. The `sh` script that the user runs is constructed by GNU's `autoconf`, which we need in our development environment, but which the user does not need. At least a vanilla version of MPICH can be built in any of MPICH's target environments by going to the top-level directory of the distribution and issuing the commands

```
configure
make
```

The configure script will determine aspects of the environment (such as the location of certain include files), perform tests of the environment to ensure that all components required for the correct compilation and execution of MPICH programs are present, and construct the appropriate `Makefile`s in many directories, so that the `make` command will build MPICH. After being built and tested, MPICH can be installed in a publicly available location such as `/usr/local` with `make install`. Painless building and installation has become one of our pet goals for MPICH.

5.10 Documentation

MPICH comes with both an installation guide [25] and a user's guide [27]. Although there is some overlap, and therefore some duplication, we consider separating them to be a better approach than combining them. Although many users obtain and use MPICH just for their own use, an increasing number of them are linking their own programs to a system-wide copy of the libraries that have been installed in a publicly accessible place. For such users the information in the installation guide is a distraction. Conversely, the user's guide contains a collection of helpful hints for users who may be experiencing difficulties getting applications to run. These difficulties might well never be encountered by systems administrators who merely install MPICH.

An important but frequently overlooked part of a software project (particular for research software) is the generation of documentation, particularly Unix-style `man` pages.[2] We use a tool called `doctext` [22] that generates man pages (as well as WWW and LaTeX documentation) directly from simple, structured comments in the source code. Using this tool allowed us to deliver MPICH with complete documentation from the beginning. Examples of the documentation can be accessed on the WWW at `http://www.mcs.anl.gov/mpi/www/index.html`.

6 Toward a Portable Parallel Programming Environment

Although MPI specifies a standard library interface and therefore describes what a portable parallel *program* will look like, it says nothing about the *environment* in which the program will run. MPICH is a portable implementation of the MPI standard, but also attempts to provide more for programmers. We have already discussed `mpirun`, which provides a portable way to run programs. In this section we describe briefly some of the other tools provided in MPICH along with the basic MPI implementation.

6.1 The MPE Extension Library

MPE (Multi-Processing Environment) is a loosely structured library of routines designed to be "handy" for the parallel programmer in an MPI environment. That is, most of the MPE functions assume the presence of some implementation of MPI, but not necessarily of MPICH. MPE routines fall into several categories.

Parallel X graphics There are routines to provide all processes with access to a shared X display. These routines are easier to use than the corresponding native `Xlib` routines and make it quite convenient to provide graphical output for parallel programs. Routines are provided to set up the display (probably the hardest part) and draw text, rectangles, circles, lines, etc. on it. It is not the case that the various processes communicate with one process that draws on the display; rather, the display is shared by all the processes. This library is described in [23].

Logging One of the most common tools for analyzing parallel program performance is a time-stamped event trace file. The MPE library provides simple calls to produce such a file. It uses MPI calls to obtain the time-stamps and to merge separate log files together at the end of a job. It also automatically handles the misalignment and drift of clocks on multiple processors, if the system does not provide a synchronized clock. The logfile format is that of `upshot` [33]. This is the library for a user who wishes to define his own events and program states. Automatic generation of events by MPI routines is described in Section 6.3.1.

Sequential Sections Sometimes, a section of code that is executed on a set of processes must be executed by only one process at a time, in rank order. The MPE library provides functions to ensure that this type of execution occurs.

[2]This is not to say that the format of man pages cannot be improved; rather, every Unix user knows how to get information this way and rightly expects man pages to be provided.

Error Handling The MPI specification provides a mechanism whereby a user can control how the implementation responds to run-time errors, including the ability to install one's own error handler. One error handler that we found convenient for developing MPICH starts the `dbx` debugger in a popup `xterm` when an error is encountered. Thus, the user can examine the stack trace and values of program variables at the time of the error. To obtain this behavior, the user must

1. Compile and link with the `-g` option, as usual when using `dbx`.
2. (a) Link with the MPE library.
   Call

   ```
   MPI_Errhandler_set( comm, MPE_Errors_call_dbx_in_xterm )
   ```

   early in the program,
   OR
   (b) Pass the `-mpedbg` argument to `mpirun` (if MPICH configured with `-mpedbg`).

6.2 Command-Line Arguments and Standard I/O

The MPI standard says little about command-line arguments to programs, other than that in C they are to be passed to `MPI_Init`, which removes the command line arguments it recognizes. MPICH ensures that on each process, the command-line arguments returned from `MPI_Init` are the same on all processes, thus relieving the user of the necessity of broadcasting the command-line arguments to the rest of the processes from whichever process actually was passed them as arguments to `main`.

The MPI Standard also says little about I/O, other than that if at least one process has access to `stdin`, `stdout`, and `stderr`, the user can find out which process this is by querying the attribute `MPI_IO` on `MPI_COMM_WORLD`. In MPICH, all processes have access to `stdin`, `stdout`, and `stderr`, and on networks these I/O streams are routed back to the process with rank 0 in `MPI_COMM_WORLD`. On most systems, these streams also can be redirected through `mpirun`, as follows.

```
mpirun -np 64 myprog -myarg 13 < data.in > results.out
```

Here we assume that "`-myarg 13`" are command-line arguments processed by the application `myprog`. After `MPI_Init`, each process will have these arguments in its `argv`. (This is an MPICH feature, not an MPI requirement.) On batch systems where `stdin` may not be available, one can use an argument to `mpirun`, as follows.

```
mpirun -np 64  -stdin data.in myprog -myarg 13 > results.out
```

The latter form may always be used.

6.3 Support for Performance Analysis and Debugging

The MPI profiling interface allows the convenient construction of portable tools that rely on intercepting calls to the MPI library. Such tools are "ultra portable" in the sense that they can be used with any MPI implementation, not just a specific portable MPI implementation.

6.3.1 Profiling Libraries

The MPI specification makes it possible, but not particularly convenient, for users to build their own "profiling libraries," which intercept all MPI library calls. MPICH comes with three profiling libraries already constructed; we have found them useful in debugging and in performance analysis.

tracing The tracing library simply prints (on `stdout`) a trace of each MPI library call. Each line is identified with its process number (rank in `MPI_COMM_WORLD`). Since `stdout` from all processes is collected, even on a network of workstations, all output comes out on the console. A sample is shown here.

```
    ...
[1] Starting MPI_Bcast...
[0] Starting MPI_Bcast...
[0] Ending MPI_Bcast
[2] Starting MPI_Bcast...
[2] Ending MPI_Bcast
[1] Ending MPI_Bcast
    ...
```

logging The logging library uses the `mpe` logging routines described in Section 6.1 to write a logfile with events for entry to and exit from each MPI function. Then `upshot` (see Section 6.3.2) can be used to display the computation, and its colored bars will show the frequency and duration of each MPI call. (See Figure 9.)

animation The animation library uses the `mpe` graphics routines to provide a simple animation of the message passing that occurs in an application, via a shared X display.

Further description of these libraries can be found in [34].

6.3.2 Upshot

One of the most useful tools for understanding parallel program behavior is a graphical display of parallel timelines with colored bars to indicate the state of each process at any given time. A number of tools developed by various groups do this. One of the earliest of these was `upshot` [33]. Since then `upshot` has been reimplemented in Tcl/Tk, and this version [34] is distributed with MPICH. It can read log files generated either by Paragraph [32] or by the `mpe` logging routines, which are in turn used by the logging profiling library. A sample screen dump is shown in Figure 9.

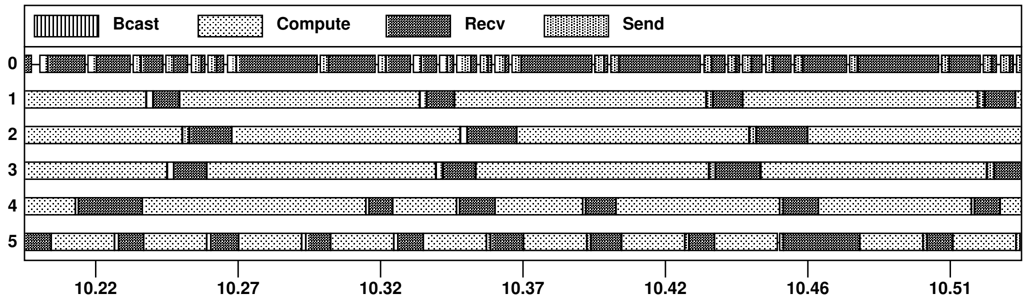

Figure 9: Upshot output

6.3.3 Support for Adding New Profiling Libraries

The most obvious way to use the profiling library is to choose some family of calls to intercept, and then treat each of them in a special way. Typically, one performs some action (adds to a counter, prints a message, writes a log record), calls the "real" MPI function using its alternate name `PMPI_Xxxx`, perhaps performs another action (e.g., writes another log record), and then returns to the application, propagating the return code from the `PMPI` routine.

MPICH includes a utility called `wrappergen` that lets a user specify "templates" for profiling routines and a list of routines to create, and then automatically creates the profiling versions of the specified routines. Thus the work required by a user to add a new profiling library is reduced to writing individual `MPI_Init` and `MPI_Finalize` routines and one template routine. The libraries described above in Section 6.3.1 are all produced in this way. Details of how to use `wrappergen` can be found in [27].

6.4 Useful Commands

Aspects of the environment required for correct compilation and linking are encapsulated in the `Makefile`s produced when the user runs `configure`. Users may set up `Makefile` for their own applications by copying one from an MPI examples directory and modifying it as needed. The resultant `Makefile` may not be portable, but this may not be a primary consideration.

An even easier and more portable way to build a simple application, and one that fits within existing complex `Makefile`s, is to use the commands `mpicc` or `mpif77`, constructed in the MPICH '`bin`' directory by `configure`. These scripts are used like the usual commands to invoke the C and Fortran compilers and the linker. Extra arguments to these commands link with the designated versions of profiling libraries. For example,

```
mpicc -c myprog.c
```

compiles a C program, automatically finding the include libraries that were configured when MPICH was installed. The command

```
mpif77 -mpilog -o myprog myprog.f
```

compiles and links a Fortran program that, when run, will produce a log file that can be examined with upshot. The command

```
mpicc -mpitrace -o myprog myprog.c
```

compiles and links a C program that displays a trace of its execution on stdout.

The mpirun command has already been mentioned. It has more flexibility than we have described so far. In particular, in heterogeneous environments, the command

```
mpirun -arch sun4 -np 4 -arch rs6000 -np 3 myprog
```

starts myprog on four Sun4's and three RS/6000's, where the specific hosts have been stored in MPICH's "machines" file.

Special arguments for the application program can be used to make MPICH provide helpful debugging information. For example,

```
mpirun -np 4 myprog -mpedbg -mpiqueue
```

automatically installs the error handler described in Section 6.3.1 that starts dbx on errors, and display all message queues when MPI_Finalize is called. This latter option is useful in locating "lost" messages.

Details on all of these commands can be found in the user's guide [27].

6.5 Network Management Tools

Although not strictly part of MPICH itself, the Scalable Unix Tools (SUT) [26] are a useful part of the MPICH programming environment on workstation clusters. Basically, SUT implements parallel versions of common Unix commands such as ls, ps, cp, or rm. Perhaps the most useful is a cross between find and ps that we call pfps (parallel find in the process space). For example, one can find and send a KILL signal to runaway jobs on a workstation network during a debugging session with

```
pfps -all -tn myprog -kill KILL
```

or locate all of one's own jobs on the network that have been running for more than an hour with

```
pfps -all -o me -and -rtime 1:00 -print
```

Graphical displays also show the load on each workstation and can help one choose the sub-collection of machines to run an MPICH job on. Details can be found in [26].

6.6 Example Programs

MPICH comes with a fairly rich collection of example programs to illustrate its features. In addition to the extensive test suite and benchmark programs, there are example programs for Mandelbrot computations, solving the Mastermind puzzle, and the game of life that illustrate the use of the `mpe` library in an entertaining way. A number of simple examples illustrate specific features of the MPI Standard (topologies, for example) and have been developed for use in classes and tutorials. Many of the examples from [29] are included. For all of these examples, `configure` prepares the appropriate `Makefiles`, but they have to be individually built as the user wishes. One example is a moderately large complete nuclear physics Monte Carlo integration application in Fortran.

7 Software Management Techniques and Tools

MPICH was written by a small, distributed team sharing the workload. We had the expected problems of coordinating both development and maintenance of a moderately large (130,000 lines of C) and complex system. We have worked to distribute new releases in an orderly fashion, track and respond to bug reports, and maintain contact with a growing body of users. In doing so, we have used existing tools, engineered some of our own, and developed procedures that have served us well. In this section we report on our experiences, in the hope that some of our tools and methods will be useful to other system developers. All software described here is freely available, either from well-known sources or included in MPICH.

7.1 Configuring for Different Systems

We have tried, as a sort of pet goal, to make building MPICH completely painless, despite the variety of target environments. This is a challenge. In earlier systems, such as p4, Chameleon, and Zipcode, it was assumed that a particular vendor name or operating system version was enough to determine how to build the system. This is too simplistic a view:

- The same hardware may run multiple operating systems (Solaris or SunOS on suns, LINUX or FreeBSD on x86's)
- Different versions of the same operating system may differ radically (SGI IRIX 5 is 32 bit, whereas IRIX 6 is 64; the number of parameters to some system calls in Solaris depends on the minor version number).
- Different compilers may use different includes, datatype sizes, and libraries.

In addition, it is rare that a system is completely free of bugs; in particular, since we distribute source code, it is imperative that the C compiler produce correct object code. In distributing MPICH, we found that many users did *not* have correctly functioning C compilers. It is best to determine this problem at configure time.

We use the GNU `autoconf` system to build a shell script (`configure`), which in turn executes various commands (including building and running programs) to determine the user's environment. It then creates `Makefile`s from makefile templates, as well as creating some scripts that contain site-specific information (such as the location of the `wish` interpreter).

The `autoconf` system as distributed provides commands for checking the part of a system that the GNU tools need; in MPICH, we have defined an additional set of operations that we found we needed in this and other projects. These include commands to test that the compiler produces correct code and to choose a vendor's compiler (with the correct options; this is particularly important for the massively parallel systems). In short, the `configure` script distributed with MPICH has evolved into a knowledge base about a wide variety of vendor environments.

7.2 Source Code Management

To allow us all to work on the code without interfering with one another, we used RCS via the Emacs VC interface.

7.3 Testing

Testing is often a lengthy and boring process. MPICH contains a test suite that attempts to test the implementation of each MPI routine. While not as systematic as some commercial testing systems, which can dwarf the size of the original package, our test suite has proven invaluable to us in identifying problems before new releases. Many of the test programs originated as bug-demonstration programs sent to us by our users. In MPICH, we automate the testing process through the use of scripts that build, test, and generate a document that summarizes the tests, including the configuration, correctness, and performance results. The testing system is provided as part of the distribution.

7.4 Tracking and Responding to Problem Reports

We realized early on that simply leaving bug reports in our email list would not work. We needed a system that would allow all of the developers to keep track of reports, including what had been done (dialog with problem submitter, answers, etc.). It also had to be simple for users to use and for us to install. We chose the `req` system [13]. This system allows users to send mail (without any format restrictions) to `mpi-bugs@mcs.anl.gov`; the bug report is kept in a separate system as well as being forwarded to a list of developers. Both GUI and command-line access to the bug reports is provided.

Over time, it became clear that some problems were much more common than others. We developed a database of common problems, searchable by keyword, which is also integrated into the manual. When a user sends in a bug report, we can query the database for a standard response to the problem. For example, if a user complains about getting the message "`Try Again`," the command

```
> mpich/help/bin/fmsg 'Try Again'
```

gives the information from the user's guide on the message "`Try Again`" (which comes not from MPICH but from `rshd`).

Announcements about new releases are sent to a mailing list (managed by `majordomo`) to which users are encouraged to subscribe when they first run the `configure` script.

7.5 Preparing a New Release

Preparing a new release for a package as portable as MPICH requires testing on a wide variety of platforms. To help test a new release of MPICH, we use several programs and scripts that build and test the release on a new platform. The `doc/port` program included in the MPICH distribution performs a build, checking for errors from the `make`, followed by both performance and correctness tests. The output from this program is a postscript file that describes the results of the build and tests; this postscript file is then made available on the WWW in a table at `http://www.mcs.anl.gov/mpi/mpich/porting/port`*version-number*`.html`. Another program then is used to do an installation and to check that both MPICH and the other tools (such as `upshot` and `mpicc`) work correctly. For networks of workstations, additional tests (also managed by a separate program) test heterogeneous collections of workstations. By automating much of the testing, we ensure that the testing is reasonably complete and that the most glaring oversights are caught before a release goes out. Unfortunately, because the space of possible tests is so large, these programs and scripts have been built primarily by testing for past mistakes.

8 Lessons Learned

One of the purposes of doing an early implementation was to understand the implications of decisions made during the development of the Standard. As expected, the implementation process and early experiences of users shed light on the consequences of choices made at MPI Forum meetings.

8.1 Language Bindings

One of the earliest lessons learned had to do with the language bindings and choices of C datatypes for some items in the MPI Standard. For example, the 1.0 version passed the `MPI_Status` structure itself, rather than a pointer to the structure, to the routines `MPI_Get_count`, `MPI_Get_elements`, and `MPI_Test_cancelled`. The C bindings used `int` in some places where an `int` might not be large enough to hold the result; most of these (except for `MPI_Type_size`) were changed to `MPI_Aint`.

In the `MPI_Keyval_create` function, the predefined "null" functions `MPI_NULL_COPY_FN` and `MPI_NULL_DELETE_FN` were originally both `MPI_NULL_FN`; unfortunately, neither of these is exactly a null function (both have mandatory return values and the copy function also sets a flag). Experience with the implementation helped the MPI Forum to repair these problems in the 1.1 version of the MPI Standard.

A related issue was the desire of the Forum to make the same attribute copy and delete functions usable from both C and Fortran; for this reason, addresses were used in the 1.0 standard for some items in C that were more naturally values. Unfortunately, when the size of the C `int` datatype is different from that of the Fortran `INTEGER` datatype, this approach does not work. In a surprise move, the MPI Forum exploited this in the 1.1 Standard, *changing* the bindings of the functions in C to use values instead of addresses.

Another issue is the Fortran bindings of all of the routines that take buffers. These buffers can be of any Fortran datatype (e.g., `INTEGER`, `REAL`, or `CHARACTER`). This was common practice in most previous message-passing systems but is in violation of the Fortran Standard [15]. The MPI Forum voted to follow standard practice. In most cases, Fortran compilers pass all items by reference, and few complain when a routine is called with different datatypes. Unfortunately, several standard-conforming Fortran implementations use a different representation for `CHARACTER` data than for numeric data, and in these cases it is difficult to build an MPI implementation that works with `CHARACTER` data in Fortran. The MPI Forum is attempting to address this problem in the MPI-2 proposal.

The MPI Forum provided a way for users to interrogate the environment to find out, for example, what was the largest valid message tag. This was done in an elegant fashion by using "attributes," a general mechanism for users to attach information to a communicator. The system attributes are attached to the initial `MPI_COMM_WORLD` communicator. The problem is that, in general, users need to set as well as get attributes. Some users did in fact try to set `MPI_TAG_UB`. MPICH now detects this as an illegal operation, and the MPI Forum clarified this in the 1.1 Standard.

8.2 Performance

One of the goals of MPI was to define the semantics of the message passing operations so that no unnecessary data motion was required. The MPICH implementation has shown this goal to be achievable. On two different shared-memory systems, MPICH achieves a single copy directly from user-buffer to user-buffer. In both cases, the operating system had to be modified slightly to allow a process to directly access the address space of another process. On distributed memory systems, two vendors were able to achieve the same result by providing vendor-specific implementations of the ADI.

In actual use, some users have noticed some performance irregularities; these indicate areas where more work needs to be done in implementations. For example, the implementation of `MPI_Bsend` in MPICH always copies data into the user-provided buffer; for small messages, such copying is not always necessary (it may be possible to deliver the message without blocking). This can have a significant effect on latency-sensitive calculations. Different methods for handling short, intermediate, and long messages are also needed and are under development.

Another source of some performance difficulties is seemingly innocuous requirements that affect the lowest levels of the implementation. For example, the following is legal in MPI:

```
MPI_Isend( ..., &request );
MPI_Request_free( &request );
```

The user need not (must not, actually) use a wait or test on the `request`. This functionality can be complex to implement when well-separated software layers are used in the MPI implementation. In particular, it requires that either the completion of the operation started by the `MPI_Isend` change data maintained by the MPI implementation or that the MPI implementation periodically check to see whether some request has completed. The problem with this functionality is that it may not match well with the services that are implementing the actual data transport, and can be the source of unanticipated latency.

Despite these problems, the MPICH implementation does achieve its goal of high performance and portability. In particular, the use of a carefully layered design, where the layers can be implemented as macros (or removed entirely, as one vendor has done), was key in the success of MPICH.

8.3 Resource Limits

The MPI specification is careful not to constrain implementations with specific resource guarantees. For many uses, programmers can work within the limits of any "reasonable" implementation. However, many existing message-passing systems provide some (usually unspecified) amount of buffering for messages sent but not yet received. This allows a user to send messages without worrying about the process blocking waiting for the destination to receive them or worrying about waiting on nonblocking send operations. The problem with this approach is that if the system is responsible for managing the buffer space, user programs can fail in mysterious ways. A better approach is to allow the user to specify the amount of buffering desired. The MPI Forum, recognizing this need, added routines with user-provided buffer space: `MPI_Bsend`, `MPI_Buffer_attach`, and `MPI_Buffer_detach` (and nonblocking versions). These routines specify that all of the space needed by the MPI implementation can be found in the user-provided buffer, including the space used to manage the user's messages. Unfortunately, this made it impossible for users to determine how big a buffer they needed to provide, since there was no way to know how much space the MPI implementation needed to manage each message. The MPI Forum added `MPI_BSEND_OVERHEAD` to provide this information in the 1.1 version of the Standard.

One remaining problem that some users are now experiencing is the limit on the number of outstanding `MPI_Request`s that are allowed. Currently, no *a priori* way exists to determine or provide the number of allowed requests.

8.4 Heterogeneity and Interoperability

Packed data needs to be sent with a "packed data" bit; this means that datatypes need to know whether any part of the datatype is `MPI_PACKED`. The only other option is to always use the same format, for example, network byte order, at the cost of maximum performance.

Many systems can be handled by using byte swapping; with data extension (e.g., 32-bit to and from 64-bit integers), most systems can be handled. In some cases, only floating point requires special treatment; in these cases, XDR may be used where IEEE format is not guaranteed.

The MPI specification provides `MPI_PACK` and `MPI_UNPACK` functions; unfortunately, these are *not* the functions that are needed to implement the point-to-point operations. The reason is that these functions produce data that can be sent to anyone in a communicator (including the sender), whereas when sending to a single, other process, there is more freedom in choosing the data representation.[3] The MPICH implementation uses internal versions of `MPI_PACK` and `MPI_UNPACK` that work with data intended either for a specific process or for all members of a communicator.

8.5 64-bit Issues

The development of MPICH coincided with the emergence of a number of "64-bit systems." Many programmers, remembering the problems moving code from 16- to 32-bit platforms, expressed concern over the problem of porting applications to the 64-bit systems. Our experience with MPICH was that, with some care in using C properly (`void *` and not `int` for addresses, for example), there was little problem in porting MPICH from 32- to 64-bit systems. In fact, with the exception discussed below, MPICH has no special code for 32- or 64-bit systems.

The exception is in the Fortran-C interface, and this requires an understanding of the rules of the Fortran 77 Standard. While C makes few statements about the length of datatypes (for example, `sizeof(int)` and `sizeof(float)` are unrelated), Fortran defines the *ratios* of the sizes of the numeric datatypes. Specifically, the sizes of `INTEGER` and `REAL` data are the same, and are half the size of `DOUBLE PRECISION` [15]. This is important in Fortran 77, where there is no memory allocation in the language and programmers often have to reuse data areas for different types of data. Further, using 64-bit IEEE floating point for `DOUBLE PRECISION` *requires* that `INTEGER` be 32 bits. This is true even if `sizeof(int)` (in C) is 64 bits.

In the Fortran-C interface, this problem appears when we look at the representation of MPI opaque objects. In MPICH, they are pointers; if these are 64 bits in size, then they cannot be stored in a Fortran `INTEGER`. (If opaque objects were `int`s, it would not help much; we would still need to convert from a 64-bit to 32-bit integer.) Thus, on systems where addresses are 64 bits and Fortran `INTEGER`s are shorter, something must be done. The MPICH implementation handles this problem by translating the C pointers to and from small Fortran integers (which represent the index in a table that holds the pointer). This translation is inserted automatically into the Fortran interface code by the Fortran interface generator `bfort` (discussed in section 5.7).

Another problem involves the routine `MPI_Address`, which returns an "address" of an item. This "address" may be used in only two ways: relative to another "address" from `MPI_Address` or relative to the "constant" `MPI_BOTTOM`. In C, the obvious implementation is to set `MPI_BOTTOM` to zero and use something like `(long)(char *)ptr` to get the address that `ptr` represents. But in Fortran, the value `MPI_BOTTOM` is a variable (at a known location). Since all arguments to routines in Fortran are passed by address,[4] the best approach is to have the Fortran version of `MPI_Address` return addresses *relative* to the address of `MPI_BOTTOM`. The advantage of this approach is that even when absolute addresses

[3] Strangely, the MPI Forum considered but did not accept the functions needed for packing and unpacking data sent between two specific processes; this decision may have been because there was less experience with heterogeneous environments.

[4] Value-result if one must be picky; in practice, the addresses are passed.

are too large to fit in an INTEGER, in many cases the address relative to a location in the user's program (i.e., MPI_BOTTOM) will fit in an INTEGER. This is the approach used in MPICH; if the address does not fit, an error is returned (of class MPI_ERR_ARG, with an error code indicating that the address won't fit).

As a final step in ensuring portability to 64-bit systems, our configure program runs some programs to determine whether the system is 32 or 64 bits. This allows MPICH to port to unknown systems or to systems like SGI's IRIX that change from 32-bit (IRIX 5) to 64-bit (IRIX 6) without any changes to the code.

8.6 Unresolved Issues

The MPI Forum did not address any mixed-language programming issues. At least for MPI-1, Fortran programs must pass messages to Fortran programs, and the same for C. Yet, it is clearly possible to support both C-to-C and Fortran-to-Fortran message passing in a single application. We call this a "horizontal mixed-language portability." As long as there is no interest in transferring anything other than user data between Fortran and C strata of the parallel application, the horizontal model can be satisfied, provided that MPI_Init provides a consistent single initialization of MPI for both languages, regardless of which language is used actually to initialize MPI. Current practice centers on this "horizontal" model, but it is clearly insufficient, as we have observed from user feedback.

Two additional levels of support are possible, staying still with the restriction of C and Fortran 77 as the mixed languages. The first is the ability to pass MPI opaque objects locally within a process between C and Fortran. As noted earlier, C and Fortran representations for MPI objects will often be arbitrarily different, as will addresses. Although user-accessible interoperable functions already are required in MPICH (for the benefit of its Fortran interface), the MPI Standard does not require them. Such functionality is likely to appear in MPI-2 (as a result of our users' experience) and with other MPI systems as well. Such functionality has the added benefit of enhancing the ability of third parties to provide add-on tools for both C and Fortran users, without working with inside knowledge of the MPICH implementation (for instance, see [6]).

The second level of "vertical" support is to allow a C routine to transmit data to a Fortran routine. This requires some correspondence between C and Fortran datatypes, as well as a common format for performing the MPI operations (e.g., the C and Fortran implementations must agree on how to send control information and perform collective operations). The MPI Forum is preparing a proposal that addresses the issues of interlanguage use of MPI datatypes for MPI-2.

9 Status and Plans

We begin this section by describing the current use of MPICH by vendors (as a component of their own MPI implementations) and others. We then describe some of our plans for improving MPICH both by optimizing some of its algorithms for better performance and by extending its portability into other environments.

9.1 Vendor Interactions

As described above, one of the motivations for MPICH's architecture was to allow vendors to use MPICH in developing their own proprietary MPI implementations. MPICH is copyrighted, but freely given away and automatically licensed to anyone for further development. It is not restricted to noncommercial use. This approach has worked well, and vendor implementations are now appearing, many incorporating major portions of MPICH code.

- IBM obtained an explicit license for MPICH and collaborated with us in testing and debugging early versions. During this time, MPI-F [19] appeared. This IBM implementation does not use the ADI, but maps MPI functions directly onto an internal IBM abstract device interface. Our contact at IBM was Hubertus Franke.
- SGI worked closely with us (see Section 4.3) to improve the implementation of the ADI for their Challenge and Power Challenge machines. Functions were added to IRIX to enable single-copy interprocess data movement, and SGI gave us lock-free queue-management routines in assembler language. Those involved at SGI were Greg Chesson and Eric Salo.
- Convex worked closely with us to optimize an implementation of the channel interface and then of the ADI. We worked with Paco Romero, Dan Golan, Gary Applegate, and Raja Daoud.
- Intel contributed a version of the ADI written directly for NX, bypassing the channel interface. The Intel person responsible was Joel Clarke.
- Meiko also contributed to the publicly distributed version a Meiko device, thanks to the efforts of Jim Cownie.
- Laurie Costello helped us adapt MPICH for the Cray vector machines.
- DEC has used MPICH as the foundation of a memory-channel-based MPI for Alpha clusters.

We obviously do not claim credit for the vendor implementations, but it does appear that we met our original goal of accelerating the adoption of MPI by vendors through providing them a running start on their implementations. The architecture of MPICH, which provided multiple layers without impact on performance, was the key.

9.2 Other Users

Since we make MPICH publicly available by `ftp`, we do not have precise counts on the number of users. It is downloaded about 300 times per month from our `ftp` site, `ftp.mcs.anl.gov`, which is also mirrored at Mississippi State, `ftp.erc.msstate.edu`. Judging from the bug reports and subscriptions to the `mpi-users` mailing list, we estimate that between five hundred and one thousand people are currently active in their use of MPICH.

9.3 Planned Enhancements

We are pursuing several directions for future work based on MPICH.

New ADI To further reduce latencies, particularly on systems where latency is already quite low, we plan an enhanced ADI that will enable MPICH to take advantage of low-level device capabilities.

Better collective algorithms As mentioned in Section 5.3, the current collective operations are implemented in a straightforward way. We would like to incorporate some of the ideas in [1] for improved performance.

Thread safety The MPI specification is thread-safe, and considerable effort has gone into providing for thread safety in MPICH, but this has not been seriously tested. The primary obstacle here is the availability of a test suite for thread safety of MPI operations.

Dynamic, lighter-weight TCP/IP device We are nearing completion of a portable device that will replace p4 as our primary device for TCP/IP networks. It will be lighter weight than p4 and will support dynamic process management, which p4 does not.

RDP/UDP device We are working on a reliable data protocol device approach, built on UDP/IP (User datagram protocol), which extends and leverages the initial work done by D. Brightwell [3].

Multiprotocol support Currently MPICH can use only one of its "devices" at a time. Although two of those devices, the one based on Nexus [18] and the one based on p4, are to a certain extent multiprotocol devices, we need a general mechanism for allowing multiple devices to be active at the same time. We are designing such a mechanism now. This will allow, for example, two MPPs to be used at the same time, each using its own switches for internal communication and TCP/IP for communication between the two machines.

Ports to more machines We are working with several groups to port MPICH to interesting new environments. These include

- the Parsytec machine;
- NEC SX-4 and Cenju-3;
- Microsoft Windows NT, both for multiprocessor servers and across the many different kinds of networks that NT will support; and
- Network protocols that are more efficient than TCP/IP, both standard (for example, MessageWay [10]) and proprietary (for example, Myrinet [2]).

Parallel I/O We have recently begun a project to determine whether the concepts of the ADI can be extended to include parallel I/O. If this proves successful, we will include an experimental implementation of parts of MPI-IO [11, 12] into MPICH.

9.4 MPI-2

In March 1995, the MPI Forum resumed meeting, with many of its original participants, to consider extensions to the original MPI Standard. The extensions fall into several categories:

- Dynamic creation of processes (e.g., `MPI_SPAWN`).
- One-sided operations (e.g., `MPI_PUT`).
- Extended collective operations, such as collective operations on intercommunicators.
- External interfaces (portable access to fields in MPI opaque objects).
- C++ and Fortran-90 bindings.
- Extensions for real-time environments.
- Miscellaneous topics, such as the standardization of `mpirun`, new datatypes, and language interoperability.

The MPICH project began as a commitment to implement the MPI-1 Standard, with the aim of assisting in the adoption of MPI by both vendors and users. In this goal it has been successful. The degree to which MPI-2 functionality will be incorporated into MPICH depends on several factors:

- The actual content of MPI-2, which is far from settled at this time.
- The degree to which the MPI-2 specification mandates features whose implementation would be feasible only with major changes to MPICH internals.
- The enthusiasm of MPICH users for the individual MPI-2 features.

At this writing, it seems highly likely that we will extend MPICH to include dynamic process management as defined by the MPI-2 Forum, at least for the workstation environment. This extension will not be difficult to do with the new implementation of the channel interface for TCP/IP networks, and it is the feature most desired by those developing workstation-network applications. We expect also to aid tool builders (including ourselves) by providing access to MPICH internals specified in the MPI-2 "external interfaces" specification. For the other parts of MPI-2, we will wait and see.

10 Summary

We have described MPICH, a portable implementation of the MPI Standard that offers performance close to what specialized vendor message-passing libraries have been able to deliver. We believe that MPICH has succeeded in popularizing the MPI Standard and encouraging vendors to provide MPI to their customers, first, by helping to create demand, and second, by offering them a convenient starting point for proprietary implementations.

We have also described the programming environment that is distributed with MPICH. The simple commands

```
configure
make
cd examples/basic
mpicc -mpilog -o cpi cpi.c
mpirun -np 4 cpi
upshot cpi.log
```

provide a *portable* sequence of actions by which even the beginning user can install MPICH, run a program, and use a sophisticated tool to examine its behavior. These commands are the same, and the user's program is the same, on MPPs, SMPs, and workstation networks. MPICH demonstrates that such portability need not be achieved at the cost of performance.

11 Acknowledgments

We thank the entire MPICH user community for their patience, feedback, and support of this project. Furthermore, we thank Gail Pieper for her help with proofreading and suggesting improvements to the presentation of the material.

We thank Ralph Butler (University of North Florida) for his help in adapting parts of p4 to support MPICH.

We acknowledge several students who contributed to this work: Thomas McMahon and Ron Brightwell (Mississippi State), Patrick Bridges (University of Arizona), and Ed Karrels (University of Wisconsin).

References

[1] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor collective communications library (intercomm). In *Proceedings of the Scalable High Performance Computing Conference*, pages 356–364. IEEE Computer Society Press, 1994.

[2] Nanette J. Boden et al. Myrinet—a Gigabit-per-Second Local-Area Network. *IEEE-Micro*, 15(1):29–36, 1995.

[3] David Brightwell. The Design and Implementation of a Datagram Protocol (UDP) Device for MPICH. Master's thesis, December 1995. Dept. of Computer Science.

[4] Ronald Brightwell and Anthony Skjellum. Design and Implementation Study of MPICH for the Cray T3D. In preparation, February 1996.

[5] R. Alasdair A. Bruce, James G. Mills, and A. Gordon Smith. CHIMP/MPI user guide. Technical Report EPCC-KTP-CHIMP-V2-USER 1.2, Edinburgh Parallel Computing Centre, June 1994.

[6] Greg Burns and Raja Daoud. MPI Cubix—collective POSIX I/O operations for MPI. Technical Report OSC-TR-1995-10, Ohio Supercomputer Center, 1995.

[7] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[8] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[9] Lyndon J. Clarke, Robert A. Fletcher, Shari M. Trewin, R. Alasdair A. Bruce, A. Gordon Smith, and Simon R. Chapple. Reuse, portability and parallel libraries. In *Proceedings of IFIP WG10.3—Programming Environments for Massively Parallel Distributed Systems*, 1994.

[10] Danny Cohen and Craig Lund. Proposed Standard for the MessageWay Inter-SAN Routing. IETF Working Group Document, September 1995.

[11] Peter Corbett, Dror Feitelson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: A parallel file I/O interface for MPI, version 0.3. Technical Report NAS-95-002, NAS, January 1995.

[12] Peter Corbett, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, Parkson Wong, and Dror Feitelson. MPI-IO: A parallel file I/O interface for MPI, version 0.4. `http://lovelace.nas.nasa.gov/MPI-IO`, December 1995.

[13] Remy Evard. Managing the ever-growing to do list. In *USENIX Proceedings of the Eighth Large Installation Systems Administration Conference*, pages 111–116, 1994.

[14] S. I. Feldman and P. J. Weinberger. A portable Fortran 77 compiler. In *UNIX Time Sharing System Programmer's Manual*, volume 2. AT&T Bell Laboratories, tenth edition, 1990.

[15] American National Standard Programming Language Fortran. ANSI X3.9-1978.

[16] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on `netlib`.

[17] The MPI Forum. The MPI message-passing interface standard. `http://www.mcs.anl.gov/mpi/standard.html`, May 1995.

[18] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus task-parallel runtime system. In *Proc. 1st Intl. Workshop on Parallel Processing,*, pages 467–462. Tata McGraw Hill, 1994.

[19] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, Jean-Pierre Prost, and Marc Snir. MPI on IBM SP1/SP2: Current status and future directions. Technical report, IBM T. J. Watson Research Center, 1995.

[20] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[21] William Gropp. Users manual for `bfort`: Producing Fortran interfaces to C source code. Technical Report ANL/MCS-TM-208, Argonne National Laboratory, March 1995.

[22] William Gropp. Users manual for `doctext`: Producing documentation from C source code. Technical Report ANL/MCS-TM-206, Argonne National Laboratory, March 1995.

[23] William Gropp, Edward Karrels, and Ewing Lusk. MPE graphics—scalable X11 graphics in MPI. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 49–54. Mississippi State University, IEEE Computer Society Press, October 1994.

[24] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Preprint MCS-P342-1193, Argonne National Laboratory, 1994.

[25] William Gropp and Ewing Lusk. Installation guide for `mpich`, a portable implementation of MPI. Technical Report ANL-96/5, Argonne National Laboratory, 1994.

[26] William Gropp and Ewing Lusk. Scalable Unix tools on parallel processors. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 56–62. IEEE Computer Society Press, 1994.

[27] William Gropp and Ewing Lusk. User's guide for `mpich`, a portable implementation of MPI. Technical Report ANL-96/6, Argonne National Laboratory, 1994.

[28] William Gropp and Ewing Lusk. MPICH working note: Creating a new MPICH device using the channel interface. Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1995.

[29] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[30] William D. Gropp and Ewing Lusk. A test implementation of the MPI draft message-passing standard. Technical Report ANL-92/47, Argonne National Laboratory, December 1992.

[31] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[32] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

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

[34] Edward Karrels and Ewing Lusk. Performance analysis of MPI programs. In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 195–200. SIAM, 1994.

[35] Donald Knuth. *The Art of Computer Programming, Vol. 3*. Addison-Wesley, 1973.

[36] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[37] Joerg Meyer. Message passing interface for Microsoft Windows 3.1. Master's thesis, University of Nebraska at Omaha, December 1994.

[38] Joerg Meyer and Hesham El-Rewini. WinMPI: Message passing interface for Microsoft Windows 3.1. `/ftp://csftp.unomaha.edu/pub/rewini/WinMPI/`, 1994.

[39] Para//ab, University of Bergen. *Programmer's Guide to MPI for Dolphin's SBus-to-SCI adapters*, version 1.0 edition, November 1995. Available at ftp://ftp.ii.uib.no/pub/incoming/progr-guide-v10.ps.

[40] IEEE standard for scalable coherent interface (SCI) (1-55937-222-2) [SH15255]. IEEE 596-1992, 1993.

[41] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The design and evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994.

[42] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[43] Georg Stellner. Cocheck: Checkpointing and process migration for MPI. In *Proceedings of the IPPS*. IEEE Computer Society Press, 1996.

[44] Georg Stellner and Jim Pruyne. Resource Management and Checkpointing for PVM. In *Proceedings of the 2nd European PVM Users' Group Meeting*, pages 131–136, Lyon, September 1995. Editions Hermes.

[45] Paula L. Vaughan, Anthony Skjellum, Donna S. Reese, and Fei Chen Cheng. Migrating from PVM to MPI, part I: The Unify System. In *Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 488–495, McLean, Virginia, February 1995. IEEE Computer Society Technical Committee on Computer Architecture, IEEE Computer Society Press.

[46] David Walker. Standards for message passing in a distributed memory environment. Technical report, Oak Ridge National Laboratory, August 1992.