A Survey of MPI Implementations

William Saphir
Lawrence Berkeley National Laboratory
University of California
Berkeley, CA 94720

November 6, 1997

Abstract

The Message Passing Interface (MPI) standard has enabled the creation of portable and efficient programs for distributed memory parallel computers. Since the first version of the standard was completed in 1994, a large number of MPI implementations have become available. These include several portable implementations as well as optimized implementations from every major parallel computer manufacturer. The ubiquity and high quality of MPI implementations has been a key to the success of MPI. This review describes and evaluates a number of these implementations.

1 Introduction and History

The early days of parallel computing[1] were characterized by experimentation, proof-of-concept demonstrations, and a willingness to re-implement programs from scratch for every new computer that came along. This is a fine way to learn how to do parallel computation, but a lousy way to build the infrastructure necessary for growth and stability and for making parallel computing interesting to anyone but academics. Such infrastructure requires standardization.

During 1993 and 1994, a group of representatives of the computer industry, government labs and academia met to develop a standard interface for the "message passing" model of parallel programming. This organization, known as the Message Passing Interface (MPI) Forum, finished its work in June, 1994, producing an industry standard known as MPI [1]. Since this initial (1.0) standard, the

[1] In this article, *parallel computing* always refers to scientific computing on distributed memory multiprocessors.

MPI Forum has produced versions 1.1 (June, 1995) and 1.2 (July, 1997), which correct errors and minor omissions, and version 2.0 (July, 1997), which adds substantial new functionality to MPI-1.2 [2]. At the time of this writing, there are not yet any full MPI 2.0 implementations. In the rest of this document, MPI refers to MPI 1.1 unless otherwise noted.

An MPI program consists of a set of processes and a logical communication medium connecting those processes. These processes may be the same program (SPMD - Single Program Multiple Data) or different programs (MPMD - Multiple Programs Multiple Data). The MPI memory model is logically distributed: an MPI process cannot directly access memory in another MPI process, and interprocess communication requires calling MPI routines in both processes.[2] MPI defines a library of subroutines through which MPI processes communicate — this library is the core of MPI and implicitly defines the programming model.

The most important routines in the MPI library are the so-called "point-to-point" communication routines, which allow processes to exchange data cooperatively — one process sends data to another process, which receives the data. This cooperative form of communication is called "message passing."

The MPI standard is a specification, not a piece of software. What is specified is the application interface, not the implementation of that interface. In order to allow implementors to implement MPI efficiently, the MPI standard does not specify protocols, or require that implementations be able to interoperate. Moreover, so that MPI can make sense in a wide range of environments, the standard does not specify how processes are created or destroyed, and does not even specify precisely what a process is.

The most important considerations in the design of MPI were:

- **Portability.** An MPI application should require only recompilation to use a different MPI implementation. Furthermore, it should be possible to implement MPI on any MIMD (Multiple Instruction, Multiple Data) parallel computer. MPI should support (though not require) execution in heterogeneous environments.

- **Efficiency.** It should be possible to implement MPI efficiently. In particular, good MPI implementations should perform as well as proprietary "native"

[2]In the MPI model, processes may be implemented within the same virtual address space, but all data is private to a process, and data in other processes can be accessed only through MPI subroutines. MPI does not forbid coexistence with other models, though interaction with these other models is not defined by MPI. MPI-2, which includes so-called one-sided communication, still presents a distributed memory model. MPI-2 also more clearly defines how a multithreaded MPI process behaves.

message passing libraries.

- **Robustness.** MPI should provide all important functionality in "common practice," and then some. MPI provides significant support for the development of parallel libraries.

This review discusses several representative implementations of the MPI standard. These include MPICH, LAM and CHIMP, which are freely available multi-platform implementations, as well as optimized implementations supplied and supported by SGI/Cray, IBM, HP, Digital, Sun and NEC. While an attempt has been made to report on the most visible implementations, a few less well-known ones may have been left out, as well as implementations for hardware that is no longer available.

The primary conclusion is that MPI implementations are almost all of high quality, both robust and efficient. While there are minor problems here and there, the application developer considering using MPI can be confident that there is a well-supported MPI implementation on almost every commercially important parallel computer. Performance of MPI implementations is usually close to what the hardware can provide, though this review does not discuss performance, as discussed in Section 3.

While this is the main story, there are a few side stories, having to do with behavior the MPI standard does not specify, such as the integration of MPI applications into a parallel environment, tools for tracing and debugging, handling of standard input and output, documentation, buffering strategies, etc. These side stories are as much the subject of this review as are the MPI implementations themselves. The goal of this review is to orient the potential MPI user in the world of MPI and to describe interesting features of MPI implementations, rather than to compare and rank them, which would be an unproductive exercise.[3]

2 Relation of MPI to PVM and HPF

In the context of software standards for parallel computing, two other names are bound to pop up — Parallel Virtual Machine (PVM) and High Performance Fortran (HPF). Both of these have close ties to MPI. While this review is about MPI, it is intended as an orientation for new users, and to this end, it is appropriate to see how MPI fits in the larger context.

[3] A disclaimer: One person's "feature" is another's "bug,", and the opinions in this review are subjective. Furthermore, MPI is an area of active development, with new versions released often. The information in this article is believed to be current as of its writing.

PVM is a package of software that provides message passing functionality as well as infrastructure for building a virtual parallel computer out of a network of workstations [3]. It is often thought of as a competitor to MPI, but it is really a different beast. PVM is a research project of the University of Tennessee at Knoxville and Oak Ridge National Laboratory. While quite popular for writing message passing programs, PVM is a vehicle for performing research in parallel computing rather than a parallel computing standard. Its weaknesses with respect to MPI are also its strengths: it is not bound by an absolute requirement for backward compatibility; its design is not constrained to be efficient on every imaginable MIMD parallel architecture; there is no rigorous specification of PVM behavior. In some sense the tradeoff is between efficiency and portability in MPI, and flexibility and adaptability in PVM.

Successful features of PVM are finding their way into MPI, though MPI is unlikely to provide any support for fault tolerance or a virtual distributed operating system in the near future. Moreover, since PVM is defined by a full implementation rather than a specification, possibilities for interoperability in PVM are higher than in MPI.

HPF is an industry standard for the data parallel model of parallel computation. HPF was standardized a year earlier than MPI, and the successful HPF standardization process was copied by the MPI Forum. Despite the conceptual appeal and simplicity of HPF, MPI is much more widely used than HPF, for several reasons. These include:

- HPF is much more difficult to implement, and to implement efficiently. MPI, on the other hand, has benefited greatly from the large number of good implementations, including an implementation that was available at about the same time the standard was released.
- MPI is a more general model, and can be used to implement almost any parallel computation, while HPF is applicable only to certain types of problems.
- Obtaining high performance in an HPF program can be more difficult than would be expected from the superficial simplicity of the HPF model. It is an open question whether this is a fundamental obstacle or can be addressed by more mature compilers.

3 Review Criteria

This review looks at several different aspects of MPI implementations. Some of the important ones are:

Compliance. One of the main reasons for having a standard is that a code written using one MPI implementation should be able to use another implementation without any source code changes. There are several test suites that can automatically find problems in an MPI implementation. These include a test suite from IBM [4], a test suite from Intel [5], and test codes distributed with MPICH [6], which is described in Section 4.

In the implementations surveyed for this study, there are few problems with MPI compliance. The problems are relatively minor and often have their origins in the behavior of earlier versions of MPICH, from which many implementations are derived. The two most common problems are:

1. Several implementations do not provide an `MPI_Cancel` that meets the MPI specification. This function can be quite difficult to implement for anything other than unmatched receives. Indeed, a significant fraction of the MPI community believes that it was a mistake for MPI to require that this function apply to sends as well, and that applications requiring the ability to cancel sends are quite rare. In the opinion of this reviewer, therefore, not fully implementing `MPI_Cancel` is a small problem.

2. Several implementations do not correctly implement MPI constants in Fortran. In particular, MPI 1.1 requires that they be usable in initialization expressions. For example, the following code should work.

   ```
   include 'mpif.h'
   integer mytype
   parameter (mytype=MPI_REAL)
   ```

   In some implementations, however, many MPI "constants" are actually the names of variables in common blocks and aren't initialized until `MPI_Init` has been called. Note that this behavior was compliant with version 1.0 of MPI, and that MPI 1.1 invalidated some formerly compliant implementations (though all compliant MPI *applications* remained compliant). However, MPI 1.1 has been been out for more than two years, and new vendor implementations continue to contain this bug.

Unspecified Behavior. MPI does not specify behavior of some aspects of MPI implementations. These include

- **Buffering.** MPI does not specify how much buffering must be provided by an implementation.

- **Standard I/O.** MPI does not require that standard language I/O facilities be provided.

- **Process startup and management.** MPI does not specify how processes are started up or shut down. In particular, the state of a program before `MPI_Init` and after `MPI_Finalize` is not specified.

It is where MPI does not specify behavior that implementations differ. Applications that do not make assumptions about these features are portable. Portability issues are discussed in more detail in Section 7.

Integration with the environment. One of the most important aspects of the usability of an MPI implementation is how well MPI applications are integrated into the environment. Some of the important issues are:

- Is the operating system or some other system software aware of a parallel application as a distinct entity, rather than as a collection of unrelated serial processes?

- Does the implementation provide flexibility in handling standard input and output, such as optional broadcasting of `stdin` and per-process labeling of `stdout`. Is standard output available, without excessive buffering, from all processes?

- Is process management robust? Is a terminal interrupt propagated to all processes in an MPI application? Can a parallel application leave unkilled "orphaned" processes that must be detected and killed off manually?

In answering these questions, it is often difficult to separate the MPI implementation from the parallel environment. This review tries to keep the focus on MPI itself.

Performance. It is difficult to separate the performance of an MPI implementation from the performance of the underlying hardware. Since this report is not about hardware (see [7] for information about hardware) performance is only a secondary issue. Also, it is not easy to give a complete picture of performance. For instance, latency and bandwidth numbers give only a small part of the overall picture. Therefore this review does not discuss absolute performance numbers.

Tools. Parallel applications are inherently more difficult to develop and tune then serial applications. This difficulty is compounded by a lack of tools for developing parallel programs. The most important development tools are a parallel debugger and performance analysis tools, including tools for message trace visualization. A parallel debugger should not require a separate window for each process, and should understand and be able to display MPI message queues.

Tools are separate from the MPI implementation itself, and are treated only superficially in this report.

4 MPICH

Without question, the most important MPI implementation is MPICH[4], a freely available portable implementation of MPI developed at Argonne National Laboratory and Mississippi State University [6]. MPICH has played an important role in the development of MPI.

MPICH is the parent of a large number of commercial implementations of MPI. These include vendor-supported implementations from Digital, Sun, HP, SGI/Cray, NEC and Fujitsu. In some cases (e.g., SGI and HP) the implementation has evolved far from its roots; in others (e.g., Digital and Sun) the implementation is young and still close to MPICH. Only two of the major vendor-supported implementations are not directly derived from MPICH: the Cray T3D/E implementation (which derives from the CHIMP implementation) and the IBM SP implementation (for which MPICH still provided substantial inspiration). The HP implementation also has a second parent in LAM. MPICH is also the basis for most experimental and research versions of MPI.

The first version of MPICH was written during the MPI standardization process. The experiences of the MPICH authors provided important feedback to the MPI Forum, including a proof-by-example that it was not necessary to define a subset of MPI in order to make the implementation of MPI less burdensome. MPICH was released at approximately the same time as the original MPI 1.0 standard.

The portability of MPICH stems from its two-layer design. The bulk of MPICH code is device independent and is implemented on top of an Abstract Device Interface (ADI). The ADI interface hides most hardware-specific details, allowing MPICH to be easily ported to new architectures. The ADI design allows for efficient layering, and the device-independent top layer takes care of the majority of MPI syntax and semantics.

[4]MPICH is pronounced "em-pee-eye-see-aitch" not "empitch"

With release 1.1 of MPICH (current as of this writing) the ADI layer was changed. The new interface, known as ADI-2, is not yet proven on a large number of architectures, though it was designed using lessons learned from ADI-1. ADI-2 should add new opportunities for performance. For example, a set of optional datatype functions expose MPI datatypes to the implementor, allowing him or her to take advantage of special hardware or use optimized techniques for transferring noncontiguous datatypes.

Each implementation of the ADI layer is called a "device," and each MPICH device effectively defines a new implementation. The following sections describe the devices that are distributed with MPICH.

4.1 General comments on all devices

Goodies. MPICH comes with a large amount of supporting material, including a number of examples and test programs, utilities for compiling and running MPI programs, a program that automatically generates profiling wrappers and Unix manual pages. There is also a library called MPE (MultiProcessing Environment) that contains routines for producing event logs, simple run-time visualization, and timing. Finally there are performance visualization tools called "upshot" and "nupshot" that display data from trace files (which can be produced by MPE).

Building MPICH. MPICH is easy to configure and build. A complex "configure" script recognizes almost all common systems. The only drawback to this is that many parts of MPICH wind up with hard-coded path names (relative to the MPICH installation directory) and it can be painful to extract pieces of the MPICH distribution to use elsewhere.

Debugging. Since MPICH is portable, it has no built-in debugging support. For some devices, it is possible to start the root process under a debugger, or to start the entire application under a parallel debugger. The best debugging option for MPICH is the Totalview debugger [8], a commercial product from Dolphin Interconnect Solutions that is one of the best parallel debuggers available. Totalview has a good GUI with an intuitive "dive" feature and a coherent approach to debugging parallel programs. It is integrated with MPICH (the `ch_p4`, `ch_shmem`, and `ch_lfshmem` devices), understands MPI communicators and MPI message queues and "captures" MPI processes as they start. Totalview runs on IBM RS6000, Sun and Digital platforms. By the end of 1997 it is expected to run on SGI R10000 platforms as well.

Problems. MPICH does not implement `MPI_Cancel` according to the MPI specification (it can't always cancel sends). As described above, this "bug" has been repeated in many MPICH-derived implementations but is not too important. An earlier version of MPICH used Fortran common blocks to implement some Fortran MPI constants. This is fixed (with a lot of work) in the current MPICH release (1.1), but remains in many MPICH-derived implementations. While MPICH is generally carefully written, there are many MPICH routines that are not implemented as efficiently as they could be. Sometimes these are fixed in MPICH-derived implementations, but often not. An example is the `MPI_Alltoall` routine, which is implemented in a way that is likely to cause hot spots in any network.

4.2 The `ch_p4` device

This is the "network of workstations" implementation of MPICH. P4 (Portable Programs for Parallel Processors) is an older message passing library that was used to implement the MPICH ADI[9]. The "ch" in "`ch_p4`" stands for "channel." The ADI is in fact implemented in terms of a simpler "channel" interface, and the channel interface is implemented in terms of P4. The layering is not strict.

The `ch_p4` device is characterized by the following.

- P4 runs on Sun/SunOS, Sun/Solaris, Solaris86, Cray, HP, Dec 5000, Dec Alpha, Next, IBM RS6000, Linux86, FreeBSD, IBM3090, SGI (5, 6), and others.
- The device uses process-to-process sockets, for processes not on the same host, or shared memory (using the "-comm=shared" configuration flag), for processes on the same host.
- The user provides a list of programs and machines to start them on in a P4 "procgroup" file. P4 starts remote processes using `rsh` (or optionally, using a "secure server" that provides faster startup). I/O and signal propagation are handled by `rsh`. P4 processes start a "listener" subprocess that helps to establish process-to-process connections if there aren't enough TCP connections to fully connect the MPI application.
- An interesting feature of the `ch_p4` device is that the user starts up a single process, and that process starts the other MPI processes inside `MPI_Init`. To do this, `ch_p4` relies on the `argc` and `argv` arguments to `MPI_Init`.
- The `ch_p4` device handles heterogeneous MPI applications — applications with processes running on more than one architecture. Data representation

conversion, if needed, is automatically performed.

While the `ch_p4` device provides a way to run MPICH on networks of workstations, it is not very friendly to users.

- The procgroup file is difficult to work with and the documentation is not easy to find. Fortunately the complexity is often hidden behind local utilities or an "mpirun" command.

- There is no concept of a "virtual machine." Unlike PVM, the network of workstations used by an application is defined by where the application is running, not by an infrastructure that exists before and persists afterwards. Consequently, there are no "ps" or "kill" equivalents that understand parallel jobs, and no automatic way to examine the state of remote nodes or perform load balancing. The lack of such infrastructure also contributes to the signal propagation and I/O problems described below. In some cases, the lack of machine state is a bonus, particularly when MPI programs are started automatically by a batch system.

- Because signal propagation is managed through `rsh`, it is very easy to end up with "orphaned" processes that don't realize the rest of an application has gone away. These orphaned processes often interfere with the running of subsequent parallel jobs and are difficult to find.

- Because standard I/O relies on `rsh`, output from remote nodes is often heavily buffered, and doesn't appear on the screen until well after it is written. This can make debugging with `printf` very difficult.

4.3 The `ch_shmem` and `ch_lfshmem` devices

The `ch_shmem` and `ch_lfshmem` devices allow communication through shared memory on a number of SMP platforms. `ch_lfshmem` uses so-called lock-free queues to reduce synchronization overhead and is therefore preferred. Both devices run on most SMP platforms including Sun, SGI, HP and Digital platforms.

Usability of applications using these shared memory devices is high, because process management is handled by normal operating system mechanisms. E.g. process startup is trivial, `ps` shows every process in a parallel job (though still doesn't have the concept of a parallel process), signals are propagated (usually) and I/O is buffered to the same extent as normal language I/O.

However, these devices are restricted to communication within an SMP, aren't thread-safe, and aren't themselves multithreaded. All of these are potentially desirable in SMP clusters.

4.4 The ch_nexus Device

The ch_nexus device uses Nexus for its underlying communication layer and for process management. Nexus is discussed elsewhere [10]. This device is still in a preliminary state, and has so far only been tested in a Solaris environment.

The most interesting use of ch_nexus is that Nexus provides multimethod communication, which can use optimized communication where possible. ch_nexus thus provides an interoperability mechanism to allow MPI to run efficiently on clusters of heterogeneous systems.

4.5 MPP devices: ch_cenju3, ch_mpl, ch_nx, nx, t3d

The MPP[5] devices all implement MPI on top of native communication. For the devices whose names start with with ch_, the device uses the simpler channel interface, minimizing the amount of platform-dependent code. For the other devices, the ADI is implemented directly on the native communication library.

For all MPP devices, MPICH uses the native mechanism for starting and managing processes, for handling I/O, etc.

The performance of these MPICH implementations is in most cases very close to that of the underlying communication layer.

5 Other Freely Available Implementations

5.1 LAM — Local Area Multicomputer

The LAM implementation of MPI is a freely available and portable implementation developed at the Ohio Supercomputer Center [11]. LAM and MPICH are the two most important free options for running MPI on a network of workstations. LAM existed before MPI and was adopted to implement the MPI interface. LAM runs on many platforms, including RS6000, Irix 5, Irix 6, Linux86, HPUX, OSF/1 and Solaris.

LAM provides an infrastructure to turn a network of workstations (possibly heterogeneous) into a virtual parallel computer. A user-level daemon running on each node provides process management, including signal handling and I/O management. LAM also provides extensive monitoring capabilities to support tuning and debugging. The xmpi tool that comes with LAM (and has since been adopted

[5]"MPP" used to stand for "Massively Parallel Processor and then "Moderately Parallel Processor," but these days just means a highly integrated and tightly coupled distributed memory parallel computer, sold as a single machine.

by the HP and SGI implementations as well) provides visualization of message traces and allows inspection of message queues.

By default, full message monitoring is enabled, and communication goes through the daemons. It is also possible to enable direct client-to-client communication using TCP sockets or shared memory.

LAM also provides so-called Guaranteed Envelope Resources (GER), which is a promise about how much pending communication LAM can support. Such a guarantee is missing from the MPI specification, and there is debate over whether it is needed or not. In theory, a compliant MPI implementation could have so few internal resources (e.g. buffers for message envelopes) that reasonable MPI programs would fail. Fortunately, there are few MPI implementations which fail due to resource exhaustion for average codes, and most have tunable parameters to allow them to deal with unusual codes. Through GER, LAM makes explicit quantitative guarantees on resource availability.

LAM is compliant with MPI 1.1 and also implements dynamic process management routines from MPI-2.

Usability. LAM is a good solution for networks of workstations, but has a number of usability problems. With some small changes, it has the possibility to become the "PVM" of the MPI world because of its ability to turn a network of workstations into a parallel computer. From the user's point of view, it does not go quite far enough in a few key areas. For instance, the "virtual machine" abstraction is incomplete: there is no easy way to find out what hosts are part of the virtual machine; the equivalent to `ps` requires a specification of node numbers; starting up a daemon when there is already one running kills the old one, rather than noticing that one is already running. LAM also hurts itself by using strange names — such as `wipe` instead of something like `lamhalt`, putting include files in `share/h` instead of `include`, insisting on using the word "schema" whenever possible; commands that should be able to figure out the current machine state (e.g. `wipe`) require a schema when they should be able to figure out the information themselves.

Tools. LAM is considered one of the best environments for development because of its extensive monitoring capabilities. The `xmpi` tool, which is distributed with LAM, allows visualization of message traces and examination of message queues in deadlocked programs. LAM also provides better I/O handling and fewer opportunities for orphaned processes than MPICH on networks of workstations.

5.2 CHIMP — Common High-level Interface to Message Passing

The CHIMP project is based at the Edinburgh Parallel Computing Centre [12]. Like LAM, CHIMP started off as an independent portable message passing infrastructure and was later adapted to implement MPI. CHIMP is best known as the basis for the vendor-supplied optimized versions of MPI for the Cray T3D and T3E. Chimp is portable, running on many platforms including Solaris, Irix, AIX, OSF/1, and Meiko. To the best of this reviewer's knowledge, CHIMP is not in active development and is not widely used, at least in the U.S.

5.3 Other implementations

A number of implementations are based on MPICH but are not included with the standard MPICH release. Some of these are listed here.

NT Students at Mississippi State University have developed an MPICH ADI implementation for Microsoft NT clusters. It is a demonstration implementation, rather than a high performance implementation.
See `http://www.erc.msstate.edu/mpi/mpiNT.html` for more information.

Win32 A researcher in Portugal has implemented the MPICH ADI device for Microsoft Windows.
See `http://alentejo.dei.uc.pt/ fafe/w32mpi/` for more information.

Active Messages A student at the University of California at Berkeley has implemented the MPICH ADI (ADI-2) on top of Generic Active Messages (GAM) and Active Messages II (AM2).
See `http://now.cs.berkeley.edu/Fastcomm/MPI/` for more information.

Fast Messages Students at the University of Illinois at Urbana Champaign have implemented the MPICH ADI on top of Fast Messages, which runs on PCs running NT or Linux with Myrinet or Winsock 32.
See `http://www-csag.cs.uiuc.edu/projects/comm/mpi-fm.html` for more information.

Multithreaded (MT) Device A researcher in Germany has implemented the MPICH ADI so that MPI "processes" are in fact threads on a multiprocessor machine. Communication between these processes can be done with a single copy.

The MT device runs on Linux with the Nthreads library and on Solaris with the Nthreads and Solaris threads libraries. To handle the problem of non-private global variables, there is support for private global variables through a preprocessor or a modified gcc compiler. This approach is quite similar to what was done for the Cray PVP implementation described in Section 6.5.1.
See `http://noah.informatik.tu-chemnitz.de/members/radke/mtdevice/mtdevice.html` for more information.

TransMPI TransMPI is an implementation of MPI for transputers (MPI 1.0, C bindings only). It is of interest because it is a full implementation of MPI, unrelated to MPICH, LAM or CHIMP, to which most other implementations trace their origins. For more information, contact *thomasd@netcom.com*.

6 Vendor Implementations

6.1 IBM

IBM has been a consistently strong supporter of MPI. IBM's implementation of MPI for its SP systems was one of the first vendor-supported MPI implementations. MPI has replaced IBM's proprietary library MPL as the preferred message passing library on SP systems. The first optimized version of MPI available for SP systems, MPI-F, was a research prototype based on MPICH. The currently available implementation of MPI (hereafter referred to as IBM MPI) is rewritten from scratch [13].

IBM MPI runs on IBM SP systems and AIX workstation clusters in one of two modes. In User Space (US) mode, an MPI application has direct access to the SP high performance switch (if one exists). This provides the best performance, with the restriction that only one process may access the switch on each node. In IP mode, MPI processes communicate using IP — over the high performance switch if it exists, or over any other network if not. Latency (minimum message transfer time) in US mode is an order of magnitude lower than in IP mode.

IBM MPI is integrated with IBM's Parallel Environment (PE) and Parallel Operating Environment (POE), which are layered software packages that provide the "glue" allowing an SP (or cluster) to function as a single machine. Assuming PE and POE are installed correctly (no small feat) MPI works as an integrated piece of software: compilers `mpcc` and `mpxlf` compile C and Fortran MPI programs; `poe` launches parallel MPI programs; I/O is handled in a reasonable way; signals are propagated from the `poe` launcher to MPI processes; the debuggers `pdbx` and pedb understand parallel programs.

Usability. MPI is well-integrated with the PE/POE infrastructure. This infrastructure provides, among other things:

- Parallel job startup, including optional automatic space sharing of parallel applications.
- Signal propagation to all processes in a parallel application.
- Flexible handling of standard I/O: standard output may be labeled by processor number and/or ordered by processor number; standard input may be broadcast to all processes or sent to a single process.
- An integrated batch queuing system called Loadleveler.

Unfortunately there remain a number of usability problems related to PE/POE. Though not problems with the MPI implementation itself, these interfere with the usability of MPI. For example:

- Numerous user-settable options for poe do not have reasonable defaults.
- Signal propagation is not entirely foolproof, so that orphaned processes are not uncommon
- Parallel jobs are still second-class citizens. For example, there is no good way to see what parallel jobs are running on the system. The standard utility, jm_status, produces verbose output that is not easily parsed by humans.
- Despite improvement since earlier releases, the batch system Loadleveler requires substantial local customization and tools to be useful.

Tools. Two debuggers are available from IBM: pdbx is a command-line debugger built on dbx and pedb is a parallel debugger with an X interface. Both debuggers have reasonable though not outstanding interfaces. Neither understands message queues, so that finding out why a program is deadlocked can be difficult, for instance. A third debugger, Totalview, is available from Dolphin Interconnect Solutions[8]. This is probably the best MPI debugger available in IBM systems, but currently it understands only MPICH, not IBM MPI.

Message trace collection and visualization is integrated with IBM MPI. With a command line option or by setting an environment variable, MPI programs can automatically collect message trace information that can be displayed with a tool called vt.

6.2 HP

HP provides an implementation of MPI that runs on all current HP hardware, including the S-class and X-class Exemplar systems [14]. HP MPI was derived from MPICH, but also was significantly influenced by LAM.

HP MPI uses whatever communication medium it has access to: TCP/IP between hosts, shared memory within a host, and a hardware data mover for long messages on Exemplar systems. HP MPI is interoperable among all supported HP systems. HP MPI is well tuned on the high-end systems, with both very low latency and high bandwidth on Exemplar servers. It has also been optimized to use shared memory to implement collective operations where possible, rather than layering on top of point-to-point routines. HP MPI is compliant with MPI 1.2.

Usability HP MPI is not a part of a comprehensive parallel environment that manages parallel programs. This is not too bad on an Exemplar server, where there is a single system image, but can make is more difficult to run on a network of workstations.

HP provides several scripts for use with MPI. These include compiler scripts `mpicc/CC and mpif77/f90`, a program for listing currently running parallel jobs (`mpijob`), and a program for killing parallel jobs (`mpiclean`). Together, these provide a bit of an illusion of a parallel environment, but not quite as much as is needed. In particular, state is stored in the file system, rather than in daemons that can detect inconsistent state and react to it. There is no special handling of standard I/O for parallel jobs. HP MPI is also not integrated with a batch environment.

Tools HP MPI ships with `xmpi`, the trace visualization tool originally developed as part of LAM. `xmpi` is well-integrated with HP MPI. An additional flag to the `mpirun` command causes MPI programs to automatically generate trace files that can be displayed by `xmpi`. Furthermore, `xmpi` can run an MPI application interactively. For more information about `xmpi` see Section 5.1.

HP MPI is integrated with the debugger `cxdb`. The `cxdb` interface isn't great, but it's adequate for simple tasks. Unfortunately it isn't able to show message queues.

6.3 Sun

The Sun implementation of MPI is quite recent. Version 2 is in beta release as of this writing and should be generally available soon. Version 1 was a repackaged MPICH.

Sun MPI is derived from MPICH. For version 2, it has been integrated with a new Sun HPC environment and optimized for SCI, though it can run over any network using TCP. The Sun HPC environment is layered software that includes parallel job management. Users can launch (`tmrun`), examine (`tmps`) or kill (`tmkill`) parallel jobs. There is considerable flexibility in specifying where jobs are started, how standard input and output should be handled (approximately the same as the functionality in IBM MPI, plus a bit more), etc.

Because of the early status of this implementation, it is difficult to assess its strengths and weaknesses. There are two potentially exciting developments in the Sun implementation. The first of these is the Prism debugger. This debugger, originally developed by Thinking Machines Corporation, won nearly universal praise for its excellent user interface, and for its built-in performance and visualization features. In its new incarnation, Prism can debug HPF applications as well as MPI applications. The second exciting feature is the MPI-2 I/O library. This reviewer was not able to test this MPI-2 functionality, or the parallel file system associated with it, but it appears that much of the work has been done. Providing this functionality put Sun, which has historically lagged other vendors in MPI support, in a leading position with respect to MPI development.

On the other hand, robustness of the HPC package has not yet been demonstrated. A quick test of the beta version of the software revealed bugs: for instance, breakpoints sometimes weren't displayed correctly by Prism, and the software became confused about what parallel jobs were running. It does not appear that the process management software will be able to do a good job of process placement, and there is no good batch queuing system. There is no special scheduling of parallel jobs, such as gang scheduling or a mechanism to dedicate processors to a single application. Load Sharing Facility (LSF), which ships with Sun HPC software, is not integrated with Sun process management tools and has not yet shown itself to be effective at managing parallel jobs in any case. Also, since Sun MPI is derived from an earlier version of MPICH, the implementation is not compliant with respect to `MPI_Cancel` and Fortran constants, as described above.

6.4 Digital

Digital is another newcomer to the MPI world, having recently released a version for clusters of Alpha SMP servers connected by Digital's proprietary Memory Channel interconnect [15]. Digital MPI is quite close to the original MPICH, with special optimizations for communication over local shared memory and over the memory channel.

Digital's implementation of the MPICH ADI uses a lower level communication layer, UMP (Universal Message Passing), that provides low-level communication functionality over the Memory Channel and over shared memory. For long messages, UMP uses a background thread to allow overlap of communication and computation.

While Digital MPI is well-optimized for its hardware, there is not yet a robust cluster environment in which to embed it. For instance, there are no tools for the management of parallel jobs, I/O is no better than MPICH, jobs are not automatically distributed over an Alpha cluster. There is no special scheduling of parallel jobs, such as gang scheduling or a mechanism to dedicate processors to a single application. The best debugging option on such a cluster is currently to use MPICH with Totalview. Digital MPI also inherits the two standard bugs from an earlier version of MPICH — lack of support for `MPI_Cancel` and incorrect treatment of Fortran constants. On the other hand, this is a brand-new implementation, and the situation is likely to improve.

6.5 SGI

Now that Silicon Graphics, Inc (SGI) has bought Cray Research Inc. (CRI), SGI has three separate MPI implementations for its three types of machines — parallel vector (e.g. J90/C90/T90), Irix (including Origin 2000), and T3E. These implementations all have different roots and are therefore treated as separate implementations here. SGI is in the process of merging at least two of the implementations, In each case, MPI is part of a package called MPT (Message Passing Toolkit) that also includes SGI/Cray's `shmem` library and PVM.

6.5.1 SGI/PVP

PVP MPI (not a standard name) is derived from MPICH. It supports MPI applications within a single PVP (Parallel Vector Processor, such as the Cray J90, C90 and T90), using shared memory for communication, or spanning several PVPs (using TCP for communication). Because of the rarity of PVP clusters and the much slower speed of TCP communication, the rest of this discussion is about the shared memory version.

Shared memory PVP MPI is implemented in an interesting way, demonstrating the flexibility of the MPI process model. MPI processes are in fact threads within a single process. Through the use of special compiler options, all user-declared variables are local to a thread, so that separate threads do not directly "see" each other's data. Since all "processes" share the same address space, message transfers can be done with a single copy from source to destination (instead of using an intermediate buffer) and synchronization can be done using fast thread mechanisms. Applications using this process model are restricted to the SPMD model, where each MPI process is the same executable.

PVP MPI is fairly well-integrated into the environment when run on a single host, as far as process control goes — an MPI application looks just like any multithreaded application. Special scheduling (e.g. gang scheduling) is possible, but not well supported (and not at all supported on multiple nodes). Multiple host jobs (using TCP) suffer from all of the problems of MPICH with the `ch_p4` device. However, even for the shared memory version there are no special options for handling I/O, debuggers that understand MPI jobs, etc. Furthermore, PVP MPI suffers from the usual bugs associated with earlier versions of MPICH, including lack of `MPI_Cancel` and the problem with MPI-defined constants in Fortran.

6.5.2 SGI/T3E

T3E MPI is derived from the T3D implementation developed at the Edinburgh Parallel Computing Centre. The T3D version was in turn derived from the Chimp implementation. Though the T3D version allegedly suffered from performance and robustness problems, these seem to have been fixed in the T3E implementation.

T3E MPI is robust, and well-integrated with the environment. Parallel jobs are understood by the operating system as distinct entities and are managed directly by the operating system, rather than by layered software. Many standard tools (e.g. `ps`, accounting) understand parallel applications. T3E MPI is generally quite easy to use.

On the performance front, an interesting feature is that MPI is able to take advantage of special hardware on the T3E for sending strided arrays. This is discussed in more detail in Section 7.4. On the other hand, the T3E copies non-aligned data slowly, so make sure to use buffers that are 8-byte aligned — this is automatic for the usual case of sending double precision data.

There are a few minor but longstanding problems. For instance, tools to show what parallel applications are running are primitive; there is no flexibility in how standard I/O is handled.

On the tools front, the Totalview debugger is available for debugging parallel programs. This is not the debugger from Dolphin Interconnect Solutions, but is a Cray product with a common ancestor. Cray Totalview has fallen behind its counterpart in ease of use and functionality, but is still useful. In particular, Cray Totalview cannot display message queues. There are no tools to extract or view message traces.

6.5.3 SGI/Irix

SGI recently released version 3.0 of its "Array" software for clustering. This package includes an implementation of MPI that runs on all current SGI MIPS-based systems. SGI MPI is originally derived from MPICH, but has evolved considerably. It has also incorporated `xmpi` from LAM.

SGI MPI is optimized for shared memory inside SMP servers and for a special HIPPI "bypass" that provides low-latency communication over HIPPI and striping over multiple HIPPI connections for large messages. It also uses TCP if HIPPI isn't available or the bypass is disabled. SGI MPI is interoperable among different SGI systems as long as all parts of an MPI application are compiled for 32-bit or 64-bit mode.

In most cases, MPI applications are run inside a single Origin 2000 system. SGI's array services software provides infrastructure to allow running on a cluster of systems. Applications running on this cluster are identified by an array session handle (ash). There are array equivalents for `ps` and `kill` that allow array sessions to be treated as a single unit. Array software is required even when running on a single node. Array software must be installed and maintained by a system administrator.

SGI MPI is compliant with MPI 1.2.

Usability SGI MPI is most usable on a single host, such as a large Origin 2000, and is slightly less usable in clusters. Because of the array services software, MPI applications are managed as a single unit, even when spread across multiple nodes. Starting an application on a single host is easy, but the syntax for starting on multiple hosts is somewhat painful, and isn't managed directly by the array services software.

Standard I/O is fine when using a single host but is currently not handled well when using multiple hosts. Output from processes on remote nodes is lost and there is no optional labeling of output lines by process number. These will be addressed in a forthcoming release.

Tools SGI MPI ships with `xmpi`, the trace visualization tool originally tool originally developed as part of LAM. `xmpi` can be used to start MPI programs and collect trace information. There is no mechanism to generate trace information or examine trace files without `xmpi`.

There is no debugger that understands MPI applications.

6.6 NEC SX-4

NEC MPI is another new implementation. NEC has experimented with several very different implementations. The one described here is just becoming available on the SX-4 as of this writing and should be standard on the SX-4 in the near future [16].

NEC MPI is a recent descendant of MPICH, starting from the `ch_lfshmem` device, which was originally implemented for the SX-4. NEC MPI has been highly optimized for both a single-node SX-4, where MPI uses shared memory for communication, and a multi-node SX-4, where communication between nodes is done through the Internode Crossbar Switch (IXS).

NEC MPI is integrated with the VAMPIR and VAMPIRtrace tools from Pallas, which allows users to visualization message trace information to optimize programs [17].

Other features of the NEC implementation are at this time limited to what is available in MPICH. Because of its recent release, I have not had an opportunity to assess its usability.

6.7 Others

This section briefly mentions several other MPI implementations that are available.

Mercury Race Hughes Aircraft Co. has implemented MPI for Mercury RACE systems [18]. RACE MPI is derived from MPICH. There are a few interesting features of this implementation that are worth noting.

- On SHARC systems, where a "byte" (defined by ANSI C to be the size of a `char`) is 32 bits, not 8 bits, this implementation exposed a portability problem for MPI codes.
- The MPICH library has been modified to conserve as much space as possible. Only needed routines are linked, argument checking and strings are omitted.
- Several collective operations have been optimized to use shared memory.

Hitachi Hitachi provides an implementation of MPI based on MPICH for its SR2201 series computers. This implementation uses the SR2201's remote DMA facility. See: `http://www.hitachi.co.jp/Prod/comp/hpc/eng/sr1.html`.

NEC Cenju-3 NEC provides an MPICH device for its Cenju-3 computers. See: `http://www.ccrl-nece.technopark.gmd.de/ mpich/mpich_cenju3.html`.

Alpha Data Alpha Data provides an implementation of MPI for its AD66 systems. This implementation was developed jointly with the the Edinburgh Parallel Computing Centre and is presumably related to CHIMP.
See `http://www.alphadata.co.uk/softhome.htm`.

Fujitsu MPI for the Fujitsu VPP machines has recently been developed by Pallas (`http://www.pallas.de`), for Fujitsu. This implementation is reported to be in final testing, but no further information is available. MPI for the Fujitsu AP1000 is available from Australian National University [19].
See `http://cap.anu.edu.au/cap/projects/mpi/mpi.html` for more information.

PVMPI PVMPI is a research project based at Oak Ridge National Laboratory and the University of Tennessee at Knoxville whose goal is to allow MPI implementations from different vendors to interoperate[20]. It is advertised that the application programmer uses MPI for communication between all processes, plus a few PVMPI routines for establishing communication between MPI implementations — PVMPI was designed before MPI-2 was finalized. Internally, PVMPI uses PVM for communication between processes tied to the different implementations, and native MPI (through the MPI profiling interface) for communication among processes associated with a single MPI implementation. As of this writing, the software is not yet available, and it is not clear how many MPI details can be relied on. Based on available information, it seems that PVMPI will be a good starting point for a full meta-MPI implementation, and a good way for applications that need interoperability now and cannot wait for such an implementation, but that PVMPI will not provide full MPI interoperability, and will be missing many MPI features. For more information, see `http://www.netlib.org/mpi/pvmpi/pvmpi.html`.

7 Portability Issues

While the MPI standard enables portable parallel programs, it does not *guarantee* portable programs, in the following senses:

- MPI applications may take advantage of implementation details of one MPI implementation that are different in another implementation.

- MPI applications may correctly use MPI features that perform poorly in some MPI implementations.

This section discusses a number of potential pitfalls.

7.1 Buffering

One of the most common mistakes made by MPI users is to assume that an MPI implementation provides some amount of message buffering. To buffer a message is to make a temporary copy of a message between its source and destination buffers. This allows `MPI_Send` to return control to the caller before a corresponding `MPI_Recv` has been called. In this way, it is possible for two processes to exchange data by first calling `MPI_Send` and then calling `MPI_Recv`.

MPI does not require any message buffering, and portable applications must not rely on it. To avoid deadlock conditions, applications should use the non-blocking routines `MPI_Isend` and/or `MPI_Irecv`. It is common for programs ported from PVM or applications that use message "portability" packages to make assumptions about buffering.

The implementations described in this review vary greatly in the amount of buffering they provide, and some allow the user to control the amount of buffering through environment variables. For instance, the T3E implementation currently buffers messages of arbitrary size, by default, while The SGI implementation may buffer as few as 64 bytes, depending on environment variables.

The amount of buffering is deliberately left open by the MPI standard so that implementors can make platform-specific optimizations. Reasons to provide a large amount of buffering include reducing application synchronization and improving small message performance. Reasons not to provide a large amount of buffering include improving large message performance, reducing memory management complexity and overhead and reducing MPI internal memory use.

7.2 Non-standard Send Modes

MPI provides four send modes: standard, ready, synchronous and buffered. In almost all cases, standard mode is the right one to use for a portable application. For certain implementations, ready or synchronous mode perform better than standard mode for some message sizes, but the difference is small and the range of message sizes varies for different implementations and different release numbers. Furthermore, these modes often have unexpectedly poor behavior — for instance, ready mode is slower than standard mode for small messages in several implementations. Buffered mode is also tempting to use, especially for users porting codes from systems that provide a lot of buffering. But it is quite often the wrong thing. It is usually unnecessary (non-blocking communication can be used instead); it is usually slow; it can be difficult to use correctly.

7.3 Standard I/O

MPI does not require that standard I/O work as expected from MPI applications. For instance, data written to `stdout` in a C program may be heavily buffered or may be lost entirely. Implementations that use `rsh` for starting remote processes and handling remote standard I/O typically have problems with buffering, so that output may not appear until long after it is written. At least one implementation (SGI) currently does not handle output from remote nodes, though that will be changed in the next release. Many implementations do not provide line buffering, so that output from several processors can become intermixed and garbled.

An MPI application can inquire about regular I/O facilities by looking at the `MPI_IO` attribute. Technically, this allows one to write portable programs that use standard I/O, but it is painful to use and not helpful if the answer is "no."

Because of the importance of standard output for MPI usability (especially for debugging), it is the opinion of this reviewer that when standard output is normally available (i.e., for serial programs) high quality MPI implementations should provide standard output, without line garbling, and with little delay between when the output is written and when it appears on the terminal. Furthermore, it is not unreasonable for applications to rely on this ability, though they should not expect high performance.

Input is a different issue. When a parallel application reads from standard input, to which process should the input go? Some implementations direct standard input to process 0 in `MPI_COMM_WORLD`. This is what users often expect, though not all implementations allow it to work. Some implementations provide the optional ability to broadcast the input to all processes, with the requirement that all processes read the same data. When possible, applications should avoid reading from standard input, choosing instead to open files. When reading from a file is necessary, the most portable option is to read it only on process 0 of `MPI_COMM_WORLD`, but do not expect this to always work.

7.4 Performance Considerations with User-Defined Datatypes

One of the most interesting features of MPI is the ability for applications to define MPI datatypes. MPI datatypes can describe almost any C or Fortran data object except C structures with pointers (there is no easy way to automatically "follow" the pointer).

User-defined datatypes are part of MPI for two basic reasons: they allow automatic data conversion in heterogeneous environments and they allow certain performance optimizations. The PVM Pack/Unpack approach allows automatic conversion in heterogeneous environments while the MPI approach additionally allows optimizations such as using special hardware or a coprocessor to perform scatter/gather, or pipelining scatter/gather with message transfer.

Unfortunately, while this is a nice idea in principle, most implementations do not implement these optimizations and using certain MPI datatypes can dramatically slow down communication performance. The two important cases are:

- Most implementations implement sending and receiving of non-contiguous data very inefficiently.

- Some implementations do not correctly detect that a nonuniform but contiguous datatype is in fact contiguous in memory.

In both these cases, the implementation resorts to very slow copying of the send/receive buffer into/out of an internal MPI buffer, resulting in greatly reduced bandwidth — as much as two orders of magnitude slower than for simple messages. In these cases, it is often more efficient for the user to explicitly pack and unpack the data to and from user-managed buffers (sending as an array of `MPI_BYTE` or some other simple datatype) than to let MPI do it automatically.

This is not the "MPI way" of doing things and may not always be the most efficient. For the foreseeable future, compiled user code will usually be able to pack data faster than an MPI library (but not specialized hardware). The wildcard is multithreaded MPI implementations where a thread running on another processor can pack data overlapped with computation.

One exception to the above rule is sending strided arrays on the T3E, where specialized hardware can send strided arrays faster than an MPI program can pack them. However, the hardware kicks in only if you use `MPI_TYPE_VECTOR`, not an contiguous array of building blocks that contain data followed by "holes."

I therefore reluctantly suggest that MPI applications implement two different methods for sending non-contiguous data. One should use the "MPI way" with non-contiguous datatypes, the other should pack into a user-managed buffer, and the choice should be made at compile time or runtime. Unfortunately this is not an elegant solution, but it is the best available at this time.

7.5 Thread-safe MPI

One of the promises of the MPI standard is that it is possible to implement MPI so that it is thread-safe. This does not mean that MPI is automatically thread-safe, but that an implementor can make it thread-safe. Despite this potential, there have been until recently no thread-safe commercial implementations of MPI. Both Sun and IBM plan to release thread-safe versions of MPI in the late 1997 time frame.

In both implementations, users will specify at link time which version (regular or thread-safe) of the library to use. Applications using the thread-safe version will be able to call MPI routines concurrently from separate threads. A blocked MPI call in one thread will not obstruct MPI operations in other threads. However, in the Sun implementation, the user must make sure there are enough lightweight processes (LWPs). It is not clear how the user should determine the correct number, and it is arguable whether this is the right behavior. In the end, whether these implementations are effective (i.e., solve user problems) will be determined by details of thread scheduling and MPI polling policies.

In the IBM MPI implementation, an MPI process that is single-threaded from the user's point of view will use multiple threads under the covers, allowing greater concurrency (e.g. overlap of communication and computation). These new features will be useful for multithreaded applications or applications running on SMPs. Single-threaded applications on uniprocessors will usually get better performance from the single-threaded MPI.

In the Sun implementation, the implementation itself is not multithreaded. All single-threaded programs will run more efficiently with the non-threadsafe MPI.

In both implementations, the behavior of the multithreaded MPI is consistent with behavior specified in the MPI-2 standard, though extra MPI-2 calls to enable run-time requests are not provided.

8 MPI-2

The MPI-2 standard was finalized in July, 1997 [2]. MPI-2 provides substantial new functionality, including support for dynamic process management, one-sided communication, cooperative I/O, extended collective operations and numerous other features. Most vendors have said they will implement part or all of MPI-2, but have given no time frames.

The MPI-2 specification for thread-safe MPI has been mostly implemented by Sun and IBM as described in Section 7.5. These implementations do not provide the extra MPI-2 calls to enable run-time requests.

MPI-2 I/O is the first major piece of MPI-2 that is likely to be widely available. There are at least two research projects, ROMIO and PMPIO, that are implementing portable versions of MPI-2 I/O [21, 22]. Of these, the ROMIO implementation is the first out of the gate, with enough of MPI-2 I/O implemented for applications to start using it. ROMIO works with MPICH, SGI MPI and HP MPI, and can use a Unix filesystem, NFS filesystem, or IBM's PIOFS. PMPIO has been around longer, but implements an older version of the I/O specification. PMPIO is also tied to MPICH, though it runs on top of several different filesystems. There does not appear to have been much recent activity in PMPIO development.

Sun is the first vendor to demonstrate an implementation of MPI-2 I/O. This reviewer has not yet used it, but a beta version scheduled for release later this year appears to be mostly complete. It uses a "parallel file system" called PFS that is currently accessible only through MPI-2 I/O. The usability and performance of this implementation remain to be demonstrated.

Both SGI MPI and HP MPI implement MPI-2 routines necessary for layering MPI-2 I/O as a third-party library. This is how ROMIO is able to interact with these implementations.

The LAM implementation provides MPI-2 dynamic process management, As of this writing the function names are not the final versions (e.g. `MPI_SPAWN` instead of `MPI_COMM_SPAWN`), but this is a small problem.

The prospects for dynamic process management in other implementations and one-sided communication in any other implementation are less clear. These are much more difficult to implement, and are impossible to layer on top of existing implementations.

9 Conclusions

While the discussions of in the last several sections have hopefully illuminated and clarified the landscape of MPI implementations, the overall picture had been a bit unbalanced, because of the natural tendency to focus on what doesn't work, rather than what does. The primary message you should carry away is still that there are many MPI implementations out there, that they are well-supported by vendors, and that despite the occasional bug or missing feature, they work quite well. For instance, while I have mentioned that several implementations are not MPI 1.1 compliant with respect to the definition of certain Fortran constants, the overwhelming majority of the MPI standard has been correctly implemented by all implementations.

The area in which there remains the most room for improvement is integration of MPI applications into a robust environment in which parallel applications are first-class citizens, understood directly and handled appropriately by the operating system, debuggers, analysis tools, and user interface.

Also high on the MPI wish list is a freely available, robust and highly usable MPI for networks of workstations (NOWs). While MPICH (through P4) and LAM fill some of this need, they have some usability problems as discussed above. What is needed is something with the process and virtual machine management of PVM, but with the MPI message passing API. This type of software is needed both at the user level (so that users can put together personal NOWs) and at the system level (so that administrators can build highly integrated MPP-like systems).

10 Acknowledgements

I would like to acknowledge the following people and organizations who have have provided information about MPI implementations or access to MPI implementations: The National Center for Supercomputing Applications (John Townes) provided access to SGI and HP machines, NASA Ames Research Center (Mary Hultquist) provided access to an SGI machine, Lawrence Livermore National Laboratory (John May and Greg Tomaschke) provided access to a Digital Alpha cluster, Argonne National Laboratory (Rusty Lusk) provided access to an IBM SP, Sun Microsystems (Steve Walter) provided access to a beta version of Sun MPI, Rolf Rabenseifner of the University of Stuttgart and Rolf Hempel of NEC provided information about the NEC implementation, Lloyd Llewins of Hughes Aircraft Company provided information about the Mercury RACE implementation, Raja Daoud of HP provided information about the HP and LAM implementations, and Eric Salo of SGI provided information about the SGI implementation.

This work was supported by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

References

[1] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications*, 8, 1994. Special issue on MPI.

[2] Message Passing Interface Forum.
`http://www.mpi-forum.org`. All official documents of the MPI Forum are available here.

[3] Al Geist, Adam Begeulin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. See also
`http://www.epm.ornl.gov/pvm/`.

[4] IBM Corporation. Message Passing Interface Test Case Suite.
`ftp://info.mcs.anl.gov/pub/mpi/mpi-test/ibmtsuite.tar`.

[5] Intel Corporation. MPI V1.1 Validation Suite.
`http://www.ssd.intel.com/mpi.html`.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996. See:
`http://www.mcs.anl.gov/mpich/`.

[7] Aad J. Van der Steen and Jack J. Dongarra. Overview of recent supercomputers. *NHSE Review*, 1, 1996.
`http://nhse.cs.rice.edu/NHSEreview/96-1.html`.

[8] Dophin Interconnect Solutions. Totalview Multiprocess Debugger.
`http://www.dolphinics.com/tw`.

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

[10] Ian Foster, Carl Kesselman, and Steve Tuecke. The nexus task-parallel runtime system. In *Proceedings of the First International Workshop on Parallel Processing*, 1994. See also
`http://www.mcs.anl.gov/nexus/` and
`http://www.mcs.anl.gov/mpi-nexus/index.html`.

[11] Ohio Supercomputer Center. *MPI Primer/Developing with LAM*.
`http://www.osc.edu/lam.html`.

[12] R. Alasdair, A. Bruce, James. G. Mills, and A. Gordon Smith. *CHIMP/MPI User Guide*.

[13] IBM. *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference Version 2, Release 2, Document Number GC23-3894-01*, November 1996.
http://www.rs6000.ibm.com/resource/aix_resource/sp_books/pe/index.html.

[14] Hewlett Packard. *HP MPI User's Guide, HP Part No. B6011-90001.*, November 1996.
http://www.hp.com/wsg/ssa/mpi/mpihome.html.

[15] Digital Equipment Corporation. *Digital MPI User Guide*, February 1997.
http://www.digital.com/hpc/software/dmpi.html.

[16] R. Hempel, H. Ritzdorf, and F. Zimmermann. Implementation of mpi on nec's sx-4 multi-node architecture. In *Proceedings of the Euro PVM-MPI Workshop*, 1997.
http://www.ccrl-nece.technopark.gmd.de/ mpich/mpich_nec.html.

[17] Pallas. Vampir and vampirtrace.
http://www.pallas.de.

[18] Lloyd J. Lewins. Mpi for the mercury race processor. For more information, mail to *llewins@ccgate.hac.com*.

[19] David Sitsky. Implementing mpi using interrupts and remote copying on the ap1000/ap1000+. In *Proceedings of the Fourth Parallel Computing Workshop*, London, England, October 1995.
http://cap.anu.edu.au/cap/projects/mpi/mpi.html.

[20] J. Dongarra G. Fagg and A. Geist. Heterogeneous mpi application interoperation and process management under pvmpi. Technical Report CS-97-???, University of Tennessee Computer Science Department, June 1997.

[21] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of The 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[22] Sam Fineberg, Bill Nitzberg, Ian Stockdale, and Parkson Wong. Pmpio - a portable mpi-io library.
http://parallel.nas.nasa.gov/MPI-IO/pmpio/pmpio.html.