Implementation of MPICH on top of MP_Lite

by

Shoba Selvarajan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Daniel Berleant (Major Professor)
Ricky Kendall
Srinivas Aluru
Dave Turner

Iowa State University

Ames, Iowa

2002

Graduate College

Iowa State University

This is to certify that the master's thesis of

Shoba Selvarajan

has met the thesis requirements of Iowa State University

Daniel Berleant

Major Professor

Vijay Vittal

For the Major Program

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Dr. Dave Turner, my supervisor, for enthusiastically guiding and helping me during the course of this research. He has been a constant source of motivation, inspiration and knowledge, all along my way.

My special thanks are due to Dr. Berleant, my major advisor, for his continuous support and guidance throughout my graduate studies.

I gratefully thank Dr. Ricky Kendall and Dr. Srinivas Aluru for serving as my committee members, and for reviewing my thesis manuscript.

I would like to thank Dr. Bill Gropp, Argonne National Laboratory, for clarifying my doubts regarding the MPICH implementation. I deeply acknowledge his help and suggestions.

I thank my parents, without whom I would not have completed my graduate studies successfully. I am forever indebted to their moral support and motivation.

ABSTRACT

The goal of this thesis is to develop a new Channel Interface device for the MPICH implementation of the MPI (Message Passing Interface) standard using MP_Lite. MP_Lite is a lightweight message-passing library that is not a full MPI implementation, but offers high performance. MPICH (Message Passing Interface CHameleon) is a full implementation of the MPI standard that has the p4 library as the underlying communication device for TCP/IP networks. By integrating MP_Lite as a Channel Interface device in MPICH, a parallel programmer can utilize the full MPI implementation of MPICH as well as the high bandwidth offered by MP_Lite.

There are several layers in the MPICH library where one can tie a new device. The Channel Interface is the lowest layer that requires very few functions to add a new device. By attaching MP_Lite to MPICH at the lowest level, the Channel Interface, almost all of the performance of the MP_Lite library can be delivered to the applications using MPICH. MP_Lite can be implemented either as a blocking or a non-blocking Channel Interface device.

The performance was measured on two separate test clusters, the PC and the Alpha mini-clusters, having Gigabit Ethernet connections. The PC cluster has two 1.8 GHz Pentium 4 PCs and the Alpha cluster has two 500 MHz Compaq DS20 workstations. Different network interface cards like Netgear, TrendNet and SysKonnect Gigabit Ethernet cards were used for the measurements.

Both the blocking and non-blocking MPICH-MP_Lite Channel Interface devices perform close to raw TCP, whereas a performance loss of 25-30% is seen in the MPICH-p4 Channel Interface device for larger messages. The superior performance offered by the MPICH-MP_Lite device compared to the MPICH-p4 device can be easily seen on the SysKonnect cards using jumbo frames. The throughput curve also improves considerably by increasing the Eager/Rendezvous threshold.

CHAPTER 1

INTRODUCTION TO MESSAGE PASSING

1.1 The Message-Passing Model

Message passing is one of many parallel-programming paradigms that are used for parallelizing computational intensive applications. In the message-passing model, the application is split into a number of programs that operate independently, usually on different processors. The processors have their private local memories and are linked to one another by means of a communication network. Each processor executes its own copy of the code and interacts with other processors by exchanging messages. It is the programmer's responsibility to preserve the underlying logic that controls the working of the application. The programmer has to partition the data among the processors and explicitly specify any interaction among them.

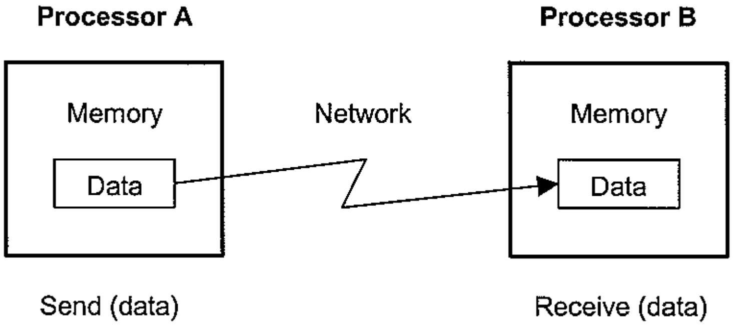

Figure 1.1. Message passing.

The message-passing model can be defined as:

- A set of processes having only local memory.
- Processes communicate by sending and receiving messages.
- Data transfer between processes requires cooperative operations to be performed by each process (a send operation must have a matching receive).

All message-passing libraries provide a key set of facilities for the application developer:

- The ability to create and terminate processes on remote machines.
- The ability to monitor the state of those processes.
- Routines that enable programs to send messages, or signals, to other programs.
- Routines to do collective/group communications and synchronization.

1.2 Message Passing Terminologies

Blocking communication

A communication routine is blocking if it returns only after the action is complete. For a send operation, the routine must block until the data is successfully sent or safely copied so that the buffer that contained the data is available for reuse. In the case of a receive, the routine must block till the data is at its final destination so the application can use it.

Non-blocking communication

A non-blocking communication routine returns without waiting for any communications events to complete (such as copying of message from user memory to system memory or arrival of the message). The communication between the two processes may also overlap with computation.

Synchronous communication

In synchronous communications, both the source and the destination nodes are blocking. The source node has to delay sending the message until the destination node posts a matching receive, and has started to receive the message. An exchange of a message represents a synchronization point between the two processes.

Asynchronous communication

In asynchronous communications, the source node is non-blocking; it initiates the send and returns immediately without waiting for the destination node to receive the message. The source and destination nodes function independently and place no constraints on each other in terms of completion.

1.3 Pros and Cons of the Message-Passing Model

1.3.1 Advantages

The message-passing model is extensively used in the field of parallel computing because of the following advantages.

- Universality: It matches the hardware of most of today's parallel supercomputers, cluster of workstations (separate processors connected by a communication network) and shared-memory multi-processors.
- Performance: Memory and bandwidth are scalable to the number of processors. The programmer has more control over the locality of memory accesses. Performance depends on the programmer's ability to write efficient parallel code.
- Functionality: It has a full set of functions that offers complete control over data movement, which helps the programmer to express most of the parallel algorithms.
- Debugging: It is easier to debug than other parallel programming models because the programmer has explicit access to the memory.

1.3.2 Limitations

Some of the limitations of the message-passing model are presented below.

- Hard to program: The programmer must identify all the parallel regions in the code and divide the work efficiently among different processors. The programmer must explicitly implement a data distribution scheme and all interprocess communication and is also responsible to resolve data dependencies, and avoid deadlock and race conditions.

- Significant communication overhead is introduced for small transactions. In order to minimize overhead and latency, data may be accumulated in large chunks and delivered before the destination node needs it.
- No portability path from serial systems.
- Difficult to design modules for reusability.

1.4 Alternatives to the Message-Passing Model

1.4.1 One-sided libraries

In the traditional message-passing model, communication is two-sided; both the source and destination nodes must co-operate. There are also libraries [1,2] that support one-sided communication operations such as "*get*" and "*put*" functions. One-sided communication assumes that a process can access data on a remote node asynchronously, without explicit cooperation of the process on the remote node. The latency and overhead costs of the one-sided operations are comparable to those of the standard send and receive operations.

This model is better than the message-passing model in the sense that no coordination is required on both sides for each data transfer. However, it is slightly difficult to use since it often requires the programmer to understand and manage data locality, and to perform manual handshaking (which is automatic in two-sided communications).

The one-sided communication model is useful when parallel programs need to make unpredictable references to remote data. It is particularly useful for applications that use dynamic load balancing and have wide variation in task size.

The SHMEM (Shared Memory Access) library developed by Cray is a one-sided library for the Cray T3E and SGI Origin systems. The SHMEM calls have significantly lower startup latencies and higher bandwidths. However, it is not portable to other computer systems. The GPSHMEM (Generalized Portable SHMEM) library [3] is a general purpose SHMEM library that attempts to achieve full portability. It provides the same one-sided interface but is implemented on top of lower level libraries.

1.4.2 Global arrays

The message-passing programming model is widely used because of its portability. But, in some applications, coding becomes complex when the programmer tries to maintain a balanced computation load and avoid redundant computations. The shared-memory programming model simplifies coding, but it is not portable and often provides little control over inter-processor data transfer costs. Global Arrays (GA) [4,5] combines the better features of the message-passing and shared-memory models, leading to a tradeoff between ease of programming and loss of efficiency.

Global Arrays allows for simple coding and efficient execution for a class of applications that appears to be fairly common. It provides a portable interface through which each process in a MIMD (Multiple Instruction Multiple Data) parallel program can independently, asynchronously, and efficiently access logical blocks of physically distributed matrices, with no need for explicit cooperation by other processes. In this respect, it is similar to the shared-memory programming model. However, the GA model also acknowledges that access to a remote portion of the shared data is slower than to the local portion. The locality information for the shared data is available and a direct access to the local portions of shared data is provided. In these respects, it is similar to message passing.

The programmer is free to use both the shared-memory and message-passing paradigms in the same program, and to take advantage of existing message-passing software libraries.

1.4.3 Threads model

In the threads model, a single process can have multiple flows of control called “threads”. The threads run concurrently in the context of the process that invoked them, sharing its code, data, open files, I/O channels and all other resources, and communicating with each other through global memory. This requires synchronization constructs like semaphores and locks to ensure that not more than one thread is accessing the same resource at any time.

There are two different implementations of threads, namely, Pthreads (POSIX threads) and OpenMP [6,7]. Pthreads is a POSIX.1c standard established to control the spawning, execution, and termination of multiple threads within a single process. It is a system-level standard for controlling shared-memory. The Pthreads standard is not targeted towards HPC (High Performance Computing) end-users since there is minimal Fortran support. Even under C, it is difficult to use for scientific applications, as it is aimed more at task parallelism than at data parallelism.

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran and C/C++ programs. It is often implemented as a high level interface to Pthreads. OpenMP provides robust support for loop-level parallelism by spawning threads of execution for loop iterations, and is also designed to give the programmer fine control over variable scope, thread scheduling, and thread synchronization.

OpenMP uses the fork-join model of execution. The program begins execution as a single process, called the master thread of execution. The master thread executes sequentially until the first parallel region construct is encountered. When it enters a parallel region, it forks a team of threads (one of them being the master thread). The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads. Upon exiting the parallel construct, the threads in the team synchronize (join the master) and terminate leaving only the master thread.

OpenMP is a parallel programming model that is typically used for SMP systems. SMP (Symmetric Multi-processing) systems contain several CPUs in a single computer, each of which has access to the same set of memory chips, with each working as a general-purpose CPU that can execute any process in the system.

CHAPTER 2

HISTORY OF MPI

2.1 Development of the MPI Standard

The initial work on the MPI (Message Passing Interface) standard [8,9] started in 1992 at Oak Ridge National Laboratory and at Rice University. It was focused on developing an efficient message-passing library and application software that could be ported to a wide array of high performance multi-computers. Several message-passing libraries that were being used at that time did not have a common syntax, and hence were not portable. To ensure portability and to enhance the features of the existing libraries, a standard was strongly desired which would provide hardware vendors with a well-defined set of routines that could be efficiently implemented.

In April 1992, the Workshop on Standards for Message Passing in Distributed Memory Environment was sponsored by the Center for Research on Parallel Computation (CRPC). At this workshop, the essential features of the message-passing standard were discussed, and a working committee called the "MPI Forum" was established to continue the standardization process. In November 1992, four of the members of the MPI Forum produced a preliminary draft proposal, known as MPI 1.0, which presented the essential features necessary to the MPI standard. In November 1992, the MPI Forum met again at the Supercomputing conference held in Minneapolis. It formed subcommittees that would concentrate on different areas of the standard, and also decided to produce a draft of the MPI standard during the following year. In November 1993, the four attendees presented the MPI standard draft, and in May 1994 the MPI 1.0 industry standard was finally released.

MPI 1.0 primarily focused on point-to-point communications; it did not include any collective communication routines and was not thread-safe. Since then, the MPI standard had undergone various revisions - 1.1 (June, 1995) and 1.2 (July, 1997), which corrected errors and minor omissions. Even though MPI had become a widely accepted standard for message passing, it lacked a number of features. To address these concerns, the MPI Forum developed the MPI-2 version, which was released in July 1997. Major topics like dynamic processes, one-side operations, I/O, C++ language bindings, collective operations, and threads were covered by the MPI-2 standard.

The design of MPI was strongly influenced by the research work at the IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, and PARMACS. Message-passing libraries like Zipcode, Chimp, PVM, Chameleon, and PICL also contributed to the development of the MPI standard. The MPI standardization effort involved about sixty people representing forty different organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers, researchers from universities, government laboratories, and industry were also involved.

2.2 Goals of the MPI Forum

The principal goal of the Message Passing Interface is to develop a standard that can be widely used to write efficient and portable message-passing programs. The following is a complete list of goals:

- Design a portable Application Programming Interface (API).
- Allow efficient communication; avoid memory-to-memory copying, allow computation-communication overlapping and offload to a communication co-processor, if available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for the interface.
- Assume a reliable communication interface; the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that is not too different from current practice.
- Define an interface that can be implemented on many vendors' platforms, with no significant changes in the underlying communication and system software.
- The semantics of the interface should be language independent.
- The interface should allow multiple threads of execution to exist within a process.

2.3 History of MPICH

At the Supercomputing Conference held in November 1992, when the preliminary draft proposal for MPI 1.0 was presented, William Gropp and Ewing Lusk volunteered to develop an immediate implementation of the MPI standard. The goal was to point out any problems in the specification that might arise during implementation and to experiment with new ideas. Gropp and Lusk, at the Argonne National Laboratory, designed and developed the first version of MPICH (MPI CHameleon) [10,11,12,13] that implemented the pre-specification within a few days. It was mostly developed using the existing portable systems p4 and Chameleon. This implementation was gradually modified to provide increased performance and portability. At the same time it was greatly expanded to include all of the MPI specification. It borrowed algorithms for the collective operations, topologies and attribute management from Zipcode.

When the MPI standard 1.0 was released in May 1994, the MPICH implementation was complete, portable, fast, and available immediately. With the MPI standard almost stable, MPICH continued to evolve in several directions. First, the Abstract Device Interface (ADI) [14,15] architecture was developed and stabilized. The ADI layer provides basic, point-to-point message-passing services. Second, individual vendors and others took advantage of ADI to develop their own highly specialized implementations of MPICH. This resulted in extremely efficient implementations of MPI on a greater variety of machines. Third, the set of tools that form part of the MPICH parallel programming environment was extended.

2.4 Precursor Systems of MPICH

MPICH was available immediately because it made use of the stable code from existing systems. Although most of that original code was altered, MPICH still owes some of its design to those precursor systems:

- P4, a third-generation parallel programming library that includes both message-passing and shared-memory components. P4 still remains one of the “Channel Interface devices” on which MPICH can be built.
- Chameleon, a high-performance portability package for message passing on parallel supercomputers. A substantial amount of Chameleon technology is incorporated into MPICH.
- Zipcode, a portable system for writing scalable libraries. Several concepts including contexts, groups, and communicators of Zipcode were included in the design of the MPI standard.

CHAPTER 3

SUMMARY OF MPI AND OTHER IMPLEMENTATIONS

This chapter presents a brief overview of the various public domain, commercial and vendor versions of MPI and other implementations [16]. It also discusses some of the ADI and Channel Interface implementations of MPICH.

3.1 MPICH

MPICH (Message Passing Interface CHameleon) is the most important MPI implementation. It is a freely available software developed at Argonne National Laboratory and Mississippi State University. MPICH is the parent of a large number of commercial implementations of MPI including vendor-supported implementations from Digital, Sun, HP, SGI/Cray, NEC and Fujitsu. Most of the experimental and research versions of MPI were also based on MPICH.

The design of MPICH was guided by two principles; to maximize the amount of code that can be shared without compromising performance and to provide a structure whereby MPICH could be ported to a new platform quickly. Performance and portability were the two main goals in proposing the architecture of MPICH. The advantages of MPICH include portability, language bindings for C, Fortran and C++, high performance, heterogeneity and interoperability.

MPICH is highly portable because of its layered design. It has a 4-layered architecture. The top two layers contain the bindings for the MPI functions. The lowest layer is called the Channel Interface layer [17] and the one above it is called the ADI (Abstract Device Interface) layer. 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 Channel Interface layer just transfers data from the address space of one process to that of the other.

3.2 LAM/MPI

The LAM (Local Area Multicomputer) [18] implementation of MPI is a freely available and portable implementation that was originally developed at the Ohio Supercomputer Center, and is now being developed at Indiana University by Dr. Andrew Lumsdaine. LAM existed before MPI and was adapted to implement the MPI interface. LAM runs on many platforms, including RS6000, Irix, Linux, 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 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 for higher performance.

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

3.3 MP_Lite

MP_Lite [19,20] is a lightweight message-passing library that implements an efficient subset of MPI commands. It is mainly a research tool, being developed in Ames Laboratory, to study and improve the performance of the message-passing layer. It delivers the maximum performance of the underlying network layer to the applications by avoiding extra buffering and memory-to-memory copies, and allowing overlapped computation and communication. MP_Lite can run on top of TCP on workstation clusters, on the SHMEM library on Cray T3E and SGI machines and on VIA module [21].

The user can run MP_Lite under two modes of operations, namely the synchronous and the SIGIO (interrupt driven) modes. The synchronous mode is a thin layer over the TCP/IP sockets interface. It makes use of the TCP send and receive buffers and avoids buffering at any cost. The SIGIO mode operates based on interrupts. When the TCP buffers receive or empty data, SIGIO interrupts are generated. The signal_handling routine services all active socket buffers to maintain constant message progress. This is a fully robust version with performance almost as good as the synchronous version.

3.4 Chimp

CHIMP (Common High-level Interface to Message Passing) [22] is a message-passing system that was implemented by Alasdair Bruce, James (Hamish) Mills, and Gordon Smith at the Edinburgh Parallel Computing Center (EPCC) between 1991 and 1994. 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. It is portable and can run on many platforms including Solaris, Irix, AIX, OSF/1, and Meiko. CHIMP does not support Linux and is no longer in active development. It is not widely used, atleast in the US.

3.5 MPI/PRO

MPI/PRO [23] is commercial software introduced in April 1998 from MPI Software Technology, Inc. The company is a spin-off from Mississippi State University and led by Tony Skjellum. MPI/Pro supports all the 128 functions included in the MPI standard and runs on Linux, Windows, and Mercury Race Systems.

MPI/Pro has a number of features that proves it to be very efficient and robust for programming clusters of workstations. MPI/Pro provides multi-device architecture and multi-threaded design. Using multiple threads allows for independent message processing, asynchronous synchronization and notification, and a high degree of computation and communication overlapping.

Thread safety is assured at the user level. Other important design considerations are to optimize persistent mode MPI operations and derived datatypes. This allows for exploiting the high abstraction power of derived data types without loss of performance. Multiple queues are maintained for receive request to reduce the processing time and to increase the degree of concurrency. Two different protocols are used to handle short and long messages separately.

3.6 TCGMSG

The TCGMSG (Theoretical Chemistry Group Message-passing toolkit) [24] is a programming model and interface developed by Robert Harrison et al. of Pacific Northwest National Laboratory. It is used for writing portable parallel programs using message passing. It supports a wide variety of UNIX workstations, supercomputers, heterogeneous networks, and true parallel computers such as the Intel iPSC, Delta and Paragon, SGI Power Challenge, the IBM SP1/2 and Cray T3D.

TCGMSG was mainly designed having chemistry applications in mind, and provides limited functionality such as point-to-point communication, global operations and a simple load-balancing facility. It strongly enforces types and does not support wildcards. A message sent with a particular type must match that of the corresponding receive posted. The processes are connected with ordered, synchronous channels. Asynchronous communication is only provided on machines that explicitly support it.

3.7 PVM

PVM (Parallel Virtual Machine) [25] is a portable message-passing system designed to link separate host machines of varied architecture to form a “virtual machine” which is a single, manageable computing resource that can be used for concurrent or parallel computation. PVM was originally developed in 1989 as a research tool to explore heterogeneous network computing by Oak

Ridge National Laboratory (ORNL) but is now available as a public domain software package for free use.

PVM can be used at several levels. At the highest level, the transparent mode, tasks are automatically executed on the most appropriate computer. In the architecture-dependent mode, the user specifies which type of computer is to run a particular task. In low-level mode, the user may specify a particular computer to execute a task. In all of these modes, PVM takes care of all message routing, data conversion, and task scheduling across the network of incompatible computer architectures.

PVM is comprised of two main components - the PVM daemon process (pvmd3) and the library interface routines (libpvm3.a, libfpvm3.a, libgpvm3.a). The PVM daemon is a Unix process that oversees the operation of user processes within a PVM application and coordinates inter-machine PVM communications. One PVM daemon runs on each machine. The master daemon is started first, which spawns all other daemons. User processes communicate with each other through these daemons. They first talk to their local daemon via the library interface routines. The local daemon then sends/receives messages to/from remote host daemons.

Some of the limitations of PVM: the performance often depends on slow networks, low bandwidth due to lots of buffering, suffers high latency, mostly allows only coarse-grained applications, difficult to balance the load, recovery from host failure is expensive and sometimes impossible.

3.8 Unify

Unify [26] is a subset of MPI that was built on top of PVM. It is a dual-API (Application Program Interface) message-passing system that was developed at Mississippi State University. It allows users to write applications containing only MPI calls, or a mixture of MPI and PVM calls. The resulting executable runs in the PVM environment.

The intention of developing Unify was to demonstrate the relative ease of implementation of MPI, and to enable users to take current PVM applications and slowly migrate toward complete MPI applications, without having to make the complete conceptual jump from one system to the other. However, the project was not completed fully, although it did address the difficulty of mapping identifiers between the PVM and MPI domains, which it solved using additional function calls.

3.9 MVICH

MVICH [27,28] is an MPICH-based implementation of MPI for the Virtual Interface Architecture (VIA). It provides a high performance MPI for commodity high-speed networks (Gigabit Ethernet, Giganet, ServerNet II, or Fast Ethernet). VIA is an industry standard interface for System Area Networks (i.e. networks for clusters) that provides protected, zero-copy user-space inter-process communication. MVICH implements the MPICH Abstract Device Interface (ADI2) on VIA. MVICH is being developed at Lawrence Berkeley National Laboratory and is distributed with an open source license.

3.10 Vendor Versions

3.10.1 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 currently available implementation of MPI (IBM MPI) is rewritten from scratch.

IBM MPI runs on IBM SP systems and AIX workstation clusters. 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.

3.10.2 HP

HP provides an implementation of MPI that runs on all current HP hardware. 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 compliant with MPI 1.2.

3.10.3 Digital

Digital is a newcomer to the MPI world, having recently released a version for clusters of Alpha SMP servers connected by Digital's proprietary Memory Channel interconnect. 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), which 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.

3.10.4 SGI

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.

SGI/PVP MPI 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).

SGI/T3E MPI is derived from the T3D implementation developed at the Edinburgh Parallel Computing Center. The T3D version was in turn derived from the Chimp implementation. T3E MPI is robust, and well integrated with the environment.

SGI/Irix MPI is originally derived from MPICH, but has evolved considerably. It has also incorporated xmpi from LAM. SGI MPI is compliant with MPI 1.2.

3.11 MPICH ADI Implementations

3.11.1 MPICH for SCI-connected clusters

This paper [29] presents the design and implementation of an ADI-2 device with SCI adaptation for the current MPICH distribution. The implementation of the SCI-specific ADI-2 device ch_smi is based on the ch_shmem (shared-memory) device that is part of the MPICH distribution.

This implementation is a cost-effective cluster solution because of the extremely low latencies for small messages and the high maximum bandwidth. The free availability of the source code also helps to establish SCI connected clusters as a high-performance, solid yet affordable platform for technical and scientific computing next to the popular ethernet connected clusters.

3.11.2 MPI derived datatypes support in VIRTUS

The VIRTUal System (VIRTUS) project is focused on providing advanced features for high performance communication and I/O in cluster environments. This paper [30] presents the porting of MPICH 1.1.x on the Fast Messages (FM) library and the usage of the features of FM to provide efficient communication for non-contiguous data structures.

The porting concerns two different internal interfaces of MPICH 1.1.x called channel and ADI-2, respectively. The ADI-2 interface offers a rich set of primitives that allow the implementation of communication support to MPI derived data types.

These results confirm the effectiveness of FM's interface and implementation in delivering the raw hardware performance of the communication subsystem to the applications.

3.11.3 Porting MPICH ADI on GAMMA with flow control

The Genoa Active Message MAchine (GAMMA) [31] is an experimental prototype of a lightweight communication system based on the Active Ports paradigm and designed for efficient implementation over low-cost Fast Ethernet interconnects.

In order to make best use of the GAMMA programming interface while providing an MPI interface to the user, the original ADI layer was substantially changed atop GAMMA. Only two protocols were used for message delivery, namely, the *Eager* and the *Rendezvous* protocols.

The GAMMA ADI implementation is two-threaded, allowing for on-the-fly inspection of expected messages queue and minimal copy on receive. As a side effect of multi-threading implementation of the ADI level, the porting of MPICH should be thread-safe. Porting the ADI layer to GAMMA greatly speeds up point-to-point MPI communications, but is not as much a satisfactory answer for collective calls.

3.11.4 Design and implementation of MPI on Puma portals

The Puma operating system provides a flexible, lightweight, high performance message-passing environment for massively parallel computers. Message passing in Puma is accomplished through the use of a portal.

This paper [32] discusses the issues regarding the development of the MPICH on top of portals. It also describes the design and implementation of both MPI point-to-point and collective communications, and MPI-2 one-sided communications.

3.11.5 Multiple devices under MPICH

This paper [33] describes an enhanced MPICH architecture. Whereas other MPICH implementations support only one communication medium for internode communication at a time, the enhanced MPICH implementation supports different ones too. The basic idea is to introduce a so-called multi-device in addition to individual devices, so called subdevices, each of them supporting a certain communication medium.

An "ordinary" ADI-2 device performs only the mapping of a unique network to a flat virtual structure. In such a case there is only one communication device. Whereas in a heterogenous network several principles, e.g. global shared-memory (SCI) as well as packet or stream based ones, are available for increased performance. Hence, the multi-device has to support several devices, and therefore, an auxiliary interface within multi-device is needed. The performance obtained was nearly the same as that of a pure ADI-2 device.

3.11.6 MPICH on the T3D: A case study of high performance message passing

This paper [34] describes the design, implementation, and performance of MPICH to the Cray T3D massively parallel processing system. The Cray T3D contains up to 2048 processors connected by a high-speed, 3-D torus communication network. It has a physically distributed shared-memory, where each processing element (PE) has local memory that is globally addressable.

Cray's SHared MEMory access library (SHMEM) for remote memory transfers is used for the implementation. This library contains a plethora of functions for point-to-point and collective communication, synchronization, and cache manipulation.

3.12 MPICH Channel Interface Implementations

3.12.1 Wide-area implementation of the Message Passing Interface

The wide-area environment introduces challenging problems for the MPI implementor, due to the heterogeneity of both the underlying physical infrastructure and the software environment at different sites. This paper [35] describes an MPI implementation that incorporates solutions to these problems.

The MPICH implementation of MPI was extended to use communication services provided by the Nexus communication library and authentication, resource allocation, process creation/management, and information services provided by the I-Soft system and the Globus metacomputing toolkit.

Nexus provides multi-method communication mechanisms that allow multiple communication methods to be used in a single computation with a uniform interface; I-Soft and Globus provided standard authentication, resource management, and process management mechanisms.

The result is a system that allows programmers to use simple, standard commands to run MPI programs in a variety of metacomputing environments (freely combining heterogeneous workstation and massively parallel resources), while making efficient use of underlying networks.

3.12.2 MPICH-PM: Design and implementation of Zero Copy MPI for PM

MPICH-PM [36] consists of the MPICH implementation of the MPI standard, ported to the high-performance, communications library PM. This research paper presents the MPI implementation using a zero-copy message transfer mechanism, called Zero Copy MPI, which was designed and implemented based on the MPICH "Channel Interface". The PM communication driver is used as the low-level communication layer, which supports not only a zero-copy message transfer but also message-passing mechanisms.

The Zero Copy MPI achieves good performance compared to other zero-copy implementations. It also supports a multi-user environment where many MPI applications may run simultaneously on the same nodes.

3.12.3 MPI-StarT: Delivering network performance to numerical applications

This article [37] describes the development of MPI-StarT, an MPICH "Channel Interface" implementation for a cluster of SMPs interconnected by the StarT-X cluster interconnect. StarT-X allows a cluster of PCI-equipped host platforms to communicate with an order-of-magnitude better performance than a conventional local area network. MPI-StarT implementation is centered around preserving and delivering the StarT-X communication performance to user applications.

MPI-StarT represents a collaboration between a numerical applications programmer and the StarT-X architect. The collaboration started with the modest goal to satisfy the communication needs of MITMatlab. However, by supporting the MPI standard, MPI-StarT has been successful in extending support to other MPI applications.

Although the MPI-StarT was implemented on the Channel Interface, some changes were also made to the MPICH's ADI and Protocol Layers for correct and optimal operations.

3.12.4 MPICH/Madeleine: A true multi-protocol MPI for high performance networks

This paper [38] introduces a version of MPICH handling different networks simultaneously and efficiently. The core of the implementation relies on a device called ch_mad, which is based on a generic multi-protocol communication library called Madeleine.

One approach for multi-protocol support in MPICH is to use the Abstract Device Interface (ADI) layer, which allows plugging different network support modules. In practice, however, a heavy integration work has to be done each time a new device is to be supported, in order to preserve inter-device coexistence. As a consequence, there is currently no MPICH version supporting network heterogeneity.

An alternate solution is to get a multi-protocol version of MPICH through the use of a generic multi-protocol communication library such as Madeleine, the communication subsystem of the PM environment. This multi-protocol version of MPICH generally outperforms other free or commercial implementations of MPI.

CHAPTER 4

PERFORMANCE COMPARISON

This chapter briefly summarizes the point-to-point communication performance of most of the current message-passing libraries (described in a paper by Turner et al [39]). The throughput graphs of raw TCP, MPICH, LAM/MPI, MPI/Pro, MP_Lite, PVM and TCGMSG libraries are compared using several Gigabit Ethernet NICs (Network Interface Cards). The NetPIPE [40,41] graphs are plotted using throughput (Mbps) versus message size (Bytes) on a logarithmic scale. It clearly shows the throughput for each transfer block size and the maximum throughput that can be achieved.

All the graphs were plotted from data taken on two 1.8 GHz Pentium 4 PCs with 768 MB of PC133 memory and 32-bit/33-MHz PCI slots, loaded with Linux 2.4.7-10 kernels. The two machines were connected back-to-back using Gigabit Ethernet (GE) cards. The performance was measured on three GE cards, namely, the inexpensive TrendNet copper GE cards, Netgear fiber GE cards and the expensive SysKonnect GE cards. The SysKonnect GE cards provide a low latency and high bandwidth for jumbo frames with an MTU (Message Transfer Unit) size of 9000 Bytes. The Netgear and the TrendNet GE cards use the standard MTU size of 1500 Bytes. All the message-passing libraries were appropriately tuned and available parameters optimized to provide peak performance.

The message passing performance of all libraries on the Netgear fiber GE cards and the TrendNet copper GE cards are presented in Figure 4.1 and Figure 4.2 respectively. The raw TCP performance for both cards is around 550 Mbps with a message latency of 120 μs and 200 μs respectively.

On the Netgear fiber GE cards, most of the libraries deliver performance close to that of raw TCP. However, for large messages, MPICH and PVM show a 20-25% loss in performance. On the cheaper TrendNet copper GE cards, most libraries have problems and they peak out between 200-300 Mbps. Only MPICH and MP_Lite perform well on these cards. The poor performance of the other libraries is due to the smaller TCP socket buffer sizes used, which is hard-coded and is not available to the user as a tunable parameter.

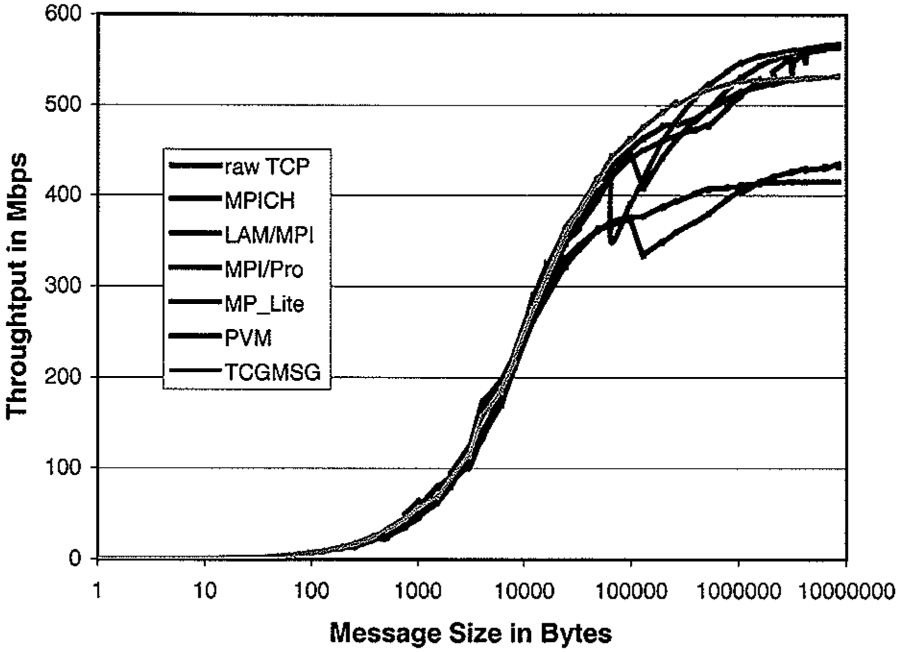

Figure 4.1. Throughput graph across the Netgear fiber GE cards on the PC cluster.

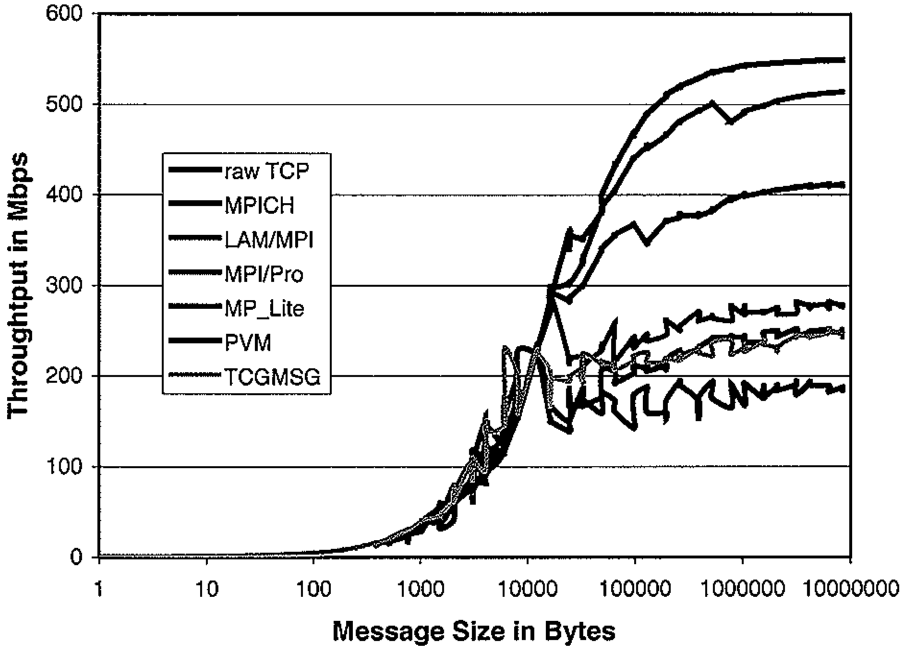

Figure 4.2. Throughput graph across the TrendNet copper GE cards on the PC cluster.

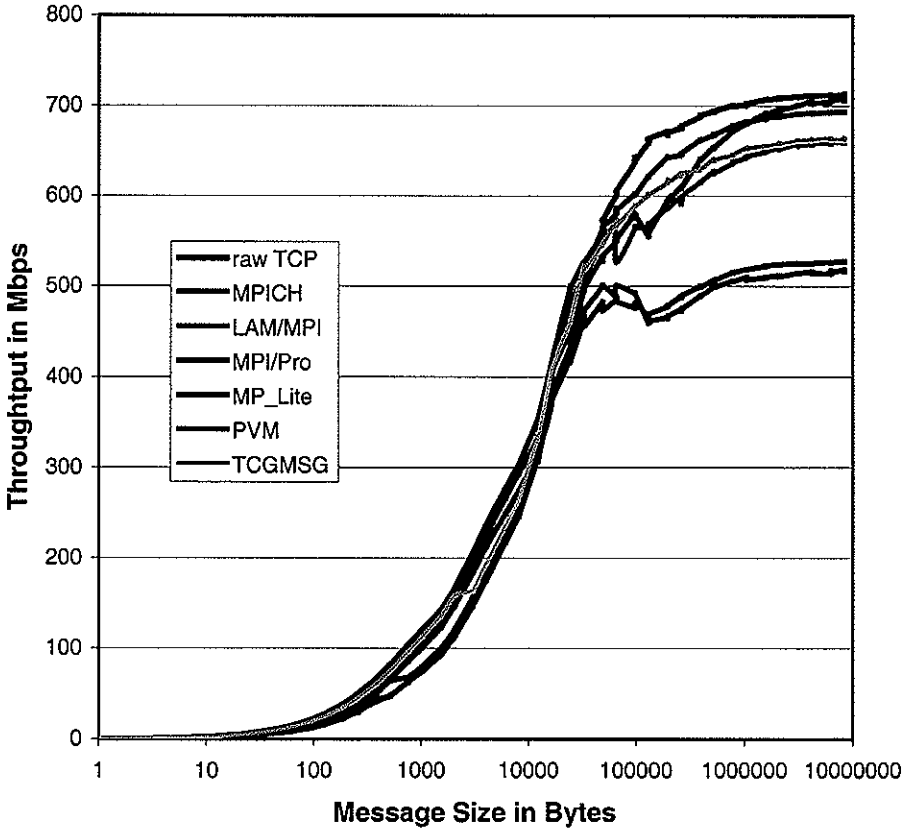

Figure 4.3. Throughput graph across the SysKonnect GE cards on the PC cluster with jumbo frames.

The use of jumbo frames (9000 Bytes MTU size) on the SysKonnect cards, as shown in Figure 4.3, enhances the raw TCP throughput to 700 Mbps with a low message latency of 32 μs. All the libraries seem to perform reasonably well matching the throughput of the raw TCP except MPICH and PVM.

From the above results on the Gigabit Ethernet cards, it is evident that MP_Lite performs very well compared to MPICH. The use of larger messages on the SysKonnect cards clearly illustrates the superior performance of MP_Lite compared to MPICH. By implementing MPICH on top of MP_Lite at a lowermost level (the "Channel Interface" level), the high throughput of MP_Lite can be delivered to MPICH. Also, the full MPI specification of MPICH can be retained, which is not offered by MP_Lite.

CHAPTER 5

INTRODUCTION TO MP_LITE

5.1 Overview

MP_Lite is a lightweight message-passing library that was primarily designed to deliver peak performance to the applications. It is an ideal research tool that is portable and has many user-friendly features built into it. MP_Lite implements the core set of functions like blocking and non-blocking "*sends*" and "*receives*", common global operations, synchronization and broadcasting, which form the most widely used commands in parallel codes.

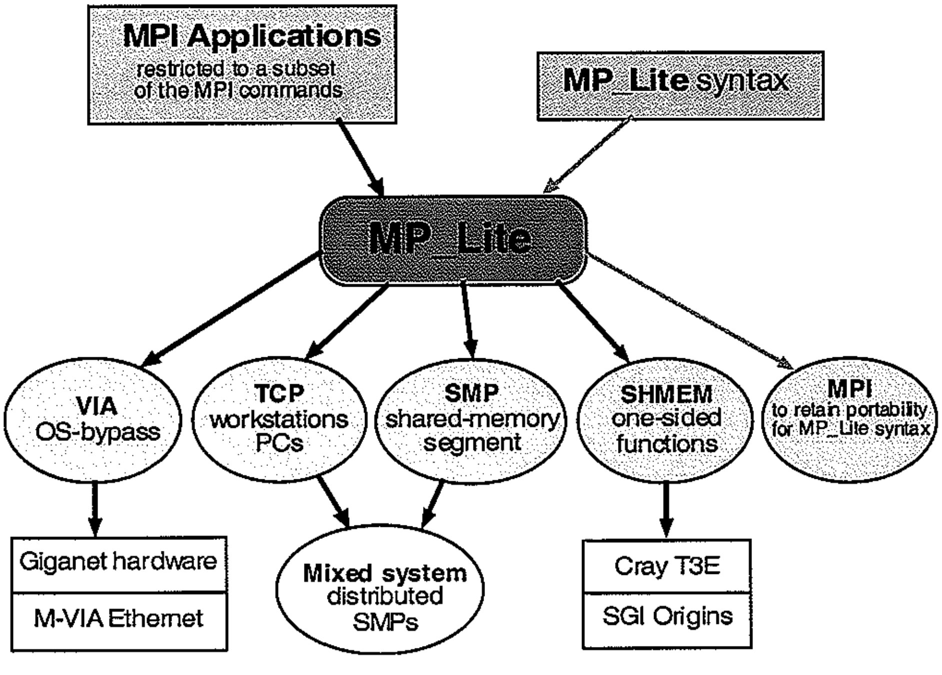

Figure 5.1. The structure of MP_Lite.

MP_Lite is not a full implementation of the MPI standard. A full implementation of MPI (MPICH, for instance) supports advanced features like derived datatypes, communicators, parallel

I/O, remote memory access, and dynamic process management. MP_Lite, on the other hand, provides enough functions that most parallel codes need. Thus, by keeping it simple, buffering is minimized and almost all the performance of the TCP layer is delivered to the application layer.

The structure of MP_Lite is presented in Figure 5.1. MP_Lite can be run on TCP on clusters of workstations, on SMP systems, on Cray T3E's native SHMEM library and SGI Origin systems. It can also be run on VIA module, which bypasses the operating systems to provide lower latency and higher bandwidth. In order to retain complete portability, applications using the MP_Lite syntax can also be run on systems where MPI is installed.

5.2 MP_Lite Controllers

MP_Lite currently has three different controllers; the synchronous controller, SIGIO interrupt driven controller and Pthreads controller. The first two controllers are discussed below. The Pthreads controller is still under development.

5.2.1 Synchronous controller

This is the simplest version in which the performance almost equals that of the raw socket calls. The send and receive TCP buffer sizes are increased to their maximum values so that is can accommodate very large messages. This avoids extra buffering and memory-to-memory copies, thus providing superior performance. The user is however responsible to keep the message traffic within the limits and see that the traffic does not exceed the TCP buffers at any time. Otherwise, the communication may just freeze up.

In this version, the asynchronous sends send the message fully and the wait function does nothing. The asynchronous receives just log the message information allowing the wait function to handle the actual receives. The send function pushes the message to the TCP buffer directly. The receive function first checks the 'message queue' (which buffers out-of-order messages). If it does not find the message in the 'message queue', it checks the TCP buffer. Messages in the TCP buffer that do not have a matching header corresponding to the posted receive are copied to the 'message queue'. Thus, out-of-order messages are handled by buffering, thereby reducing the efficiency. A message with 'any source' checks all the messages in the TCP buffer and buffers all of them until a suitable match is found. The programmer should therefore avoid writing code that causes this extra buffering.

One limitation of the synchronous controller is that if the nodes send more messages than the capacity of the TCP buffers, a lock-up condition occurs. The user must ensure that the message traffic is within the TCP buffer size.

5.2.2 SIGIO interrupt driven controller

This is the fully robust version and is the default mode for UNIX systems. It does not suffer from the lock-up condition that happens in the synchronous version. It performs very well even with the default TCP buffer size, which is usually around 64 kB, but performs even better when the TCP buffer size is increased.

The asynchronous controller handles asynchronous sends and receives using the SIGIO interrupt that is generated when there is some data in the TCP buffer. For an asynchronous send, a SIGIO is generated when the data moves out from the TCP buffer. A *sigio_handler()* routine captures the SIGIO interrupts and services them by pushing more data into the appropriate send buffers. An asynchronous receive gets all the data from the corresponding TCP receive buffer. When more data arrives in the TCP buffer, a SIGIO interrupt is generated which services the active receives. The wait routine just blocks until the data transfer is complete. The blocking sends and receives are simply the asynchronous routines followed by a call to the wait function.

The lock-up condition does not happen here, because the source node does not block on a send. The wait routine allocates a send buffer and copies the extra data to the send buffer. When there is space available in the TCP buffer, the *sigio_handler()* completes the transfer by pulling data from the send buffer instead of its original place and frees the memory after the transfer is complete. Hence, it is robust and safe.

5.3 Features of MP_Lite

The following is a list of features in a nutshell that is provided by the MP_Lite library for writing parallel codes:

- Simple and freely available.
- Implements the core message-passing routines used by most parallel codes.
- Can be run as three different modes – synchronous, asynchronous and Pthreads modes.
- Offers better performance (closer to that of raw TCP) compared to most other message-passing libraries. This is especially true in the case of high-speed networks like Gigabit Ethernet.
- Portable and runs on different platforms like the TCP, SHMEM and VIA modules.
- An ideal research tool that takes only few seconds to compile.
- User-friendly with several debugging and trace options.

CHAPTER 6

THE ARCHITECTURE OF MPICH

MPICH has a layered design where every layer corresponds to an abstraction of a communication device, which provides a set of services to the upper layer and, in turn, requires a set of services that should be provided by a lower layer. Specifically, the upper layer provides the API whereas the lower layer can be customized to exploit the hardware architecture, thus optimizing performance. The layered approach also allows for maximizing code sharing across implementations.

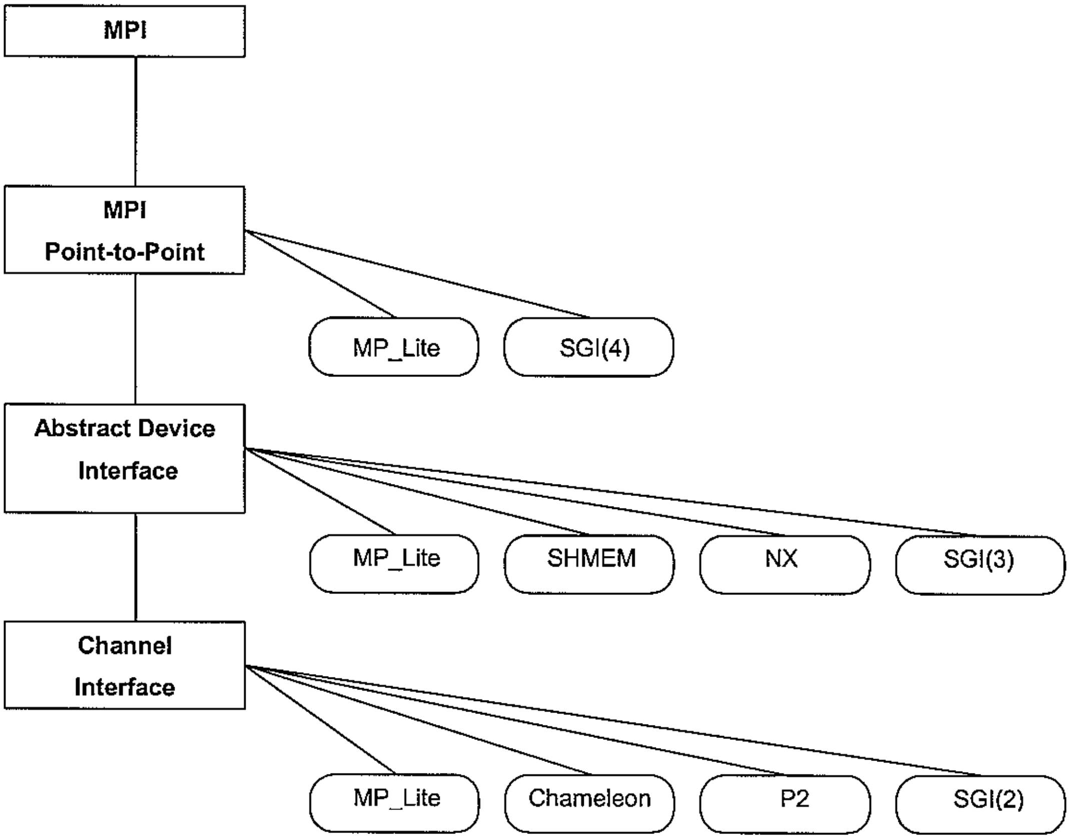

Figure 6.1. The structure of MPICH.

MPICH contains four layers. From bottom to top, they are:

- *Channel Interface device layer* - This includes various operating system facilities and software drivers for different communication devices.
- *ADI layer* - This layer encapsulates the differences of various communication devices and provides a uniform interface to the upper layer. The ADI layer exports a point-to-point communication interface.
- *MPI point-to-point primitives* - This is built directly upon the ADI layer. It manages high-level MPI communication semantics such as contexts, communicators and datatypes.
- *MPI collective primitives* - This is built upon the point-to-point primitive layer. Collective-communication primitives include operations such as barrier, broadcast, reduce and gather.

Messages share the same channel for both point-to-point communication and collective communication. MPICH uses special tags to distinguish messages that belong to a user, point-to-point communication, and internal messages for collective operations.

MP_Lite can be integrated at the Channel Interface, ADI or point-to-point levels. Implementing MP_Lite at the lowest level, the Channel Interface level, delivers most of its performance to applications that use MPICH. Hence, the ADI and point-to-point devices were not implemented using MP_Lite.

6.1 Abstract Device Interface

The Abstract Device Interface (ADI) is the key component in the layered architecture of MPICH, and is responsible for providing a portable, point-to-point message-passing interface to the generic upper layers. All the user-callable MPI functions are implemented using a set of forty different macros and function definitions that constitute the ADI layer. The ADI layer provides hardware independent access to the communication and synchronization primitives in the lower layer. It performs the following functions:

- Specifies messages to be sent or received.
- Moves data between the API and the message-passing hardware.
- Manages lists of pending messages (both sent and received).
- Provides basic information about the execution environment.
- Provides software emulations of any functions that may not be supported by some devices.

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.

There are many ADI devices (implementations) in the MPICH source tree that make it portable. Some devices may provide limited functionality while others may provide more complex functionality. One such implementation of the ADI layer is in terms of a lower layer called the "Channel Interface" layer. The Channel Interface is a much simpler interface, and it is the fastest way to add a new device to MPICH. This approach was used to attach MP_Lite to MPICH as a Channel Interface device.

6.2 The Channel Interface

The Channel Interface is the easiest way to port MPICH to a new environment. It is a low-level communication interface that focuses on simple data-transfer operations. It essentially transfers data from the address space of one process to that of another process. The Channel Interface uses two kinds of messages, namely control and data messages.

Control message:

It is used to rapidly transfer control information or small user-data. MPICH employs the following four types of control messages:

- Small user-data message (encapsulated in a control message).

- *Ready-to-Send* message sent by the source to the destination node to announce the availability of a message.
- *Ready-to-Receive* message sent by the destination to the source node to indicate that the destination node is ready to receive a large data.
- Flow control messages.

Data message:

This is used to transfer data on the network. Messages smaller than the *Eager/Rendezvous* threshold are sent using the *Eager* protocol. Messages larger than the threshold value cannot be buffered at the destination since it introduces delays due to memory-memory copies and eats up large amount of memory. Therefore, very large messages are sent using the *Rendezvous* protocol that performs a handshake before the data is sent. The handshaking ensures that the destination node is ready to receive the large amount of data.

6.3 Channel Interface Functions

The Channel Interface consists of a minimal set of five required functions, which are responsible for sending and receiving contiguous messages (carrying data or control information). The simplest set of required functions for the Channel Interface are presented below.

MPID_ControlMsgAvail

Does a non-blocking check for the availability of a control message.

```
int MPID_ControlMsgAvail( void )
```

MPID_RecvAnyControl

Reads the next control message. If no messages are available, blocks until one can be read.

```
void MPID_RecvAnyControl( MPID_PKT_T *pkt, int size, int *from )
```

MPID_SendControl

Sends a control message.

```
void MPID_SendControl( MPID_PKT_T *pkt, int size, int dest )
```

MPID_RecvFromChannel

Receives data from a particular channel.

```
void MPID_RecvFromChannel( void *buf, int maxsize, int from )
```

MPID_SendChannel

Sends data on a particular channel.

```
void MPID_SendChannel( void *buf, int size, int dest )
```

In addition to the above functions, the Channel Interface may also provide support for non-blocking operations. They are not mandatory, but can be used if available. The non-blocking operations improve the efficiency by overlapping computation and communication, and offer greater robustness. The Channel Interface also provides out-of-band operations, which perform remote memory operations without local intervention. The *Rendezvous* protocol that transfers large messages makes use of the out-of-band capability. A complete list of functions for the blocking, non-blocking and out-of-band operations for the MPICH-MP_Lite device is presented in the Appendix.

6.4 The Channel Interface Protocols

In MPICH, a message consists of two different parts: the message body that contains the data to be transmitted, and the envelope that has the header information (message source, destination, tag and length). Based on the message length, the Channel Interface uses three different message transfer protocols for data exchange.

6.4.1 The Short protocol

In the *Short* protocol, the data is sent along with the header to the destination. The total size of the data and the header must be less than the MTU (Message Transfer Unit) used. It is used only for very short messages and offers very low latency. The *Short* protocol may be a performance optimization for interconnect networks that send fixed size packets.

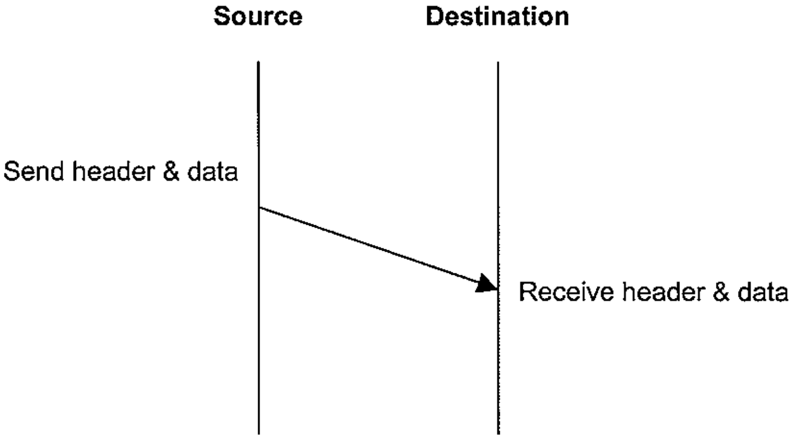

Figure 6.2. The *Short* protocol.

6.4.2 The Eager protocol

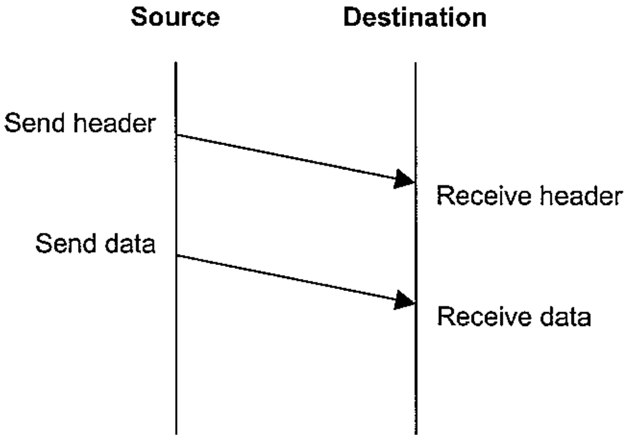

Figure 6.3. The *Eager* protocol.

In the *Eager* protocol, the header is sent first followed by the data in two separate packets. The data is delivered without waiting for the destination node to request it. The protocol assumes that the destination node has enough space to store the data.

If a receive was already posted, the data is copied from the TCP buffer directly to the user space. If a receive was not pre-posted, some space has to be allocated on the destination node to store the data locally. When the receive is actually posted, the data is copied from the local buffer to the user space. Thus, the data is copied twice for unexpected messages. This protocol is used for messages that do not fit within a single packet yet small enough to be buffered at the destination.

6.4.3 The Rendezvous protocol

The *Eager* protocol relies on statically allocated resources and is not suitable for messages that exceed the size of the receive buffer. The *Rendezvous* protocol directly copies the message to the application's memory bypassing the TCP buffers. It does a handshake before transmitting the message; therefore, it is more robust and safe.

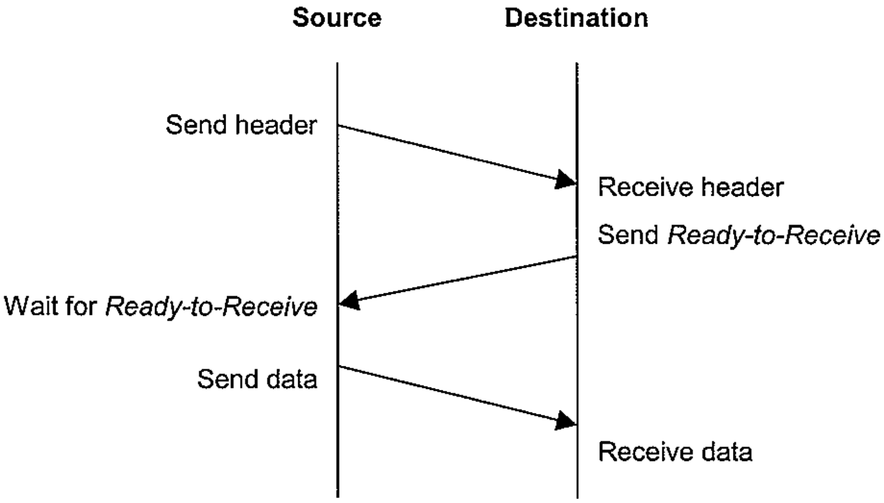

Figure 6.4. The *Rendezvous* protocol.

The *Rendezvous* protocol does not deliver the data until the destination node requests it. The source node initiates the communication by sending the header to the destination node. When the destination node is ready to receive the data, it acknowledges by sending a *Ready-to-Receive* packet to the source node. The source node then sends the actual data to the destination node.

The source and destination nodes are synchronized before the actual data is transferred. Thus, there is no need for intermediate buffering except that of the header. The data is delivered only when user space is available at the destination node and, therefore, the *Rendezvous* protocol is robust and safe. However, the handshaking during synchronization introduces additional delays. This protocol is used for very large messages.

6.4.4 Threshold values for the MPICH message protocols

MPICH, by default, uses the *Short* protocol for messages of size less than 1024 Bytes, the *Eager* protocol for messages of size between 1024 Bytes and 128 kB, and the *Rendezvous* protocol for messages larger than 128 kB. The *Short/Eager* threshold value can be changed by modifying the default value 1024 of the macro MPID_PKT_MAX_DATA_SIZE in the *mpid/ch2/packets.h* file. To change the *Eager/Rendezvous* threshold value, the default value 128000 must be changed in the files *mpid/ch2/chinit.c* and *mpid/ch2/chcancel.c.*

The switch from *Short* to *Eager* protocol must happen when the cost of copying the data (in the *Short* protocol) is the same as the cost of sending an additional control message (in the *Eager* protocol). The same holds for the switch from *Eager to Rendezvous* protocol.

6.4.5 Blocking and non-blocking communication

The *Eager* and *Rendezvous* protocols are further classified based on the method by which the data is delivered:

- Non-blocking

  In this mode, the source node (or destination node) can call a system service routine to initiate the send (or receive) and then return back to the user process without waiting for the action to complete. The *mpid/ch2/chneager.c* and *mpid/ch2/chnrndv.c* implement the required functions for the non-blocking versions of the *Eager* and *Rendezvous* protocols respectively.

- Blocking

  In the blocking mode, the source node (or destination node) waits until the send (or receive) action is complete before it returns to the user process. This mode is implemented in the *mpid/ch2/chbeager.c* and *mpid/ch2/chbrndv.c* files for the *Eager* and *Rendezvous* protocols respectively.

The *Short* protocol always uses the blocking method for communication.

6.5 Implementing MP_Lite as a MPICH Channel Interface Device

This section discusses the integration of MP_Lite as an efficient MPICH Channel Interface device. MP_Lite is a lightweight message-passing library that delivers performance close to that of the raw TCP layer. MPICH is a full implementation of MPI that offers less performance compared to MP_Lite. By implementing MP_Lite as a MPICH Channel Interface device, the performance of MP_Lite can be delivered to MPICH and at the same time full MPI implementation of MPICH can be retained.

MP_Lite can be implemented as two devices, ch_mplite_blk and ch_mplite_nblk for the blocking and non-blocking communications respectively.

Steps involved in creating a ch_mplite_blk device:

1. The command `NewDevice` in MPICH is used to create a new device.

   ```
   cd mpich-x/mpid
   NewDevice -raw mplite_blk
   ```

   This creates a new directory called `ch_mplite_blk` in the `mpich-x/mpid` directory.

2. Configure MPICH for the new device including other parameters like compiler, architecture, etc.

   ```
   ./configure --prefix=~/mpich-x --with-device=ch_mplite_blk -rsh=ssh
   ```

3. The following files in the `ch_mplite_blk` directory have to be edited:

   `channel.h` (See Appendix)

`mplite_blkpriv.c`

`chdef.h`

`mpid_time.h`

`Makefile`

4. Create a directory `mplite` in the `ch_mplite_blk` directory and copy MP_Lite source code to it.
5. Edit the `mpirun.in` and `mpirun.args.in` scripts in the `mpich-x/util` directory.
6. Copy the `mpirun.mplite_blk` file to `mpich-x/bin` directory and the `mpirun.ch_mplite_blk` file to the `mpich-x/mpid/ch_mplite_blk` directory.
7. `make |& tee make.log` in the `mpich-x` directory.
8. To execute a program, compile it and use `mpirun`.

   ```
   mpich-x/bin/mpirun -np 2 program_name arguments
   ```

   Use the `-nolocal` option if the program does not run on the local machine (only for the p4 device). Before doing `mpirun`, make sure that the `machines.$arch` file in the directory `mpich-x/util/machines` has the list of hosts that execute the code.

To implement a non-blocking device, all the steps are the same except that the word `mplite_blk` is replaced by `mplite_nblk` and the `mpich-x/mpid/ch_mplite_nblk/Makefile` is modified to include the non-blocking files `chneager.c` and `chnrndv.c`.

Functions added to MP_Lite:

The function MP_Aprobe was added to MP_Lite to support the implementation of the Channel Interface device. MP_Aprobe does a non-blocking test for a message.

```
int MP_AProbe( int nbytes, int source, int tag, int *flag )
```

If the message is present in the message queue or in the TCP buffer, it returns a true; otherwise, it returns a false. If a message is present in the TCP buffer but the header information does not match the request, then that message is pushed to the message queue and the TCP buffer is probed again.

CHAPTER 7

PERFORMANCE RESULTS

7.1 The Test Environment

The performance measurements were executed on a PC mini-cluster and an Alpha cluster mini-cluster. The PC cluster consists of two 1.8 GHz Pentium 4 PCs with 768 MB of PC133 memory and 32-bit/33-MHz PCI slots, running the RedHat Linux 2.4.7-10 kernel. The Alpha cluster consists of two 500 MHz Compaq DS20 Alpha workstations with 1.5 GB memory, running RedHat Linux 2.4.17. The DS20s have a wider 64-bit/33-MHz PCI slots. Both the hosts within the two clusters are connected using Gigabit Ethernet (GE).

The libraries were tested using a variety of network hardware. The Netgear fiber, TrendNet copper and SysKonnect GE NICs were used for the PC cluster, and the Netgear fiber and SysKonnect GE NICs were used for testing on the Alpha cluster.

The current MPICH version, MPICH-1.2.3, was used for testing. The performance of raw TCP, MP_Lite (SIGIO version), MPICH-p4 (a blocking device) and MPICH-MP_Lite (both blocking and non-blocking devices) were compared.

Table 7.1. Test-bed.

Cluster Name	Processor	RAM	NICs	OS
PC cluster	1.8 GHz Pentium 4	768 MB	Netgear Fiber TrendNet Copper SysKonnect	Linux 2.4.7-10
Alpha cluster	500 MHz Compaq DS20	1.5 GB	Netgear Fiber SysKonnect	Linux 2.4.17

7.2 NetPIPE

NetPIPE (Network Protocol Independent Performance Evaluator) is a tool developed at Ames Laboratory to measure network bandwidth. It uses multiple ping-pong tests to evaluate the point-to-point performance between two idle nodes on a network. It starts with a simple 1-byte message and gradually increases the message size at regular intervals. Each data point is taken using several ping-pong measurements to increase the accuracy.

The output is a file that contains the transfer time, throughput, block size and transfer time variance for each data point. The throughput graph can be obtained by plotting throughput versus transfer block size.

7.3 Performance using Gigabit Ethernet

7.3.1 Performance on the PC mini-cluster

Figure 7.1 shows the throughput comparison of raw TCP, MP_Lite and MPICH devices (ch_p4, ch_mplite_blk and ch_mplite_nblk) on the PC cluster with the Netgear fiber GE cards. Raw TCP offers a maximum throughput of 565 Mbps. MP_Lite performs close to TCP with a peak network bandwidth of 540 Mbps. Both the blocking and non-blocking MPICH-MP_Lite implementations almost trace the MP_Lite throughput curve. The MPICH-p4 device delivers only a maximum throughput of 430 Mbps. For larger messages, it suffers a loss of 25% in throughput compared to a loss of 5% suffered by the MPICH-MP_Lite device with respect to the raw TCP curve.

The small dip in the MPICH curves (ch_p4, ch_mplite_blk and ch_mplite_nblk devices), near the 512 kB message size, is due to the switch in the MPICH message transfer protocol from *Eager* to *Rendezvous*.

Similar performance can be noticed on the TrendNet copper GE cards on the PC cluster (Figure 7.2). The MPICH-MP_Lite devices are close to the MP_Lite throughput curve and offer throughput higher than that offered by the MPICH-p4 device.

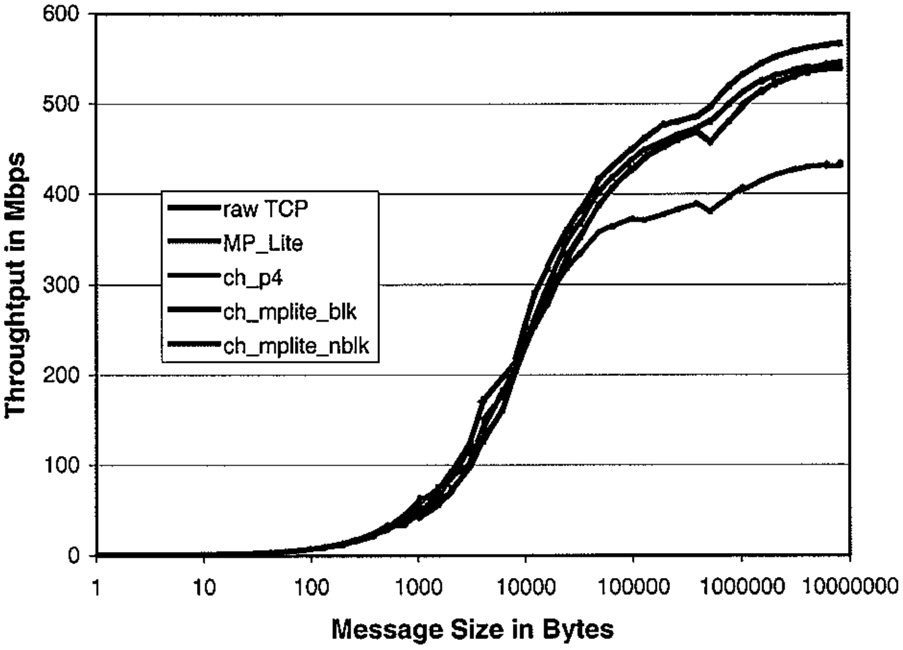

Figure 7.1. Throughput on the PC cluster with Netgear fiber GE cards.

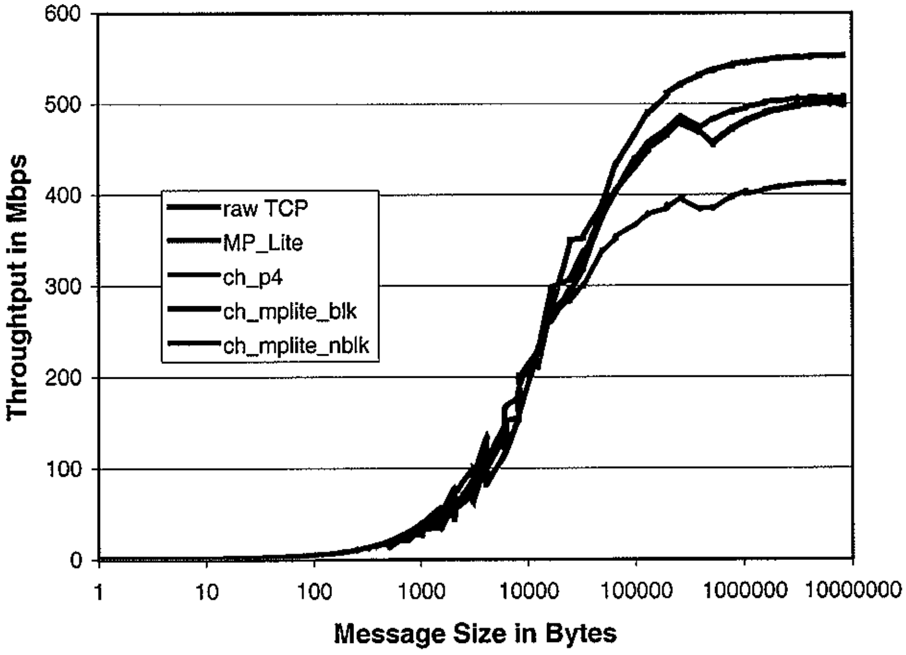

Figure 7.2. Throughput on the PC cluster with TrendNet copper GE cards.

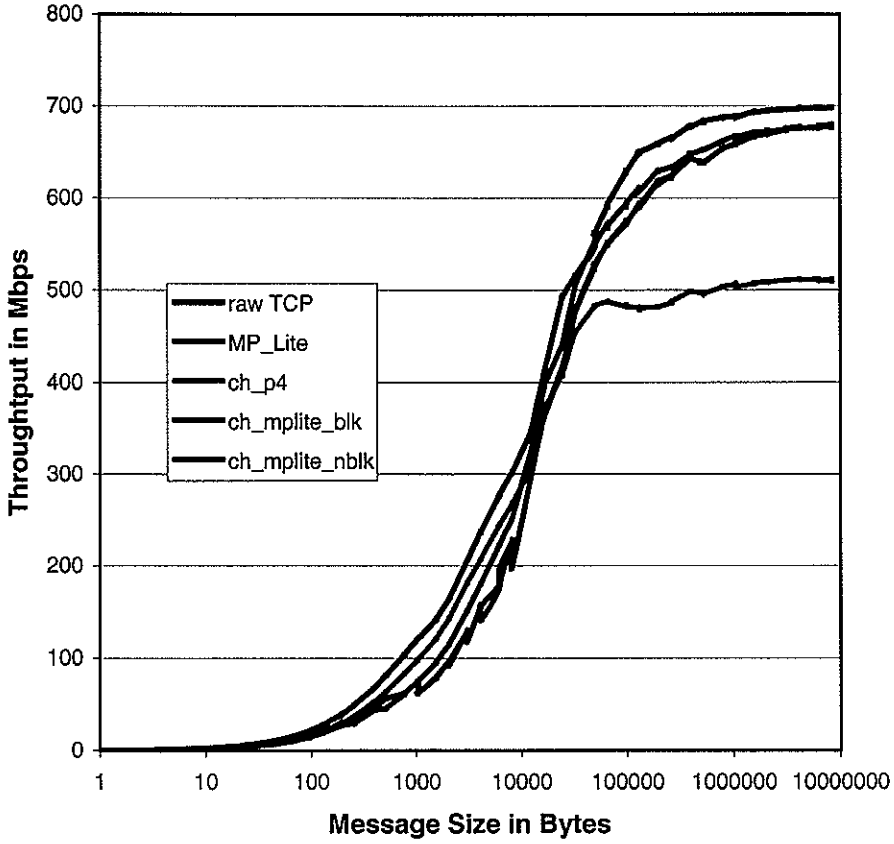

Figure 7.3. Throughput on the PC cluster with SysKonnect GE cards using jumbo frames.

Figure 7.3 shows the throughput on the PCs with the SysKonnect GE cards for jumbo frames of MTU size 9000 Bytes. The use of jumbo frames clearly demonstrates the higher throughput offered by the MPICH-MP_Lite device compared to the MPICH-p4 device. Raw TCP has a peak bandwidth of 700 Mbps. MP_Lite offers a maximum throughput of 680 Mbps, which is within 3% of the TCP results. The MPICH-p4 device delivers a maximum throughput of 510 Mbps with a performance loss of nearly 30% for large messages. Both the blocking and non-blocking MPICH-MP_Lite devices deliver the full performance of the MP_Lite library.

7.3.2 Performance on the Alpha mini-cluster

Figures 7.4 and 7.5 show the performance results on the Alpha mini-cluster with the Netgear fiber and SysKonnect Gigabit Ethernet cards. Raw TCP offers a maximum throughput of 525 Mbps and 900 Mbps on the Netgear and SysKonnect GE cards respectively. The MP_Lite library and the MPICH-MP_Lite devices match the TCP curve to within a few percentages for both sets of the Gigabit Ethernet cards.

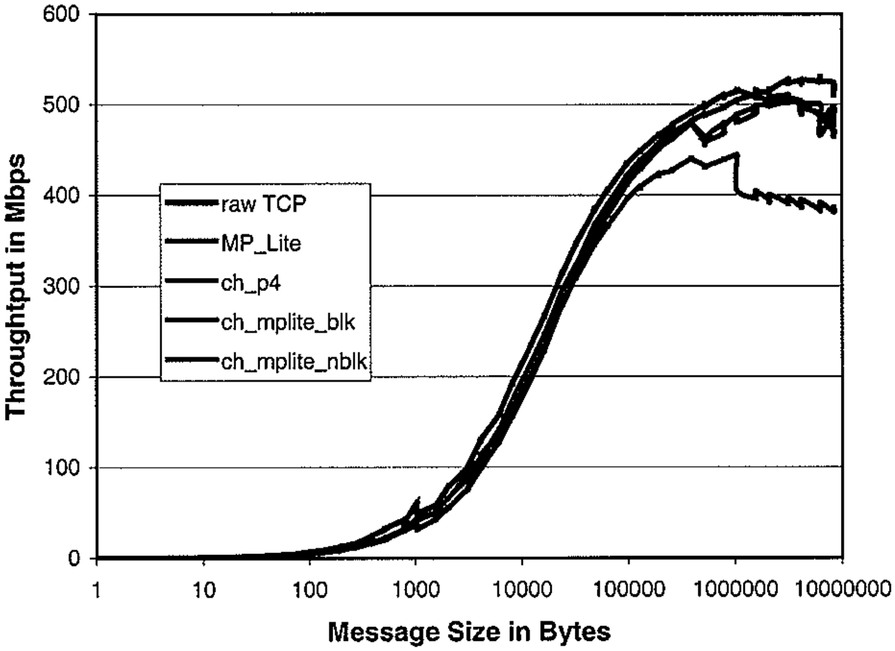

Figure 7.4. Throughput on the Alpha cluster with Netgear fiber GE cards.

For larger messages, the MPICH-p4 device peaks out at 400Mbps and suffers a performance loss of 25-30% on the Netgear fiber GE cards with a default MTU size of 1500 Bytes. It peaks out at 550 Mbps on the faster SysKonnect GE cards with jumbo frames of MTU size 9000 Bytes, suffering a performance loss of about 35-40%.

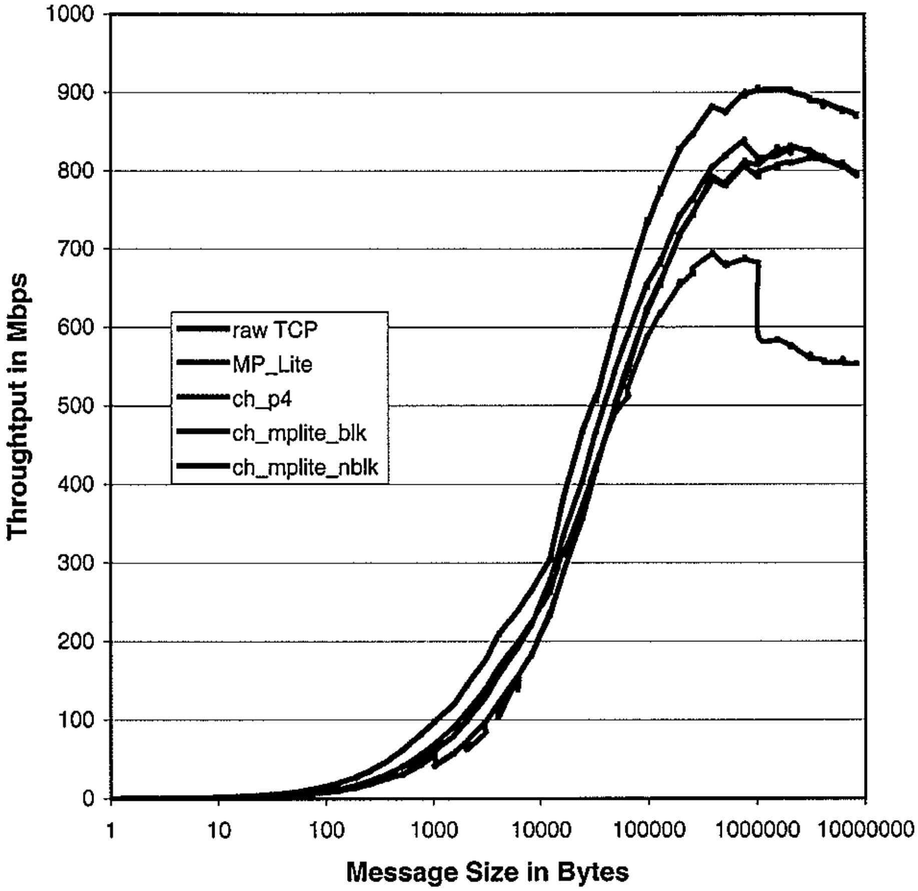

Figure 7.5. Throughput on the Alpha cluster with SysKonnect GE cards using jumbo frames.

7.4 Effect of Eager/Rendezvous Threshold on the PC Cluster

The following plot (Figure 7.6) shows the effect of the *Eager/Rendezvous* threshold of MPICH on the throughput results. Since the non-blocking version of the MPICH-MP_Lite Channel Interface device performs similar to the blocking version, it is omitted.

The default *Eager/Rendezvous* threshold value in MPICH is 128 kB, which was chosen to cater to the memory needs of the older systems when MPICH was being developed. With the default value, the performance degrades considerably near the region where the protocol changes from *Eager* to *Rendezvous*. This is due to the handshaking performed by the *Rendezvous* protocol before it transmits a large message. The dip gradually diminishes by increasing the threshold value to 256 kB, and then to 512 kB. For all the performance tests, a threshold value of 512 kB was used.

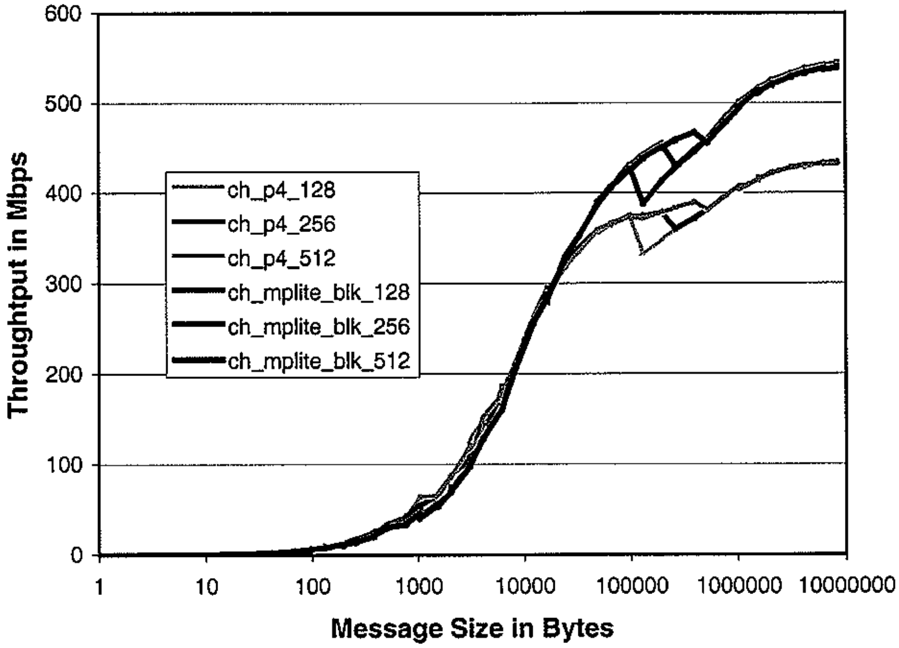

Figure 7.6. Effect of *Eager/Rendezvous* threshold on the PC cluster with Netgear fiber GE cards.

7.5 Summary

In this chapter, the performance of raw TCP, MP_Lite, MPICH-p4 device and MPICH-MP_Lite blocking and non-blocking devices were presented. Both the blocking and non-blocking devices deliver the same performance on both the test-beds and close to that of the MP_Lite library. The MPICH-MP_Lite device definitely performs better on both the PCs and Alphas compared to the MPICH-p4 device. Also, increasing the *Eager/Rendezvous* threshold value gave better results.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK NEEDED

This chapter presents the conclusions of the research on the implementation of the MPICH-MP_Lite blocking and non-blocking Channel Interface devices.

The performance of the MPICH-MP_Lite Channel Interface device was measured on the PC and Alpha mini-clusters using Netgear fiber, TrendNet copper and SysKonnect GE cards. For large messages, the MPICH-MP_Lite Channel Interface device has a bandwidth closer to that of raw TCP as compared to the MPICH-p4 Channel Interface device. This can be clearly seen in the faster environment of the Alpha mini-cluster connected using SysKonnect GE cards with jumbo frames of MTU size 9000 Bytes. On this test-bed, the MPICH-MP_Lite Channel Interface device offers a peak performance of 830 Mbps while the MPICH-p4 Channel Interface device provides a throughput of only 550 Mbps for large messages. Compared to the raw TCP peak throughput of 900 Mbps, the MPICH-p4 device suffers a loss of around 35-40%. Both the blocking and non-blocking MPICH-MP_Lite devices are within 5-10% of the raw TCP throughput, and deliver almost all of the performance of the MP_Lite library to the full MPI implementation.

Increasing the *Eager/Rendezvous* threshold from the default value of 128 kB to 512 kB improves the throughput curve, which otherwise shows a dip at the threshold value due to the initial handshaking by the *Rendezvous* protocol.

The MP_Lite library uses signals to check all the TCP buffers when a message arrives. The MPICH library, on the other hand, checks the TCP buffers only when the application makes a call to the message passing interface. Thus, the MP_Lite library guarantees message progress at all times. Therefore, the MPICH-MP_Lite device should prove even better for real applications than the NetPIPE measurements indicate.

Both the blocking and non-blocking versions of the MPICH-MP_Lite Channel Interface device offer the same throughput on the PC or Alpha mini-clusters. The non-blocking device would likely prove to be superior on real applications.

This MPICH-MP_Lite work should pass on the performance benefits of other MP_Lite modules like SHMEM, SMP and VIA to the full MPI implementation. The MPICH-MP_Lite devices have to be rigorously tested on real parallel applications on a variety of architectures, platforms and network hardware.

APPENDIX

CHANNEL INTERFACE ROUTINES

```
#define MPIDPATCHLEVEL 2.0
int flag;

/* Five essential functions required for implementing a channel device */

#define MPID_RecvAnyControl( pkt, size, from ) \
    { MPID_TRACE_CODE("BRecvAny",-1);\
      MP_Recv( pkt, size, -1, 0 ); *(from) = last_src;\
      MPID_TRACE_CODE("ERecvAny",*(from));}
#define MPID_RecvFromChannel( buf, size, channel ) \
    { MPID_TRACE_CODE("BRecvFrom",channel);\
      MP_Recv( buf, size, channel, channel+1 );\
      MPID_TRACE_CODE("ERecvFrom",channel);}
#define MPID_ControlMsgAvail( ) \
    ( MP_AProbe( 1000000000, -1, 0, &flag ), flag )
#define MPID_SendControl( pkt, size, channel ) \
    { MPID_TRACE_CODE("BSendControl",channel);\
      MP_Send( pkt, size, channel, 0 );\
      MPID_TRACE_CODE("ESendControl",channel);}
#if defined(MPID_USE_SEND_BLOCK) && ! defined(MPID_SendControlBlock)
/* SendControlBlock allows the send to wait until the message is received
   (but does NOT require it).  This can simplify some buffer handling.
 */
#define MPID_SendControlBlock( pkt, size, channel ) \
    { MPID_TRACE_CODE("BSendControl",channel);\
      MP_Send( pkt, size, channel, 0 );\
      MPID_TRACE_CODE("ESendControl",channel);}
#endif

/* If SendControlBlock is not defined, make it the same as SendControl */
#if !defined(MPID_SendControlBlock)
#define MPID_SendControlBlock(pkt,size,channel) \
      MPID_SendControl(pkt,size,channel)
#endif

/* Because a common operation is to send a control block, and decide
   whether to use SendControl or SendControlBlock based on whether the
   send is non-blocking, we include a definition for it here:
 */

#ifdef MPID_USE_SEND_BLOCK
#define MPID_SENDCONTROL(mpid_send_handle,pkt,len,dest) \
if (mpid_send_handle->is_non_blocking) {\
    MPID_SendControl( pkt, len, dest );}\
else {\
    MPID_SendControlBlock( pkt, len, dest );}
#else
#define MPID_SENDCONTROL(mpid_send_handle,pkt,len,dest) \
MPID_SendControl( pkt, len, dest )
#endif

/* Note that this must be non-blocking.  On systems with tiny buffers, we
   can't do this.  Instead, we use a nonblocking send, combined with tests
   for completion of the send and incoming messages. This will still
   require that the destination process the eager message, but that is one
   of the fundamental assumptions.
 */
#ifdef MPID_TINY_BUFFERS
#define MPID_SendChannel( buf, size, channel ) \
    { ASYNCSendId_t sid; \
      MPID_ISendChannel( buf, size, channel, sid );\
      while (!MPID_TSendChannel(sid)) {\
        MPID_DeviceCheck( MPID_NOTBLOCKING );\
          }\
      MPID_TRACE_CODE("ESend",channel);}
#else
#define MPID_SendChannel( buf, size, channel ) \
    { MPID_TRACE_CODE("BSend",channel);\
      MP_Send( buf, size, channel, myproc+1 );\
      MPID_TRACE_CODE("ESend",channel);}
#endif

/* Non-blocking versions (NOT required, but if PI_NO_NRECV and PI_NO_NSEND
   are NOT defined, they must be provided)
 */

#define MPID_IRecvFromChannel( buf, size, channel, id ) \
    {MPID_TRACE_CODE("BIRecvFrom",channel);\
     MP_ARecv( buf, size, channel, channel+1, id );\
     MPID_TRACE_CODE("EIRecvFrom",channel);}
#define MPID_WRecvFromChannel( buf, size, channel, id ) \
    {MPID_TRACE_CODE("BWRecvFrom",channel);\
     MP_Wait( id );\
     MPID_TRACE_CODE("EWRecvFrom",channel);}
#define MPID_RecvStatus( id ) \
    ( MP_Test( id, &flag ), flag )

/* Note that these use the tag based on the SOURCE, not the channel
   See MPID_SendChannel
 */

#define MPID_ISendChannel( buf, size, channel, id ) \
    {MPID_TRACE_CODE("BISend",channel);\
     MP_ASend( buf, size, channel, myproc+1, id );\
     MPID_TRACE_CODE("EISend",channel);}
#define MPID_WSendChannel( id ) \
    {MPID_TRACE_CODE("BWSend",-1);\
    MP_Wait( id );\
    MPID_TRACE_CODE("EWSend",-1);}
/* Test the channel operation */
#define MPID_TSendChannel( id ) \
    ( MP_Test( id, &flag ), flag )

/* If nonblocking sends are defined, the MPID_SendData command uses them;
   otherwise, the blocking version is used.
 */
```

```
#ifndef PI_NO_NSEND
#define MPID_SendData( buf, size, channel, mpid_send_handle ) \
if (mpid_send_handle->is_non_blocking) {\
    MPID_ISendChannel( address, len, dest, mpid_send_handle->sid );\
    dmpi_send_handle->completer=MPID_CMPL_WSEND;\
    }\
else \
    {\
    mpid_send_handle->sid = 0;\
    MPID_SendChannel( address, len, dest );\
    DMPI_mark_send_completed( dmpi_send_handle );\
    }
#else
#define MPID_SendData( buf, size, channel, mpid_send_handle ) \
    mpid_send_handle->sid = 0;\
    MPID_SendChannel( address, len, dest );\
    DMPI_mark_send_completed( dmpi_send_handle );
#endif

/*
   We also need an abstraction for out-of-band operations.  These could
   use transient channels or some other operation.  This is essentially
   for performing remote memory operations without local intervention.

   Note that since MPID_RecvTransfer is blocking (and may obstruct other
   messages), the chbrndv.c code that uses it calls it only after
   MPID_TestRecvTransfer succeeds.  This may be expensive in some
   applications.
 */

#define MPID_CreateSendTransfer( buf, size, partner, id ) {*(id) = 0;}
#define MPID_CreateRecvTransfer( buf, size, partner, id ) \
       {*(id) = CurTag++;TagsInUse++;}
/*
   Receive transfers may be blocking or nonblocking.  Since a single
   system may use both, there are separate definitions for the two cases.
 */
```

```
#define MPID_StartNBRecvTransfer( buf, size, partner, id, request, rid ) \
    {MPID_TRACE_CODE("BIRRRecv",id);\
     MP_ARecv( buf, size, partner, id, rid );\
     MPID_TRACE_CODE("EIRRRecv",id);}
#define MPID_EndNBRecvTransfer( request, id, rid ) \
    {MPID_TRACE_CODE("BIWRRecv",id);\
     MP_Wait( rid );\
     MPID_TRACE_CODE("EIWRRecv",id);\
     if (--TagsInUse == 0) CurTag = 1024; \
     else if (id == CurTag-1) CurTag--;}
#define MPID_TestNBRecvTransfer( request ) \
    ( MP_Test( (request)->rid, &flag ), flag )
#define MPID_CompleteNBRecvTransfer( buf, size, partner, id, rid )

#define MPID_StartRecvTransfer( buf, size, partner, id, request, rid ) \
    {MPID_TRACE_CODE("BIRRRecv",id);\
     rid = MPID_PT2PT2_TAG(id);\
     (request)->rhandle.buf = buf; (request)->rhandle.len = size;\
     (request)->rhandle.dev_rhandle.from_grank = partner;\
     MPID_TRACE_CODE("EIRRRecv",id);}
#define MPID_EndRecvTransfer( request, id, rid ) \
    {MPID_TRACE_CODE("BIWRRecv",id);\
     MP_Wait( rid );\
     MPID_TRACE_CODE("EIWRRecv",id);\
     if (--TagsInUse == 0) CurTag = 1024; \
     else if (id == CurTag-1) CurTag--;}

#define MPID_TestRecvTransfer( request ) \
    ( MP_AProbe( 100000000, (request)->from,
                            (request)->recv_handle, &flag ), flag )

#define MPID_CompleteRecvTransfer( buf, size, partner, id, rid ) \
        MPID_EndRecvTransfer( buf, size, partner, id, rid )

/* This is the blocking version */

#define MPID_RecvTransfer( buf, size, partner, id ) {\
    MPID_TRACE_CODE("BRecvTransfer",id);\
    MP_Recv( buf, size, partner, id );\
    if (--TagsInUse == 0) CurTag = 1024;\
    else if (id == CurTag-1) CurTag--;\
    MPID_TRACE_CODE("ERecvTransfer",id);}
#define MPID_SendTransfer( buf, size, partner, id ) {\
    MPID_TRACE_CODE("BSendTransfer",id);\
    MP_Send( buf, size, partner, id );\
    MPID_TRACE_CODE("ESendTransfer",id);}

#define MPID_StartSendTransfer( buf, size, partner, id, sid ) \
    {MPID_TRACE_CODE("BIRRSend",id);\
     MP_Send( buf, size, partner, id ); sid = 1;\
     MPID_TRACE_CODE("EIRRSend",id);}
#define MPID_EndSendTransfer( buf, size, partner, id, sid ) \
    {MPID_TRACE_CODE("BWRRSend",id);\
    MPID_TRACE_CODE("EWRRSend",id);}
#define MPID_TestSendTransfer( sid ) 1

#define MPID_StartNBSendTransfer( buf, size, partner, id, sid ) \
    {MPID_TRACE_CODE("BIRRSend",id);\
     MP_ASend( buf, size, partner, id, sid );\
     MPID_TRACE_CODE("EIRRSend",id);}
#define MPID_EndNBSendTransfer( request, id, sid ) \
    {MPID_TRACE_CODE("BWRRSend",id);\
    MP_Wait( sid );\
    MPID_TRACE_CODE("EWRRSend",id);}
#define MPID_TestNBSendTransfer( sid ) \
   ( MP_Test( sid, &flag ), flag )

/*
   These macros control the conversion of packet information to a standard
   representation.  On homogeneous systems, these do nothing.
 */
#ifdef MPID_HAS_HETERO
#define MPID_PKT_PACK(pkt,size,dest)
MPID_CH_Pkt_pack((MPID_PKT_T*)(pkt),size,dest)
```

```
#define MPID_PKT_UNPACK(pkt,size,src)
MPID_CH_Pkt_unpack((MPID_PKT_T*)(pkt),size,src)
#else
#define MPID_PKT_PACK(pkt,size,dest)
#define MPID_PKT_UNPACK(pkt,size,src)
#endif

/*
   On message-passing systems with very small message buffers, or on
   systems where it is advantageous to frequently check the incoming
   message queue, we use the MPID_DRAIN_INCOMING definition
 */

#define MPID_DRAIN_INCOMING \
    while (MPID_DeviceCheck( MPID_NOTBLOCKING ) != -1) ;
#ifdef MPID_TINY_BUFFERS
#define MPID_DRAIN_INCOMING_FOR_TINY(is_non_blocking) \
{if (is_non_blocking) {MPID_DRAIN_INCOMING;}}
#else
#define MPID_DRAIN_INCOMING_FOR_TINY(is_non_blocking)
#endif
```

BIBLIOGRAPHY

[1] Nieplocha, Jarek, Rik Littlefield, and Matt Rosing. "Beyond message-passing: A case for one-sided communication in MPI." In Proceedings of First MPI Developers Conference, 1995. http://www.osl.iu.edu/download/mpidc95/papers/html/nieplocha/ (date accessed: May 2002).

[2] Turner, Dave. "Introduction to Parallel Computing." Ames Laboratory. http://cmp.ameslab.gov/para_comp_intro/para_intro.html (date accessed: May 2002).

[3] K., Parzyszek, J. Nieplocha and R. A. Kendall. "A Generalized Portable SHMEM Library for High Performance Computing." Proceedings of the IASTED Parallel and Distributed Computing and Systems 2000, Las Vegas, Nevada, Nov. 2000, (M. Guizani and X.Shen, Eds.), pp. 401-406. IASTED, Calgary (2000).

[4] Nieplocha, Jaroslaw, Robert J. Harrison, and Richard J. Littlefield. "The Global Array programming model for high performance scientific computing." SIAM - News, Sep. 1995. http://www.emsl.pnl.gov:2080/docs/global/papers/siam.pdf (date accessed: May 2002).

[5] Nieplocha, Jaroslaw, Robert J. Harrison, and Richard J. Littlefield. "Global Arrays: A non-uniform memory access programming model for high-performance computers." The Journal of Supercomputing, 1996.
http://www.emsl.pnl.gov:2080/docs/global/papers/tjs.pdf (date accessed: May 2002).

[6] OpenMP, Lawrence Livermore National Laboratory.
http://www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html
(date accessed: May 2002).

[7] Sato, Mitsuhisa, et al. "OpenMP Design for an SMP Cluster." Real World Computing Partnership, Tsukuba, Japan.
http://phase.etl.go.jp/Omni/CSDSM/ewomp99/home.html (date accessed: May 2002).

[8] The MPI Forum, 1994. http://www.mpi-forum.org (date accessed: May 2002).

[9] Dongarra, Jack J., et al. "An Introduction to the MPI Standard." Univ. of Tennessee Technical Report CS-95-274, Jan. 1995.
http://www.netlib.org/utk/papers/intro-mpi/intro-mpi.html (date accessed: May 2002).

[10] MPICH Homepage. "MPICH - A Portable Implementation of MPI."
http://www-unix.mcs.anl.gov/mpi/mpich/ (date accessed: May 2002).

[11] Gropp, William, et al. "A high-performance, portable implementation of the MPI Message-Passing Interface standard." Parallel Computing, 1996.
http://www-unix.mcs.anl.gov/mpi/mpich/papers/mpicharticle/paper.html
(date accessed: May 2002).

[12] Gropp, William, and Ewing Lusk. "Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing." Summer 1997.

[13] Thakur, Rajeev. "MPICH on Clusters: Future Directions." Linux Supercluster Users Conference, Albuquerque, New Mexico, Sep. 2000.
http://spud-web.tc.cornell.edu/actc/Supercluster/presentations/Thakur.pdf
(date accessed: May 2002).

[14] Gropp, William and Ewing Lusk. "MPICH working note: The implementation of the second-generation MPICH ADI." Apr. 1996.
ftp://info.mcs.anl.gov/pub/mpi/workingnote/adi2imp.ps (date accessed: May 2002).

[15] Gropp, William and Ewing Lusk. "MPICH Working Note: The Second-Generation ADI for MPICH Implementation of MPI." May 1996.
ftp://info.mcs.anl.gov/pub/mpi/workingnote/nextgen.ps (date accessed: May 2002).

[16] Saphir, William. "A Survey of MPI Implementations." Lawrence Berkeley National Laboratory, University of California, Berkeley, CA, Nov 1997.
http://www.crpc.rice.edu/NHSEreview/MPI/ (date accessed: May 2002).

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

[18] LAM/MPI Parallel Computing. http://www.lam-mpi.org/ (date accessed: May 2002).

[19] Turner, Dave. "MP_Lite: A light weight message-passing library."
http://www.scl.ameslab.gov/Projects/MP_Lite/MP_Lite.html (date accessed: May 2002).

[20] Turner, Dave, Weiyi Chen, and Ricky Kendall. "Performance of the MP_Lite message-passing library on Linux clusters." In Linux Clusters: HPC Revolution, 2001.

[21] Chen, Weiyi. "Implementation of MP_Lite for the VI Architecture." Master's thesis, 2001, Dept. of Computer Science, Iowa State University, Ames, 2001.

[22] Chimp: ftp://ftp.epcc.ed.ac.uk/pub/chimp/release/ (date accessed: May 2002).

[23] MPI/PRO: http://www.mpi-softtech.com/ (date accessed: May 2002).

[24] Harrison, Robert J. "The TCGMSG Message-Passing Toolkit." http://www.emsl.pnl.gov:2080/docs/nwchem/tcgmsg.html (date accessed: May 2002).

[25] Crawford, Emily Angerer. "PVM: An Introduction to Parallel Virtual Machine." Oct. 1996. http://www.hpc.gatech.edu/seminar/pvm.html (date accessed: May 2002).

[26] Vaughan, Paula L., et al. "Migrating from PVM to MPI, part I: The Unify System." The Fifth Symposium on the Frontiers of Massively Parallel Computation, McLean, Virginia, Jul. 1994.

[27] MVICH: http://www.nersc.gov/research/FTG/mvich/ (date accessed: May 2002).

[28] Dimitrov, Rossen, and Anthony Skjellum. "Efficient MPI for Virtual Interface Architecture." 1998.

[29] Worringen, Joachim, and Thomas Bemmerl. "MPICH for SCI-connected Clusters." In Proceedings of SCI - Europe '99, Toulouse, Sep. 1999, pages 3 - 11.

[30] Cristaldi, Rosario, and Giulio Iannello. "MPI derived data types in VIRTUS." In Proceedings of CANPC '00, Toulouse, France, Jan. 2000.

[31] Chiola, Giovanni, and Giuseppe Ciaccio. "Porting MPICH ADI on GAMMA with Flow Control." In Proceedings of MWPP '99, Kent, Ohio, Aug. 1999.

[32] Brightwell, Ron, and Lance Shuler. "Design and Implementation of MPI on Puma Portals." In Proceedings of the Second MPI Developer's Conference, Jul. 1996, pages 18 - 25.

[33] Schindler, Sven, and Wolfgang Rehm. "Multiple devices under MPICH." In Proceedings of the workshops ARCS '99, Oct. 1999.

[34] Brightwell, Ronald, and Anthony Skjellum. "Design and Implementation Study of MPICH for the Cray T3D." Technical report, Computer Sci. Dept., Mississippi State Univ, 1996. http://www.cs.msstate.edu/~tony/documents/Message-Passing/mpich_t3d_paper.pdf (date accessed: May 2002).

[35] Foster, Ian, et al. "Wide-Area Implementation of the Message Passing Interface." Parallel Computing, 1998.

[36] O'Carroll, Francis, et al. "Design and Implementation of Zero Copy MPI for PM." Technical Report TR – 97011, RWC, Mar. 1998.

[37] Husbands, Parry, and James C. Hoe. "MPI-StarT: Delivering Network Performance to Numerical Applications." In SC '98, Nov. 1998.

[38] Aumage, Olivier, Guillaume Mercier, and Raymond Namyst. "MPICH/Madeleine: a True Multi-Protocol MPI for High Performance Networks." Oct. 2000.

[39] Turner, Dave and Xuehua Chen. "Protocol-Dependent Message-Passing Performance on Linux Clusters". Submitted to the Cluster 2002 conference.

[40] NetPIPE homepage. http://www.scl.ameslab.gov/netpipe/ (date accessed: May 2002).

[41] Snell, Quinn O., Armin R. Mikler, and John L. Gustafson. "Netpipe: A network protocol independent performance evaluator." Scalable Computing Laboratory/Ames Laboratory, 1997.