RICE UNIVERSITY

High Performance MPI Libraries for Ethernet

by

Supratik Majumder

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Approved, Thesis Committee:

Scott Rixner, Chair
Assitant Professor of Computer Science
and Electrical and Computer Engineering

Alan L. Cox
Associate Professor of Computer Science
and Electrical and Computer Engineering

John Mellor-Crummey
Associate Professor of Computer Science
and Electrical and Computer Engineering

Houston, Texas

September, 2004

ABSTRACT

High Performance MPI Libraries for Ethernet

by

Supratik Majumder

A MPI library performs two tasks—computation on behalf of the application, and communication in the form of sending and receiving messages among processes forming the application. Efficient communication is key to a high-performance MPI library, and the use of specialized interconnect technologies has been a common way to achieve this goal. However, these custom technologies lack the portability and simplicity of a generic communication solution like TCP over Ethernet.

This thesis first shows that even though TCP is a higher overhead protocol than UDP, as a messaging medium it performs better than the latter, because of library-level reliability overheads with UDP. Then, the thesis presents a technique to separate computation and communication aspects of a MPI library, and handle each with the most efficient mechanism. The results show a significant improvement in performance of MPI libraries with this technique, bringing Ethernet closer to the specialized networks.

Acknowledgments

I express my sincere gratitude to my advisors, Prof. Scott Rixner and Prof. Vijay S. Pai, for their guidance through this research over the past one and a half years. Their scholarship, support, patience, and diligence will always motivate me. I especially thank Prof. Scott Rixner for so painstakingly going through this thesis on multiple occasions. Without his comments and constant persuasion, this thesis certainly would not have completed.

I thank the other members on my thesis committee, Prof. Alan L. Cox and Prof. John Mellor-Crummey, for their time and valuable comments on this thesis. Richard Graham at Los Alamos National Laboratory (LANL) facilitated this research by providing access to the LA-MPI library, and valuable guidance on several occasions.

I also thank the other members of my research group, especially John and Paul, for their invaluable technical help at various stages throughout this research.

Finally, I thank my parents for their love and support throughout.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Message-passing is a paradigm for parallel programming wherein processes forming the message passing application communicate with each other exclusively through the use of messages. This paradigm of programming can be used on clusters of networked computers and shared memory multiprocessor machines. The communication, unlike in the shared memory programming paradigm, is usually producer driven, and always explicit. All processes operate in their own address spaces and can choose an arbitrary granularity for communicating data—a process might choose to communicate data which is just one byte long or it might choose to communicate data several megabytes in size. These attributes of the message-passing paradigm make it a very popular choice for parallel programming, especially on networked computer clusters. However, message passing requires considerable support in the form of user-level software libraries to manage messages and their communication between parallel processes. The Message Passing Interface (MPI) standard, formulated in the early 1990s, has served as the de-facto standard for message-passing libraries over the past decade. The MPI standard facilitates portability of message-passing applications across different hardware platforms [14]. This standard provides a rich and powerful application programming interface (API) to message-passing applications, leaving all implementation details out of the library specification. This allows library developers to provide the best possible implementations for their respective hardware platforms.

MPI applications, like other parallel applications, perform two distinct tasks—

computation and communication. The computation task is usually performed by the application directly, whereas the MPI library supports communication between the application's processes. Thus, the overall performance of a MPI application, depends as much on individual nodes' computation power, as on the communication substrate used and the library support available for communication over that substrate. As the computation power of individual nodes has increased with faster processors over the past several years, the focus of attention for improving MPI performance on workstation clusters has gradually shifted towards the communication medium and the MPI library.

Traditionally, MPI library researchers, both in academia and in industry have sought to provide high-performance MPI communication through the use of specialized interconnect technologies. The development of communication technologies, such as Quadrics and Myrinet are prime examples of this trend [4, 33]. Both of these use custom hardware and user-level protocols to provide a low-latency, high-bandwidth communication substrate for MPI. However, more than their hardware itself, their software support is what enables high-performance communication. MPI library implementations usually also support messaging on commodity interconnects such as Ethernet. For this purpose the library utilizes one of the transport layer protocols of the operating system network protocol stack—Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) [39]. Communication overheads, such as data copying between user-space and kernel-space, CPU overhead of TCP protocol processing, and acknowledgment-related interrupt processing, cause a significant increase in messaging latency with TCP/UDP over Ethernet, as compared to the specialized networks. On the other hand, user-level custom protocols used by the specialized networks eliminate almost all operating system involvement in enabling MPI communication, which gives them a performance advantage over commodity solutions, such as TCP and Ethernet.

In spite of these drawbacks of using Ethernet as a messaging substrate in MPI, it has a big advantage of being ubiquitous, highly portable and extremely robust. Furthermore, it is relatively inexpensive compared to the specialized solutions. These reasons make Ethernet an attractive alternative, even though the existing Ethernet support in MPI libraries does not make it a viable alternative, especially for high-performance message-passing.

This thesis attempts to improve the Ethernet support in existing MPI libraries and boost the performance of MPI applications over an Ethernet connected cluster. In this process, the thesis aims to show that TCP messaging over Ethernet can be a viable alternative to the specialized networks. One common perception among the MPI community is that the overheads associated with TCP—connection management, reliability of communication, and flow-control—make it unfit to be a messaging substrate for MPI libraries. Consequently, several MPI implementations have provided Ethernet support through the use of UDP datagrams for message communication. First, this thesis shows that this perception is generally wrong, especially for a well designed TCP messaging layer in the MPI library. The overhead of adding reliability and flow-control mechanisms at the library-level more than offsets the gain from using UDP, rather than TCP, as the transport protocol. A well designed TCP path is shown to outperform UDP, in both library message latency and message bandwidth, as well as overall application performance. With UDP messaging, the Los Alamos MPI (LA-MPI) library is only able to utilize a maximum of 50% of available Gigabit Ethernet bandwidth, at all message sizes above 1 KB. The results presented in this thesis show that the use of TCP as the messaging layer enables the same library to attain line-rate bandwidth for as small as 8 KB messages.

This thesis then proposes to further refine the TCP messaging support in the library by handling communication more efficiently. As already stated, MPI applications perform two basic tasks—computation and communication. Most MPI libraries (and all freely available ones) handle both with the same mechanism, essentially piggybacking communication tasks with computation tasks. This is an inefficient way to handle TCP communication in a MPI library. This thesis proposes the separation of these two tasks of the MPI library, and handling each with the best possible mechanism. Thus, the communication tasks are offloaded onto an independent thread in the library, which performs all network-related tasks in the library. This thread is designed to function in an event-driven fashion, similar to the approach followed by high-performance network servers [5, 32]. On the other hand, the computation tasks are performed by the MPI application executing in the context of the main thread of the library. Having two independent threads in the library enables greater concurrency between the computation and communication tasks in the library and significantly improves the performance of non-blocking communication primitives. The results show that the wait times of non-blocking requests are significantly reduced, and even entirely eliminated in some cases. Although this technique causes a slight increase in library overhead due to thread switching and lock contention, for benchmark MPI applications, the gains from increased library responsiveness to communication events, and increased concurrency between communication and computation, more than offsets the overhead. Furthermore, the results also show that the event-driven MPI library is more scalable with the number of nodes in the cluster, because the relative increase in the communication component of the application (with increasing number of nodes) extracts greater benefits from the event-driven communication mechanism.

Finally, the thesis presents a comparison of MPI library performance on Gigabit Ethernet and Myrinet of comparable bandwidth. Due to the user-level communication employed by Myrinet, the MPI message latency with Myrinet is up to 5 times lower than that with Ethernet. Message bandwidth with Myrinet is slightly higher than with Ethernet on account of its higher maximum bandwidth. However, the latency and bandwidth advantage of Myrinet does not directly translate into considerable superior application performance. The event-driven MPI library is able to match or outperform Myrinet on several of the benchmarks tested, showing that an efficient communication handling mechanism in the MPI library can go a long way in improving overall application performance. Benchmarks having greater overlap between computation and communication enjoy a higher performance improvement with the event-driven library. Such benchmarks show less performance disparity between Myrinet and Ethernet, and can even perform better with Ethernet, when there is sufficient overlap of computation and communication.

The main advantage enjoyed by the Myrinet MPI libraries is their use of user-level communication protocols, which effectively eliminate several of the overheads associated with TCP communication, such as data copying between user-space and kernel space, interrupt processing and CPU overhead of the operating system network protocol stack. With zero-copy sockets and offloading TCP processing (or a part of it) onto an Ethernet network interface, many of these overheads can be successfully eliminated [3, 15, 35]. These TCP optimizations, when used together with the event-driven communication model described in this thesis, can make TCP messaging over Ethernet a low-cost, portable, and reliable alternative to specialized networks for high-performance message-passing applications.

1.1 Contributions

In summary, the main contributions of this thesis are as follows.

- This thesis disproves a common perception that a low overhead protocol such as UDP performs better than TCP for MPI library communication, especially when UDP messaging has to be augmented by library-level reliability mechanisms.

- This thesis proposes and implements a technique to separate the computation and communication aspects of a MPI library and handle each of them with the most efficient technique, thereby improving overall performance of the library for message-passing applications.

- Finally, this thesis shows that with an efficient TCP communication handling mechanism in the MPI library, the application performance gap between Ethernet and Myrinet messaging can be reduced, and even reversed in some cases.

1.2 Organization

This thesis is organized as follows. Chapter 2 provides a high-level overview of the MPI standard and describes two publicly available implementations of the standard—the MPICH MPI library and the Los Alamos MPI (LA-MPI) library [20, 19]. It also gives a summary of the main benchmarking techniques employed for evaluating and comparing messaging performance of MPI libraries and message-passing systems.

Chapter 3 provides a detailed description of Ethernet messaging support in the LA-MPI library using UDP datagrams. It also proposes the design for a TCP messaging layer for the library, to replace UDP as the message transport layer over Ethernet. By using the reliability features of the TCP protocol itself, the reliability mechanisms of the library can be turned off, thereby yielding improvements in message latency, bandwidth, and overall MPI application performance.

Chapter 4 proposes a technique to separate the computation and communication aspects of a MPI application, and handling each by the most appropriate technique. Thus, the computation tasks of the application are handled in a synchronous manner, as in any conventional MPI library. The communication tasks are offloaded to an independent thread of the library, which is modeled after the event-driven architecture followed in high-performance network servers. This technique can increase the concurrency between a MPI application's computation and communication, leading to improved performance of non-blocking MPI communication. Depending upon the design of the MPI application, this can considerably improve its overall performance as well.

Chapter 5 presents a comparison of the event-driven TCP messaging layer for the

LA-MPI library against the Myrinet messaging layer. Gigabit Ethernet, even though substantially higher latency than Myrinet, as well as lower bandwidth, is shown to deliver application performance competitive with Myrinet, when used with an event-driven TCP messaging layer to perform communication over Ethernet.

Chapter 6 discusses several network communication intensive applications, such as network servers, that have efficiently used the event-driven communication model to provide higher performance. It also discusses the use of threading as a means to improve concurrency of different tasks in an application. Furthermore, the chapter discusses the communication handling mechanisms of several other MPI libraries, both free and commercial. The chapter ends with an overview of some other research efforts focused on improving MPI application performance on clustered systems, including the development of specialized networks for MPI communication.

Finally, Chapter 7 concludes that TCP over Ethernet is a viable option for supporting high-performance message-passing applications. With TCP optimizations such as copy avoidance, interrupt avoidance, and TCP offloading, TCP messaging over Ethernet can deliver performance competitive with that obtainable using more costly specialized networking technologies such as Myrinet.

Chapter 2
Background

The message-passing model of parallel programming requires extensive support in the form of user-level software libraries to manage messages and their communication between parallel processes. The Message Passing Interface (MPI) Standard provides an API specification for these message-passing libraries, excluding all implementation details. This ensures the independence of MPI applications from the individual quirks of system-dependent message-passing libraries. Since the formulation of the MPI standard, there have been several revisions to it with an aim to provide an even richer programming interface for developing message-passing applications.

There have been several different research (and industry) efforts aimed at developing high-performance MPI libraries. All of these libraries conform to one or the other version of the MPI standard and are thus, similar to one another in lot of respects. However, the various MPI implementations differ in the ways they handle communication in the library, and in the level of support they have for different interconnect technologies, which lead to significant performance variations. MPICH and LA-MPI are two freely available MPI libraries with differing software architectures and consequently substantially different performance [20, 19]. From a MPI application's point of view, one of the key determinants of its overall performance is the performance of the library's point-to-point communication primitives. Thus, the most basic message-passing primitives—message send and message receive—provide valuable insight into the expected overall performance of a MPI application (library) and the potential bottlenecks.

The two most commonly used criteria for evaluating (and comparing) the performance of a MPI library are message latency and message bandwidth. These are commonly measured by MPI microbenchmarks involving just point-to-point communication between two nodes. Since the bandwidth performance of a MPI library varies significantly with message size, the distribution of message sizes used by the application is another factor that influences overall application performance. MPI supports both blocking and non-blocking modes of communication. The non-blocking mode enables greater overlap between an application's MPI communication and its computation, typically resulting in better performance. On the other hand, message latencies are shortest with the use of blocking communication primitives. Thus, the extent to which an application uses one or the other mode of communication also has significant bearing on its overall performance, depending upon which mode of communication is better supported by the MPI library. In view of the variety of factors that determine MPI application performance, simple microbenchmarks are often not enough to accurately assess the performance capability of a MPI library. As a result, MPI library researchers and users also use several MPI application benchmarks, which are typically scaled down versions of real world MPI problems. These benchmarks are run on a much larger message-passing system involving several nodes, and usually provide a more accurate estimate of an application's performance with a particular MPI library.

The rest of this chapter is as follows. Section 2.1 introduces the MPI standard and highlights some of its key features. Section 2.2 discusses in brief the transport layer protocols used to support message-passing on Ethernet networks. Section 2.3 describes the MPICH message-passing library, its software architecture, and the basic anatomies of the message send and receive operations on Ethernet as implemented by this library. Section 2.4 provides the same details for the LA-MPI library. Commonly used MPI benchmarks and their properties are discussed in Section 2.6. Finally, Section 2.7 summarizes the chapter.

2.1 MPI Standard

The MPI standard was formulated by a broad collaboration of vendors, implementors, and users of message-passing systems [14]. The MPI Standard concerns itself only with the library API semantics and leaves the implementation details to library developers. The absence of implementation details from the standard itself has also served to satisfy another stated objective of the MPI standard—to achieve maximum portability across a range of hardware platforms and interconnect technologies. MPI library developers exploited this key attribute of the MPI standard to customize implementations of the library for a variety of conditions. On one hand, MPI libraries support high-performance message-passing with custom protocols and specialized networking technologies such as Quadrics and Myrinet [4, 33]. On the other hand, MPI libraries also support generic communication protocols, such as TCP/UDP, and commodity networking hardware, such as Ethernet, to achieve maximum portability across a range of hardware platforms. The emphasis on keeping implementation details out of the standard has also ensured that the standard itself has not been outmoded by significant advancements in interconnect technology and communication protocols. A more comprehensive list of goals for the MPI Standard can be found in the draft of the standard [14].

The initial version of the MPI Standard specified library routines for the following classes of operations:

- **Point-to-Point Communication** – This set of MPI library routines defines the most basic pair-wise communication operations of MPI—the send and receive functions, which are used for sending and receiving a message, respectively. The simplest versions of the MPI send and receive functions are the blocking versions, `MPI_Send` and `MPI_Recv`, respectively. This set also contains some other associated functions and variants of the basic send and receive functions such as for non-blocking communication, buffered communication, etc., all of which are designed to make MPI communication more powerful and efficient.

- **Collective Operations** – This set of MPI routines defines collective communication operations, that is, operations that a process group in a MPI application executes as a whole. These operations can have an arbitrary number of participants, the number of which is only limited by the specific MPI library implementation. Also, note that a process group need not necessarily be composed of all the processes in the MPI application. Some common examples of functions in this group are the `MPI_Broadcast` and `MPI_Barrier` routines.

  In practice, several MPI implementations develop these collective operations on top of the point-to-point communication layer.

- **Groups, Contexts and Communicators** – MPI provides a mechanism for treating a subset of processes as a "communication" universe. This mechanism is the *communicator*. MPI provides two different kinds of communicators—intra-communicators and inter-communicators. Intra-communicators support point-to-point and collective operations within a set of processes. Inter-communicators, on the other hand, are used for sending messages between processes belonging to disjoint intra-communicators. A *context* is a property of communicators used to partition the communication space of the MPI application. Thus, a context is a property to establish communication boundaries between processes: a message sent in one context cannot be received in another context. A *group* is an ordered set of MPI processes; each process in a group has a unique integer MPI process identifier known as the *rank*. Within a communicator a group is used to define and rank its participants.

This set of MPI routines provides the ability to form and manipulate groups of processes, obtain unique communication contexts and bind the two together to form a MPI communicator.

- **Process Topologies** – Topology in MPI is an extra, optional attribute that can be given to an intra-communicator. It provides a convenient naming mechanism for the processes of a group and may also assist the runtime system in mapping the processes onto hardware. This is specifically used for situations in which the usual linear process identifiers in a group do not serve the objective of addressing processes in that group satisfactorily. In other words, process topologies in MPI reflect logical process arrangement. Note that the process topologies in MPI are strictly *virtual*, and there is no simple relation between the process structure defined by the topology and the actual underlying physical structure of the parallel machine.

This set of routines provides the utilities for the creation and management of process topologies.

- **MPI Environmental Management** – These MPI functions provide the ability to the MPI library user to get and, where appropriate, set various parameters relating to the MPI implementation and execution environment, such as error handling. A couple of important functions belonging to this set are `MPI_Init` and `MPI_Finalize`, which are procedures for entering and leaving the MPI execution environment, respectively.

Besides specifying the above classes of operations, and providing the various MPI functions, data types, and constants, the MPI Standard also contains the following:

- **Bindings for Fortran 77 and C** – These language bindings, give specific syntax for all MPI functions, constants, and types in Fortran 77 and C.

- **Profiling Interface** – This explains a simple name-shifting convention that any MPI implementation is supposed to support. One of the motivations for having such a feature was to enable performance profiling of MPI applications without the need for access to the MPI library source code.

The subsequent revisions of the MPI standard have gone beyond the sets of operations defined by the initial standard, and progressively tried to incorporate more parallel programming features into MPI. Some of these features include support for parallel I/O, one-sided MPI communication, and process creation and management.

2.2 Ethernet Messaging

Across a broad spectrum of network applications, the two protocols of choice for communication over Ethernet are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) [39]. These are transport layer protocols layered on top of a network layer protocol, the Internet Protocol (IP). Depending on the requirements of the application itself, one or the other transport protocol is usually more appropriate for it. The support for Ethernet messaging in MPI libraries is also provided through the use of one of these two transport layer protocols.

TCP is a connection oriented protocol, and supports stream-based reliable communication. UDP, on the other hand, is a connection-less protocol, and uses unreliable datagrams for communicating data between nodes. The TCP and UDP protocols, as well as IP, are implemented as a part of the operating system network protocol stack. They interact with a user application through the socket interface.

The TCP protocol incorporates a number of reliability and flow-control mechanisms to support a stream-based reliable communication between nodes. These mechanisms include segmenting outgoing data and requiring per-segment acknowledgments from the receiver node. Also, the protocol has to perform connection management tasks to provide the notion of a connection over an inherently connectionless protocol like IP. These features of the TCP protocol, while providing valuable service to the networking applications, increase the operating system overhead of communication. The UDP protocol provides none of these features, and hence, results in significantly less operating system overhead.

2.3 MPICH

MPICH is a freely available, complete implementation of the MPI Standard 1.2 specification, developed at Argonne National Laboratory (ANL) [20]. It is designed to be both portable and efficient. The version of the MPICH library described in this section is 1.2.5.

The MPICH implementation supports a variety of different hardware platforms. On one hand, support is extended to distributed-memory parallel supercomputers, in which each process runs on a node of the machine, and different processes communicate through a high-performance network. Some examples of this category of machines includes the Intel Paragon, the IBM SP2, and the Cray T3D. On the other hand, the library also supports machines with hardware support for shared memory. The MPICH implementation on this kind of hardware platform provides particularly efficient message-passing operations by utilizing the shared-memory programming model. Some examples of this kind of hardware platform are the SGI Oynx, the SGI Origin, the SGI Altix, the Challenge, and the Power Challenge. Besides these two varieties of predominantly popular supercomputing platforms, the MPICH library also supports a parallel computing environment on an Ethernet-connected network of workstations. The interoperability of different kinds of workstations is provided by the use of TCP/IP for message-passing. Besides Ethernet, later versions of the library have also included support for the Myrinet interconnect technology.

The development of MPICH, at least initially, borrowed heavily from two precursor systems for parallel programming and message-passing on parallel supercomputers—*p4* and *Chameleon*. p4 is a highly developed and portable parallel programming

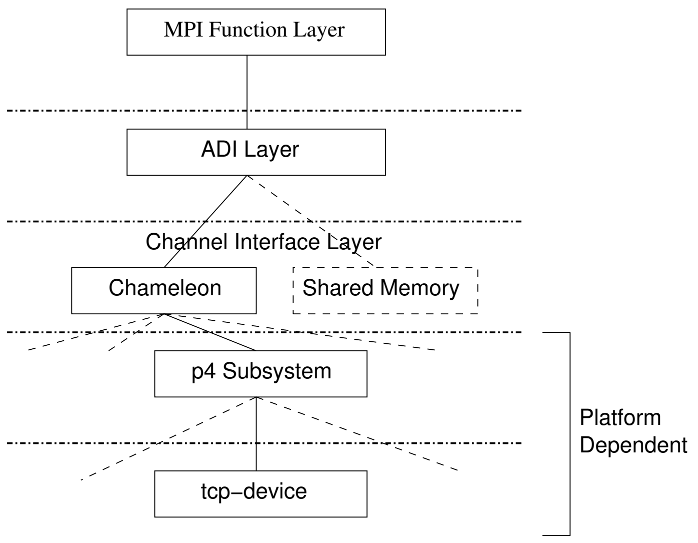

Figure 2.1 Software layering of the MPICH-1.2.5 library

library having both message-passing and shared-memory components [7]. It supports a multitude of parallel computing environments, including heterogeneous networks. To support heterogeneous networks, this library utilizes the TCP/IP protocol stack over an Ethernet network. The MPICH library support for networked workstations connected by Ethernet is based in principle on the Ethernet and TCP/IP support provided by the p4 library. Chameleon is a portable high-performance library for message-passing on supercomputers [21]. It is implemented mainly as a thin layer of macro definitions over other message-passing systems. These include both vendor specific systems, such as Intel's NX, IBM's MPL, and other publicly available systems, such as p4. The MPICH implementation also borrows significantly from the implementation techniques of Chameleon.

2.3.1 Architecture of MPICH

Figure 2.1 shows the hierarchy of software layers employed by the the MPICH library. The topmost layer is the MPI function layer, which provides the API for the MPI standard version 1.2 specification to the user application. All MPI functions are implemented in terms of macros and functions that comprise the layer just below this—the *abstract device interface* (ADI) layer. The ADI is a key mechanism used by the MPICH library to satisfy its goals of performance and portability. The MPICH ADI even though quite complex in its design, essentially provides four sets of functions: specifying a message to be sent or received, moving data between the API and message-passing hardware, managing lists of pending messages, and providing basic information about the execution environment. Thus, the ADI in MPICH is responsible for packetizing messages and attaching header information, managing multiple buffering policies, matching posted receives with incoming messages and queuing them if necessary, and handling heterogeneous communications.

Table 2.1 Data exchange protocols in MPICH

Protocol	**Semantics of Data Exchange**
Eager	Data is sent to the destination immediately.
Rendezvous	Data is sent to the destination only when explicitly permitted by the destination.
Get	Data is directly read by the receiver. However, this protocol does require a method to directly transfer data from one process's memory to another's.

As shown in Figure 2.1, the ADI itself is implemented in terms of yet another lower level interface known as the *channel interface*. This layer, in its simplest form, provides the most basic of message-passing functionalities—a way to transfer data from one process's address space to another's. However, in providing this basic functionality, this layer has to deal with the issue of buffering data, both for outgoing and incoming messages. This layer itself does not provide any buffering, leaving it to the platform-dependent layers below it, but it does implement three different data exchange protocols. Table 2.1 shows the data exchange protocols implemented by the channel interface layer of the MPICH library. Since network communication can block at any time due to a lack of resources, the channel interface also provides a mechanism to wait for the availability of resources. This mechanism is known as the *progress engine* of the library. The progress engine ensures that in case a request (send/receive) cannot be completed immediately, the other pending requests in the library get a fair chance to make progress. MPICH includes multiple implementations of the channel interface, both vendor provided implementations for specific hardware platforms and generic (and portable) implementations including Chameleon and shared memory.

In the MPICH library, messaging support for using TCP over Ethernet is provided by the Chameleon channel interface implementation. The Chameleon layer is in turn implemented by the p4 subsystem (Section 2.3). For TCP messaging, the p4 layer provides the low level functionalities, such as establishing connections between nodes of the workstation cluster, maintaining the state of these connections, and moving data into and out of the socket buffers. The actual buffering of data (whether incoming or outgoing) is also performed by this layer. As mentioned earlier, the p4 subsystem is also a highly portable parallel messaging library, and the *TCP device* is just one of the components of this subsystem. This device provides the necessary TCP messaging support to the upper layers of the MPICH library. The TCP device interacts directly with the protocol stack of the operating system through the socket API. The TCP device also implements a progress engine of the MPICH library—it is a simple loop polling all open socket connections for incoming data and write space on socket buffers.

2.3.2 Anatomy of Typical MPI Operations

The `MPI_Send` library call instructs the library to perform a blocking message send operation to a peer process. As described in Section 2.3.1, the API layer functions are implemented in terms of the macros and functions of the ADI layer. Thus, the `MPI_Send` function call results in the corresponding send function of the ADI layer being invoked. The communicator and the datatype of the message are resolved at the ADI layer. Based on the size of the message and user specification, the data transfer protocol (eager/rendezvous/get) is also resolved at this layer. The channel interface provides different message send functions for each of the data transfer protocols, and one of these is invoked from the ADI layer based on the protocol selection. For TCP messaging, the corresponding Chameleon layer send function is invoked. The Chameleon layer in MPICH is implemented in terms of the p4 subsystem, which in turn consists of various supported communication devices, among which TCP messaging is enabled by the TCP device (Section 2.3.1). The send function call of the Chameleon layer is progressively translated into lower layer functions based on the communication device supported by the system, and finally a device specific send routine is invoked at the lowermost layer to transfer data from the library to the operating system networking stack (using the Unix `write` system call). This routine first creates a header for the message being sent. This header is sent to the message destination and informs it about the size of the message. Immediately following this, the message itself is also sent to the destination. The MPI send operation is complete as soon as all the data has been copied out of the user buffer and into the socket buffer. Thus, the completion of the send also implies that the application can free its own message buffer. The reliability of communication provided by TCP ensures that the message, once copied out of the user buffer on the sender, will eventually reach the library at the destination.

Message receives in the MPICH library are slightly more involved than message transmits because the library has to receive both expected and unexpected messages. An expected message is a message whose receive has already been started (posted) by the user application before the message itself arrives. Thus, when the message arrives at the library on the receiver node, it is already expected, and can be copied directly to the user buffer. On the other hand, if the application does not post a message receive prior to its actual reception by the library, the message is termed as an unexpected receive. The unexpected receives, especially those sent by the eager protocol of the channel interface have to be buffered at the p4 layer of the library, and are matched with later receive postings by the user application. To aid in managing expected and unexpected receives, the MPICH library maintains two queues of messages—the posted (expected) receives queue, and the unexpected receives queue. All unexpected messages are put on the unexpected receives queue. All receives posted by the application, whose corresponding messages have not yet arrived, are placed on the posted receives queue.

The `MPI_Recv` function call starts a blocking message receive operation. As in the case of message transmission, the MPI function calls the corresponding receive routine of the ADI layer. In this ADI layer routine, the queue of unexpected receives is first checked to find a match for the posted receive. If a match is made, the message is copied from the library buffer to the user buffer, and the MPI receive call is completed. Otherwise, the receive request is added to the queue of posted receives, and waits for the corresponding message to arrive on the network interface. This process of waiting is accomplished by the progress engine component of the channel interface layer of the library. For TCP messaging, in the event of incoming data, the progress engine invokes the TCP device specific receive routine through the p4 subsystem macros. This function makes use of the `read` system call to read data off the socket buffer. Analogous to message send behavior, first the header of the message is read, which instructs the TCP device how much data to expect in the message. Then the message itself is read into a library buffer from the socket and matched against the receives present in the posted receives queue. If a match is found the message is copied to the user posted message buffer, and the message receive operation is completed. Otherwise, this newly arrived message is added to the queue of unexpected receives, and control is returned to the progress engine to wait for the next incoming message.

2.4 Los Alamos MPI

Los Alamos MPI (LA-MPI) library, developed at the Los Alamos National Laboratory (LANL), is another freely available and complete implementation of the MPI Standard 1.2 specification [19]. It is unique among other MPI implementations in that it provides an end-to-end failure-tolerant message-passing system. Thus, the library provides immunity to the MPI application from errors such as I/O bus, and wire transmission errors. The library is also designed to survive catastrophic failures such as a network card failure, and guarantees the delivery of in-flight messages in the presence of one. The LA-MPI library supports the concurrent use of heterogeneous network interfaces, and message striping across them between two communicating MPI nodes. The version of LA-MPI described in this section, unless otherwise mentioned, is version 1.3.x*.

The LA-MPI library also supports communication over a variety of interconnect hardware, such as Ethernet, Myrinet, and Quadrics, in addition to shared memory. The interconnect hardware dictates the use of different software communication protocols. For Ethernet, the LA-MPI library uses UDP datagrams, and thereby relies on the network protocol stack of the operating system available through the socket interface. For the other hardware interconnects, the library uses operating system bypass protocols for communication between the MPI nodes. This has an added benefit of reducing latency of MPI communication for these specialized networks. In every case, reliability of communication can be provided directly by the library through the use of library level acknowledgments. The library also provides flow-control through message fragmentation, and a simple sliding window mechanism.

LA-MPI supports such varied communication hardware (and protocols) through the use of an abstraction of the physical communication channel, also known as *path*. This abstraction provides all the services required of a communication medium in a MPI library implementation. A path controls access to one or more network interfaces cards (NICs), and is responsible for orchestrating all MPI communication passing through those NICs. In particular, every path provides functions to send and receive messages, interact with the rest of the library, and provide reliability and flow-control. Every communication channel is described in terms of this abstraction. This policy

*x is not meant to be a wild-card, the VERSION file in the LA-MPI library source mentions 1.3.x helps keep the other portions of the library relatively free of hardware specifics. In addition, it also reduces the effort required to add support for a new path in LA-MPI.

2.4.1 Architecture of LA-MPI

The architecture of LA-MPI follows a layered approach similar to the one used by MPICH. The library is broadly composed of three almost independent layers. The top layer is known as the *MPI Interface Layer*, and is responsible for providing the MPI API functions to the user application. This interface layer is compliant with the MPI Standard version 1.2 specification.

The middle layer is known as the *Memory and Message Layer* (MML), and forms the heart of the library. This layer is responsible for managing all library resources for sending and receiving messages, keeping track of the messages themselves, and selecting an appropriate path for the message to be sent on (or received on). This layer manages all the memory requirements of the library, both process private memory and shared memory. In addition, for the specialized networks, for which the library provides operating system bypass communication protocols, this layer also manages the NIC memory. All memory is managed in the form of various memory pools, from which buffers of different types are allocated on demand, and subsequently freed back into the same memory pool. This strategy ensures optimum buffer reuse, and reduces the overhead of operating system calls to manage memory allocations and freeing. Another part of this layer is the path scheduler, which is responsible for selecting an appropriate path for communication with the MPI peer node, and binds every outgoing message to a specific path. Thus, the path scheduler can schedule different messages between the same pair of communicating peers to use different available paths, thereby enabling message striping across multiple heterogeneous network interfaces. The reliability of communication in the library is also provided by the MML in tandem with the path implementations through the use of library-level acknowledgments and CRC checksums. This facilitates the reliable delivery of messages in the presence of I/O bus, network card, and wire-transmissions errors. Resilience to network device failures is achieved as a property of the path scheduler. Another component of the MML is the main progress engine of the library. This is implemented as a simple loop, which iterates over all pending requests (send/receive) in the library, and tries to make maximum possible progress on each of them.

The lowest layer (farthest from the MPI application) in the LA-MPI library is known as the *Send and Receive Layer* (SRL), and is composed of the different path implementations. This layer is responsible for fragmenting (and reassembling) messages, and sending (and receiving) these fragments. Similar to MPICH, this layer also implements different data transfer protocols based on the size of the message. For the UDP path, the LA-MPI library provides two data transfer protocols—*eager* and *rendezvous* (see Table 2.1). The LA-MPI library distinguishes between on-host and off-host message fragments and follows different message transfer schemes for the two. For on-host message fragments, simple copies are used through shared memory. For off-host messages, the available network is used, as chosen by the path scheduler.

2.5 Anatomy of Typical MPI Operations

Figure 2.2 shows the steps of a typical off-host point-to-point message-passing in LA-MPI with UDP over Ethernet. The overall architecture of LA-MPI library is also shown and the MML layer is shown expanded with the steps involved in point-

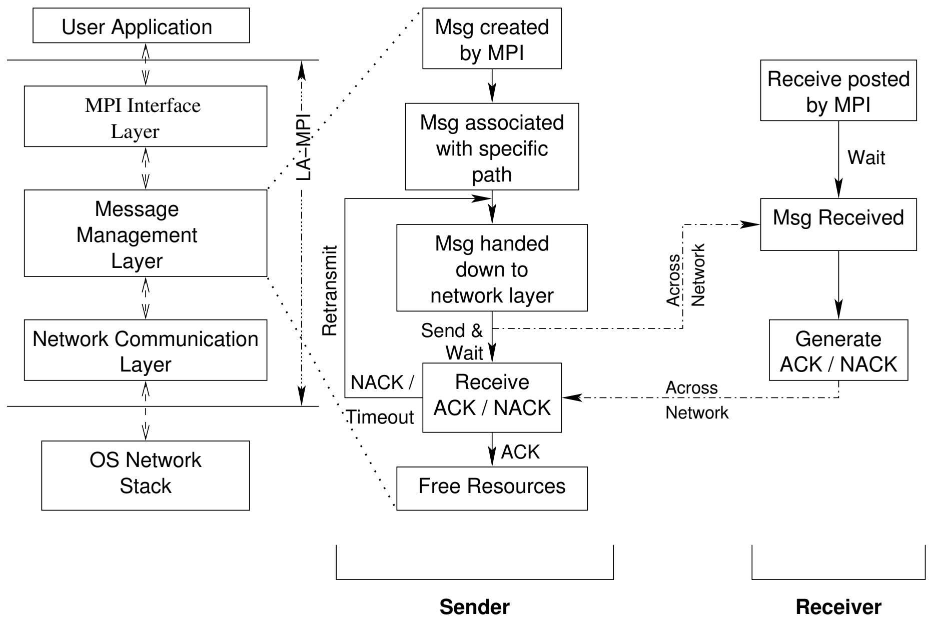

Figure 2.2 Point-to-point off-host communication in LA-MPI

to-point off-host communication. As shown in the figure, the user application calls into the library through the MPI Interface Layer. This layer performs some basic error checking on the parameters passed to the library by the user application. The interface layer calls into the MML.

For blocking MPI message send, the MML initially allocates some resources, and creates the message from the message data specified by the user. Then, the MML binds the outgoing message to a path chosen from the list of active paths in the system. The association of an outgoing message with a path is performed once for every message, and is usually not changed during the lifetime of a message. The library, then, invokes the appropriate send routine from the specific path implementation. For the UDP path, this send routine uses the `writev` system call to hand the message to the network layer of the operating system. After the message is sent (copied into the socket buffer by the `writev` system call), the library waits in progress engine of the library for an acknowledgment from the receiver. On receiving a positive acknowledgment the MML layer frees up the resources allocated for this message and the send call is completed. On the other hand, a negative acknowledgment or the lack of a positive acknowledgment within a given time frame from the receiving node causes the library to retransmit the message, as shown in Figure 2.2. To reduce the latency of the send call, the UDP path copies each outgoing message from the user buffer to its own private buffer. This enables the send call to return immediately on sending the message, without having to wait for the receiver acknowledgment. However, since UDP is inherently unreliable, the library (UDP path) has to keep a copy of the message in its private buffer as long as it does not receive a positive acknowledgment.

The library action on a blocking MPI message receive is also shown in Figure 2.2. The MML posts a message receive based on the `MPI_Recv` call made by the user application. Similar to the posted receives queue and unexpected receives queue described in Section 2.3.2, the MML of LA-MPI also maintains two separate queues for holding posted receives and unexpected receives respectively. Prior to posting a receive, the MML first checks the unexpected receives queue for a match. If a match is made, the message is copied from the library buffer to the user buffer, and the receive call completes immediately. If a match is not made, the receive is placed on the posted receives queue, and the progress engine is invoked to wait for the arrival of the message. When the message arrives, the UDP layer receive function reads it from the network buffer with the `readv` system call, and copies it to the user buffer. Depending upon whether the message was successfully received or not, a positive or negative acknowledgment is sent back to the sender. When an unexpected message arrives at the node, the UDP path allocates a memory buffer, and copies the incoming message to it. Subsequently, when the corresponding receive is posted, this message is copied to the user buffer. Thus, an unexpected receive requires one extra memory copy to be performed in moving the message out of the network buffer and into an internal library buffer. Due to the overhead imposed by this extra memory copy, unexpected receives can severely degrade the performance of communication. As already mentioned, the send path also copies every outgoing fragment before sending it. This also creates a severe performance bottleneck in the LA-MPI UDP path.

2.6 MPI Benchmarks

The evaluation of various MPI library implementations, and also of different hardware systems used for parallel programming requires the use of efficient and discerning MPI benchmarks, which can provide valuable insight into the expected performance of a MPI application on a given message-passing system. The importance of MPI benchmarks is further amplified by the fact that MPI applications are typically extremely large software systems whose performance is dependent upon multiple factors. Two factors commonly perceived as being the most significant determinants of MPI application performance are the MPI library message latency, and message bandwidth. In general, a lower message latency and a higher message bandwidth results in better performance. Beyond these factors, which are straightforward attributes of a MPI library, there are some MPI application dependent factors as well, which impact its performance. The use of non-blocking communication (as compared to blocking communication) in the application tends to lead to much better performance be-

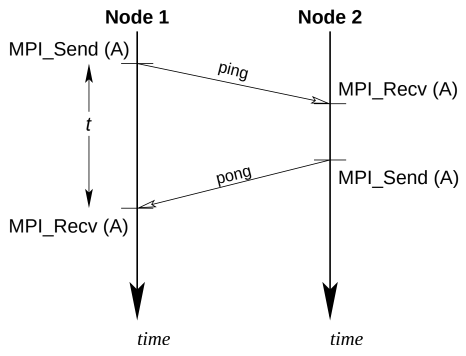

Figure 2.3 Timeline for *ping-pong* latency microbenchmark

cause of greater overlap between an application's computation and communication. Also, the distribution of message sizes used by the application impacts performance because larger messages incur lower per-byte overhead, and hence can yield higher performance.

MPI benchmarks are broadly categorized into two groups—microbenchmarks and application benchmarks. The microbenchmarks enable library developers and users to evaluate the basic communication properties of the MPI library, and/or the message-passing medium. These basic communication properties include message latency and message bandwidth. Note that the communication properties of the medium itself can also be determined by non-MPI benchmarks, which test for communication latency and bandwidth. Therefore, the MPI microbenchmarks are more useful for the evaluation of MPI libraries. The microbenchmarks typically employ point-to-point communication between a pair of nodes, and provide information about some particular aspect of the library. One microbenchmark commonly used to estimate message latency and bandwidth is the *ping-pong* latency test. The steps involved in this microbenchmark are illustrated in Figure 2.3. Node 1 sends a message of fixed size to Node 2 using the blocking MPI message send function. Node 2 receives this message after sometime and immediately sends it back to Node 1. The total time elapsed from Node 1 starting the message send to completing the message receive is the *ping-pong* latency of the message (shown as t in the figure). Half of this latency is referred to as the *ping* latency, and is indicative of the latency of sending a message from a user MPI process on Node 1 to another MPI process (of the same application) on Node 2. The message bandwidth is simply the size of the message divided by the ping latency.

Another MPI microbenchmark is usually used to estimate the maximum achievable unidirectional bandwidth of a MPI library. This test too involves two participating MPI nodes—one of them is a message sender, and the other is a message receiver. The sender sends a number of back to back MPI messages (using `MPI_Send`) to the receiver. The receiver merely receives those messages using an appropriate MPI message receive function. The message bandwidth supported by the library is obtained by dividing the total number of bytes of messages transferred from the sender to the receiver by the total time taken to complete the said number of messages. This message bandwidth is also the maximum bandwidth supported by the MPI library for the given message size. This test is much more bandwidth intensive than the ping-pong benchmark, and thus, can stress the message bandwidth capability of the MPI library more. In order to minimize cache effects on the sender node, the same message is sent repeatedly instead of allocating a fresh message each time.

Application benchmarks are useful by being more indicative of MPI application performance. They consist of the computation and communication intensive loops extracted out of real applications, and can provide an estimate of the overall expected performance of a parallel machine. Two of the most commonly used MPI application benchmarks are the *High Performance Linpack Benchmark* (HPL) and the *NAS Parallel Benchmark* (NPB) set.

The HPL benchmark is a portable as well as freely available implementation of the high-performance computing Linpack benchmark [12]. It is an application kernel that solves a random dense linear system of equations in double precision arithmetic on distributed memory computers. It can be run on an arbitrary number of nodes, and the benchmark itself is highly tunable for various parameters such as the problem size, the number of participating nodes, the grid sizes, etc. The benchmark reports its results in terms of the gigaflops (GFLOPS) it was able to achieve running on the parallel machine with the specified set of parameters.

The NAS Parallel Benchmark (NPB) set is a collection of 8 benchmarks, comprising of five application kernels and three simulated computational fluid dynamics (CFD) applications [1]. This set of benchmarks was developed at the NASA Ames Research Center, and is meant for distributed memory computers. The original NPB suite (version 1.0) was a purely pencil-and-paper benchmark. However, version 2.0 onwards the suite incorporated a MPI implementation of the benchmarks as well using a mix of Fortran 77 and C. The current NPB version 2.2 suite contains seven benchmark problems—BT, EP, FT, IS, LU, MG, and SP [2]. Of these EP, FT, IS and MG are computation kernels mimicking those usually found in CFD applications. The other three benchmarks are simulated CFD applications, which seek to faithfully reproduce much of the data movement and computation found in full application codes. These benchmarks are tunable to a certain degree, though lot less than the HPL benchmark. In particular, these benchmarks support three pre-compiled data-set sizes, and can be configured to execute on N processors, where N is constrained to be either a perfect square (for some benchmarks) or a power-of-two (for the rest). The performance results for each benchmark are reported in terms of the total time spent in executing the benchmark as well as the megaflops (MFLOPS) obtained on the parallel machine as a whole.

Table 2.2 Overall message characteristics of NAS benchmarks (used in this thesis)

Benchmark	Total Number of Messages Sent	Total MBytes of Messages Sent	Maximum Message Size (bytes)
BT.B.4	9,783	2,704	648,960
BT.B.9	32,954	5,475	294,000
BT.C.9	36,591	14,229	726,000
LU.B.4	202,139	1,182	416,160
LU.B.8	505,353	2,363	416,160
LU.C.8	805,353	5,988	1,049,760
SP.B.4	19,383	4,852	440,000
SP.B.9	65,354	9,703	203,456
SP.C.9	65,354	24,696	513,216
MG.B.4	12,317	448	268,320
MG.B.8	24,897	540	135,200
MG.C.8	27,921	2,137	532,512
IS.B.4	467	1,408	8,466,880
IS.B.8	1,703	1,408	2,130,548
IS.C.8	1,703	5,633	8,519,524

Table 2.2 shows the message characteristics of the NAS benchmarks used extensively in this thesis. The data-set size and the number of nodes, for which the benchmark was configured, are indicated as a part of the benchmark name. The

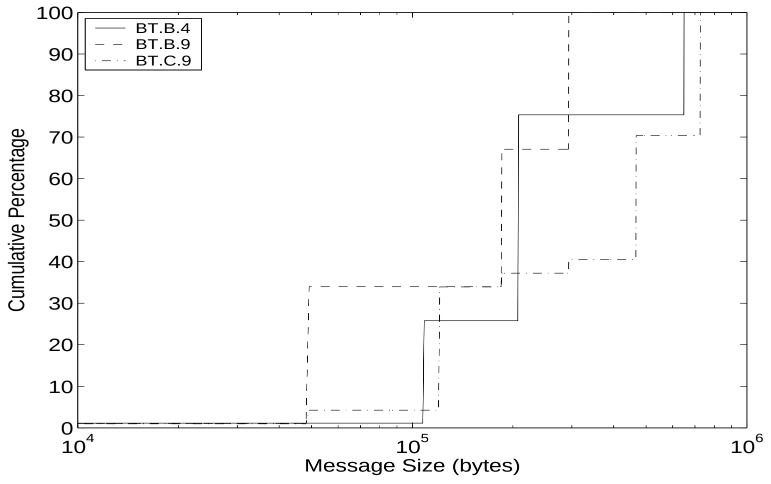

Figure 2.4 Cumulative frequency distribution of message sizes for BT

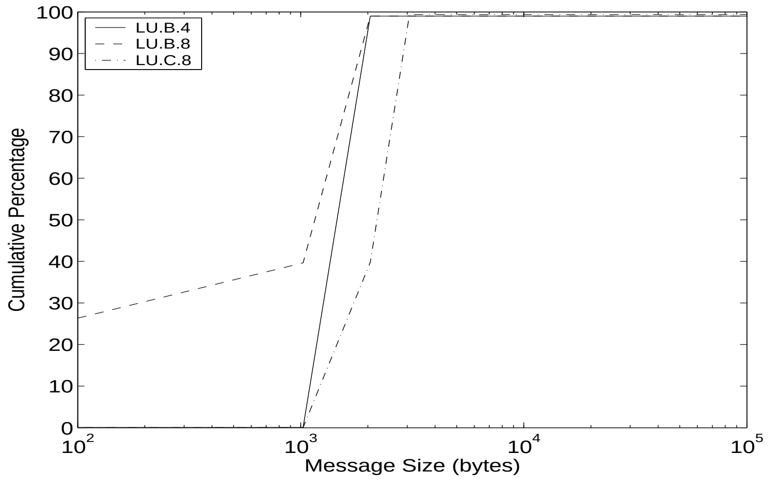

Figure 2.5 Cumulative frequency distribution of message sizes for LU

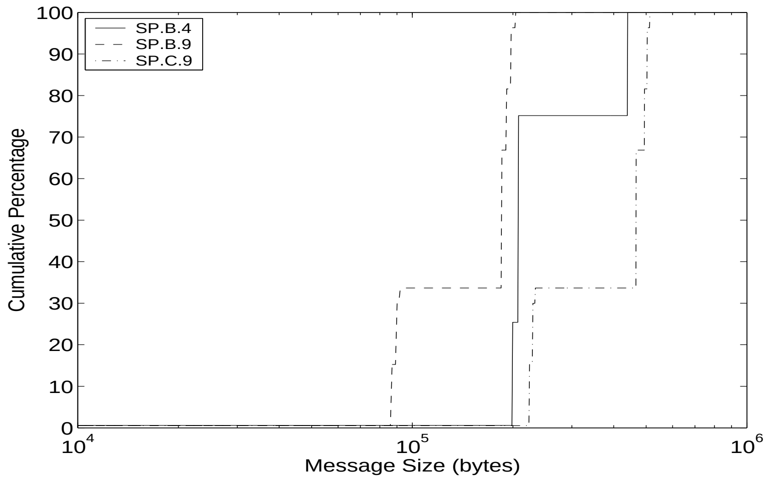

Figure 2.6 Cumulative frequency distribution of message sizes for SP

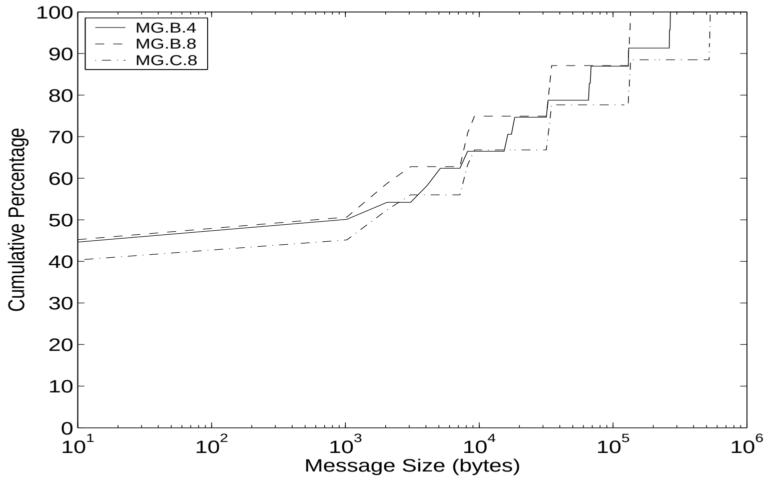

Figure 2.7 Cumulative frequency distribution of message sizes for MG

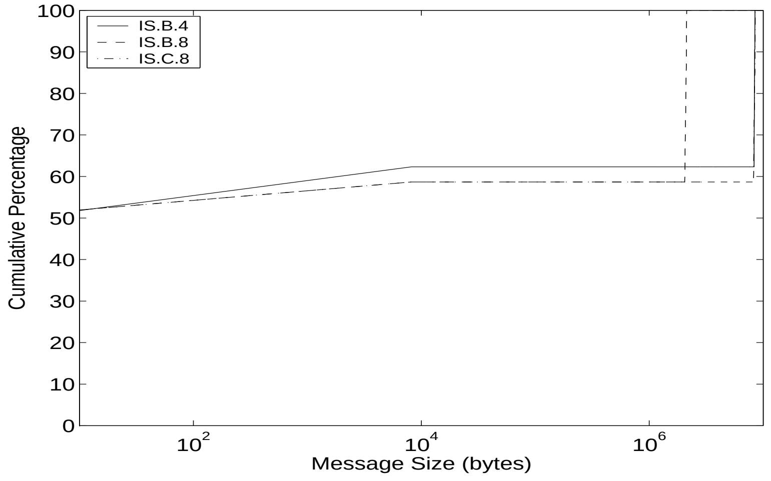

Figure 2.8 Cumulative frequency distribution of message sizes for IS

table shows that the benchmarks (and different configurations of them) are widely different in the number of messages exchanged, the total volume of message-data exchanged, and the maximum message size employed by the benchmark. While none of these attributes are directly indicative of the overall performance of a benchmark, they give good insight into understanding the performance obtained with a particular MPI library. A better indicator of performance is the distribution of message sizes employed by the application benchmark. The cumulative frequency distribution of message sizes for the five NAS benchmarks used throughout this thesis are shown in Figures 2.4, 2.5, 2.6, 2.7, and 2.8, respectively. These figures also corroborate the argument that these benchmarks are widely different in terms of the communication profiles they have.

Both the HPL benchmark and the NPB set of benchmarks are carefully constructed for maximum performance. Especially HPL, with its extensive configure options, can be tuned to a very high extent to generate the best possible performance on a parallel machine. All of these benchmarks use a mix of blocking and non-blocking MPI communication primitives, to varying degrees. Even though non-blocking communication is typically higher performance, its not always possible to find independent computation to perform while a certain non-blocking communication is taking place. In such cases, MPI application developers typically resort to blocking variants.

2.7 Summary

The MPI standard provides a nice and powerful set of APIs for parallel programming using the message-passing paradigm. The standard concerns itself with just the semantics of message-passing and makes no implementation suggestions, thereby making the standard truly portable. This has resulted in the development of a number of message-passing libraries, which conform to the MPI standard, and support a wide variety of hardware platforms. But owing to their different software architectures and level of software support for various networking technologies they differ widely in performance from one another. This chapter provides a high-level comparison of the software architecture and the point-to-point communication implementation for Ethernet messaging in a couple of commonly used, freely available MPI libraries—MPICH and LA-MPI. This comparison provides valuable insight into the expected performance and the bottlenecks of Ethernet message-passing of these libraries. At the same time, it also provides the basic understanding of how messages are sent and received in a MPI library, which remains the crux of any MPI library's functionality.

This chapter also provides an overview of MPI benchmarking techniques for evaluating MPI libraries. Two important factors affecting MPI application performance are the library message latency and message bandwidth. Simple microbenchmarks are typically used to measure these attributes of the MPI library. The performance of MPI applications, however, also depends upon other factors external to the library, such as the distribution of message sizes, and mode of communication. But the library implementation is the crucial factor determining the efficiency of MPI communication, whether in blocking or non-blocking modes, and for the performance across a range of message sizes. Simple microbenchmarks cannot evaluate these aspects of a MPI library, and so application benchmarks such as HPL and the NPB suite are used frequently to characterize a messaging system's performance.

Chapter 3
An Efficient TCP implementation for LA-MPI

MPI applications are typically run on cluster environments, which have low data-corruption rates and Ethernet packet-drop rates. However, these applications tend to deal with large amounts of data, and can run for several days at a stretch. These unique attributes of MPI applications make reliability a strict requirement for MPI communication. For messaging over Ethernet, this necessitates the use of a reliable communication protocol, such as TCP. However, a common perception in the MPI community is that TCP is an unnecessarily high overhead protocol for efficient MPI communication. Consequently, several MPI library implementations, including the LA-MPI library, provide Ethernet messaging support using UDP. Since reliability is important for MPI communication, these library implementations provide reliability and flow-control mechanisms directly at the library-level, and use UDP datagrams underneath the library for communication. The motivation here is that the lower operating system overhead of UDP would enable better MPI performance by reducing message latency and improving message bandwidth (see Section 2.2).

This chapter proposes the design of a new TCP messaging layer for the LA-MPI library. Since the TCP protocol provides a reliable communication substrate, the reliability and flow-control mechanisms in the library can be turned off. Thus, the MPI library overhead for communication is reduced by using TCP for message communication rather than UDP. However, this is at the cost of increasing the operating system overhead by using TCP for message communication. The results show that the gain from reducing library overhead more than offsets the increase in operating system overhead. Using TCP as a messaging layer for the LA-MPI library not only improves the message latency and message bandwidth of the library, but also improves overall application performance. The benefits of using a TCP messaging layer are most noticeable in library message bandwidth. LA-MPI with a UDP messaging layer never utilizes more than 50% of available Gigabit Ethernet bandwidth at message sizes above 1 KB. In sharp contrast, with the TCP messaging layer, LA-MPI achieves line-rate message bandwidth for as small as 8 KB messages. Thus, a well designed TCP messaging layer can offset the extra operating system overhead and outperform a UDP messaging layer. Furthermore, due to the years of research that has gone into the development of the TCP protocol stack, using the reliability and flow-control features of TCP is much more robust and efficient than implementing the same at the library-level.

The rest of this chapter proceeds as follows. Section 3.1 describes the UDP messaging layer of LA-MPI library in detail. Section 3.2 proposes the design of a TCP messaging layer for the LA-MPI library, to replace UDP with TCP as the messaging protocol over Ethernet. An evaluation of the TCP LA-MPI library is presented in Section 3.3. Finally, Section 3.4 summarizes the chapter and its main contributions.

3.1 UDP Path for LA-MPI

As discussed in Section 2.4, the LA-MPI library supports multiple interconnect hardware as well as their respective software communication protocols. For this purpose the library utilizes an abstraction of the interconnect technology known as the *path*. Version 1.3.x of the LA-MPI library was described in Section 2.4.1, and discussed the library support for UDP messaging over Ethernet from a high-level perspective. This section focuses on the more intricate implementation details of the UDP path in an effort to understand its performance bottlenecks better.

The UDP path is one of the several different path implementations within the SRL of the LA-MPI library (Section 2.4.1). It supports UDP message communication over Ethernet using the socket interface to the operating system's network protocol stack. Since the UDP protocol does not provide any reliability or flow-control of communication, the LA-MPI library provides it own reliability and flow-control mechanisms. The UDP path implementation accomplishes flow-control by fragmenting outgoing messages, and using a simple sliding-window scheme with per-fragment acknowledgments. These acknowledgments also serve to provide reliability of communication in the library. The MML layer of the library keeps track of the messages being sent and acknowledged. The responsibility of sending and receiving acknowledgments, however, rests with the path; in this case the UDP path. In the UDP case, the MML considers a message acknowledged only when the UDP path has received all fragment acknowledgments for the message from the receiver.

The UDP path, like other paths supported by LA-MPI, provides its own progress routine, which gets invoked by the library's main progress engine. The UDP path's progress routine is very elementary, and only checks for incoming message fragments on the UDP sockets employed by the path. The `select` system call is used for this purpose. `Select` takes three descriptor arrays (read, write and exception), and a timeout period as arguments. If there is pending activity on any of the descriptors, `select` returns with three new arrays of descriptors on which activity is pending. In the absence of any activity, `select` blocks until an activity occurs or the time period is exceeded. The UDP path makes use of a non-blocking `select` call (with 0 timeout period) to only check for arrived data on its socket descriptors. If there is some data to be read from the socket, the appropriate UDP path receive function is invoked. Transmission of pending messages is handled directly by the library's main progress engine by invoking the send function of the UDP path, and trying to send as much of the pending messages as possible. The progress engine of the library attempts to make progress on all active paths in the library in turn, and thus could invoke the UDP's path progress routine (and the send function) several times in the process of satisfying a MPI request.

Every LA-MPI node uses one instance of the UDP path, which is created and initialized during library startup. The UDP path uses two UDP sockets connected to different ports for message communication. These are termed as the *short-socket* and the *long-socket*, depending on fragment sizes that arrive (or are sent) on these sockets. The short-socket is used for receiving (or sending) message fragments shorter than a predefined limit (2 KB), and the long-socket is used for larger fragments. It is necessary to use two sockets for receiving (or sending) in the UDP path, because in the absence of flow-control across message streams, a large message can swamp resources at the receiver completely and cause other messages to get significantly delayed. During initialization, the UDP path also initializes fragment descriptor lists for handling incoming and outgoing message fragments during the run of the MPI program.

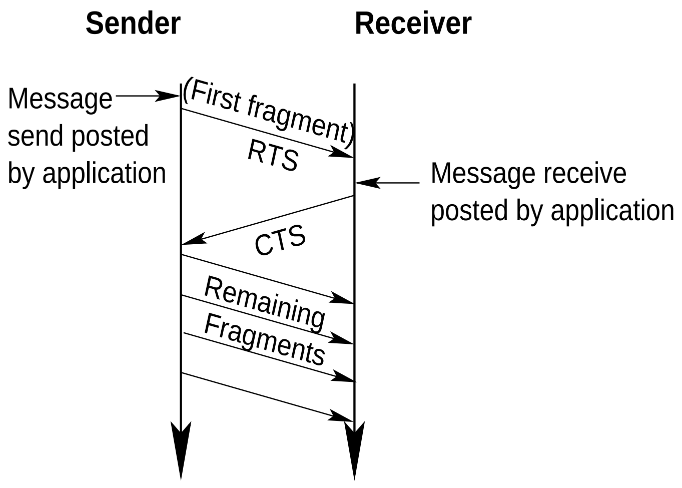
Sender
Receiver
Message send posted by application
(First fragment) RTS
Message receive posted by application
CTS
Remaining Fragments

Figure 3.1 Illustration of the *rendezvous* protocol

3.1.1 Message Transmission

Message transmission in the UDP path proceeds by first fragmenting the outgoing message. The first fragment is 2 KB in size, and all subsequent fragments can be up to 8 KB. The MPI specification does not require message fragmentation, but it helps to limit the necessary buffering on the receiving node, and enables the UDP path to implement the sliding window flow-control scheme. Each message fragment also includes a header to enable the receiver to identify the fragment. The first fragment is always sent on the short-socket, and the subsequent fragments on the long-socket, on both the sender and the receiver. If a message is small enough ($\leq$ 2 KB), it is sent as a single fragment to the destination. This is the *eager* protocol of message transfer, as the sender does not explicitly request permission from the receiver before sending the message (see Table 2.1). Larger messages (which are necessarily multi-fragment) are sent using the *rendezvous* protocol. The steps involved in this protocol are illustrated in Figure 3.1. For a multi-fragment message, the first fragment serves as the request-to-send (RTS) message from the sender to the receiver. After sending the RTS, the sender waits for an acknowledgment from the receiver, which is also termed as the clear-to-send (CTS) message. The receiver, on receiving the first fragment, and after the corresponding receive has been posted by the application, sends the CTS message. The sender on receiving the CTS, sends the remaining fragments of the message (subject to flow-control restrictions obviously). The use of RTS/CTS is not essential for flow-control in the UDP path, since the sliding window mechanism already takes care of that. But their use enables the receiver to control the order in which concurrent messages from the same or different sender(s) are processed.

The UDP path relies on per-fragment acknowledgments to provide reliability. However, waiting for the arrival of acknowledgments can increase the message latency for the sender significantly. To counter this, the UDP path copies every fragment into a local buffer prior to sending it. The UDP path marks a send request as complete when all the fragments of a message have been copied, regardless of the number of fragments acknowledged. Since the UDP path guarantees the delivery of the fragments, the MPI send message call can return control to the user application without waiting for all the acknowledgments, and thus, lower the message latency from the application's perspective. The library buffer for a fragment is freed on receiving its positive acknowledgment. Retransmission of a fragment occurs in two circumstances—arrival of a negative acknowledgment for the fragment from the receiver, or exceeding the timeout value (time for which the UDP path waits to receive a positive acknowledgment) associated with every fragment after it is enqueued on the socket.

The UDP path also implements a simple per-message-stream sliding window flow-control mechanism. It allows only a certain maximum number of fragments (equal to window size) to be sent without waiting for acknowledgments to the preceding fragments. Reaching this limit causes the UDP path to suspend sending further fragments of the message until at least some of the outstanding fragments have been acknowledged.

3.1.2 Message Reception

Message receives in the UDP path are always triggered by its progress routine when there is new data on either of its sockets. Since the first fragments of all messages arrive on the short-socket, it is enough to just monitor the short-socket for incoming data. The non-blocking `select` system call is used for this purpose by the progress routine of the UDP path (as described in Section 3.1). The progress routine of the UDP path is invoked by the library's main progress engine whenever it executes. The progress routine then dispatches the receive function of the UDP path when it finds new data on the short-socket. The first fragment of any message also informs the sender about the number of fragments to expect for this message. In case of a multi-fragment message, the subsequent fragments all arrive on the long-socket. Thus, after the UDP path has read and processed the first fragment from the socket, for a multi-fragment message it flags the long-socket as active. Subsequent progress routine invocations check for incoming data on both sockets until completion of the multi-fragment message reception deactivates the long-socket again.

As explained in Section 2.5, the MML of the LA-MPI library maintains two separate queues—the *posted receives queue* (PRQ) and the *unexpected receives queue*

(URQ). The PRQ holds receive requests made by the user application for which the corresponding messages have not yet arrived at this node. Similarly, the URQ holds the incoming fragments for which the user application has not made the corresponding MPI receive requests (posted the receive). Unexpected receives are a characteristic of the message-passing paradigm, since communication is producer-driven and the sender can always send a message to a receiver without the receiver making a request for it ahead of time (the receiver may even never request that message!).

For fragments arriving on the short-socket, the receive routine of the UDP path first reads the whole fragment from the socket buffer into a local buffer. Then, the messages on the PRQ are checked to find a match for this fragment. In case a match is found, the message data from the local buffer is copied into the user buffer. If no matching receive could be found in the PRQ, this fragment is put on the URQ to be matched against future receive postings by the application. On successful receive of a fragment into the user buffer, the UDP path sends a positive acknowledgment back to the source of the fragment. This satisfies the *rendezvous* protocol as well, since the acknowledgment is only sent when a matching receive is posted. The subsequent fragments of a multi-fragment (*rendezvous*) message all arrive on the long-socket. Since fragments arriving on the long-socket are guaranteed to have been matched with posted receives earlier, the UDP path just reads the header of incoming fragments from the long-socket buffer and then copies the data directly from the socket to the corresponding user buffer.

Since UDP is an unreliable protocol, fragments might be lost or corrupted en route. When the UDP path determines this condition for a particular fragment, it sends back a negative acknowledgment for that fragment to the sender, forcing the sender to retransmit the lost/corrupted fragment.

3.2 A TCP Path for LA-MPI

An implicit assumption in the design of the UDP path in LA-MPI is that the UDP protocol is significantly lower overhead than the TCP protocol. The extra overhead arises out of the features that TCP provides over and above UDP, namely, reliable transmission of data, flow-control of communication, and connection management. For MPI applications running on cluster environments, the reliability and flow-control mechanisms used by TCP were considered overkill for the purpose. MPI communication also does not rely on the notion of connections so long as the source of a message knows how to address it to the destination. These reasons motivated the development of a UDP messaging scheme in LA-MPI, which uses UDP datagrams for message communication between MPI nodes. Reliability and flow-control is provided by message fragmentation, and the use of per-fragment acknowledgments.

The use of library-level acknowledgments, while providing reliability and flow-control of MPI communication, also increases the per-message (and per-byte) library overhead. Waiting for acknowledgments can increase the message latency from the sender's perspective. To get around this problem, the UDP path copies outgoing messages into a local buffer and uses the local buffer instead of the user buffer for retransmissions, if required (described in Section 3.1.1). However, this technique introduces an extra memory copy in the send path of a message, in addition to the memory copy involved in moving the message from the application buffer in the user-space to the socket buffer in kernel-space. In the absence of zero-copy sockets, this second memory copy is essential for moving data from an application to the network. Copying memory is an expensive operation in modern computer systems and introduces additional latency in the library message send path for UDP. Furthermore, the use of library-level acknowledgments also has an adverse effect on message bandwidth. These acknowledgments have to transcend user and kernel boundaries on both the sender and the receiver, which adds to their latency. In contrast, the TCP acknowledgments are completely kernel-space on either end, and thus save some latency by avoiding context switches and memory copies (between application and kernel). Increasing acknowledgment latency slows down the sender (flow-control), and thus reduces message bandwidth attainable with the UDP path of LA-MPI. The performance of the UDP path is also negatively influenced by its use of small size fragments, which cause higher per-byte overhead in message communication (because of headers associated with every fragment). At the same time, it is necessary to use small size fragments in UDP to minimize the retransmission penalty.

The above observations on the UDP path of LA-MPI suggest the use of TCP as the messaging medium on Ethernet, and disabling all library-level reliability features. As a direct consequence this would eliminate the extra memory copy on the send path of every MPI message, leading to lower message latency. Moreover, the reliability and flow-control mechanisms in the TCP layer of the operating system are much more robust and efficient owing to the years of research and development effort behind them, leading to higher performance. Finally, since reliability is satisfied by the TCP protocol and not by the LA-MPI library, the library-level fragment sizes used by the TCP path can be much larger. The TCP protocol segments these fragments further in the protocol stack, and uses these segments for retramissions. Thus, the library-level fragment sizes does not adversely impact the retransmission overhead. A larger library-level fragment size also enables copying larger blocks of data at a time between user-space and kernel-space on a socket send or receive call, which contributes to improved communication performance of the TCP path.

This chapter presents the design of a TCP path for the LA-MPI library v1.3.x to replace the UDP path for messaging over Ethernet. Since the TCP protocol ensures reliable transmission of data, all reliability features in LA-MPI are turned off for the TCP path. In addition, no flow-control mechanisms are implemented as a part of the TCP path of the library, instead relying on those provided by the TCP protocol.

3.2.1 Design of a TCP Messaging Layer for LA-MPI

The TCP path of LA-MPI aims to completely replace the UDP path for messaging over Ethernet. It supports TCP communication using the socket interface to the operating system's network protocol stack. Since the TCP protocol has its own reliability and flow-control mechanisms, the LA-MPI library (or its TCP path) does not need to provide any of the mechanisms that were discussed in Section 3.1 with reference to the UDP path.

The TCP path utilizes events to manage communication. An event is any change in the state of the TCP sockets in use by the library. There are three types of events: read events, write events, and exception events. A read event occurs when there is incoming data on a socket, regardless of whether it is a connection request or a message itself. A write event occurs when there is space available in a socket for additional data to be sent. An exception event occurs when an exception condition occurs on the socket. Currently, the only exception event supported by the socket interface to the TCP/IP protocol stack is the notification of out-of-band data. The current LA-MPI library does not use any out-of-band communication, so exception events are currently unused, and only included to support possible future extensions.

Like the UDP path, the TCP path also provides its own progress routine, which gets invoked by the library's main progress engine. The TCP path maintains three separate lists, one for each type of event, to keep track of all the events of interest. At various points of execution, the TCP path pushes these events of interest on to the respective event lists. In addition, callback routines are registered for each event. The progress mechanism of the TCP path uses these lists to inform the operating system that it would like to be notified when these events occur. The `select` system call is used for this purpose. The operation of the `select` call was explained in Section 3.1. The progress routine of the TCP path in LA-MPI makes a non-blocking `select` call to retrieve events from the operating system. If this call returns some events that have occurred, they are each handled in turn by the appropriate callback routines, and finally control is returned to the progress engine. The progress engine then attempts to make progress on all other active paths in the library. Similar to the progress routine behavior of the UDP path, the TCP progress routine can be invoked several times in the process of satisfying a blocking MPI request.

3.2.2 Connection Establishment

At runtime, there is one instance of the TCP path on every LA-MPI node, which is created and initialized during library startup. The TCP path on each node allocates one socket port to listen on for incoming connection requests from peer nodes.

This listening port is obtained dynamically on each node in the process of binding the port to a TCP socket, and broadcast among all other peers using the administrative network of the LA-MPI runtime system. It is important to obtain the port dynamically since in a large cluster the availability of any pre-determined port for LA-MPI TCP connections is not guaranteed. The listening sockets are kept permanently on the read event list of the TCP path node since connection requests may arrive at any time.

The TCP path uses unidirectional connections for all message communication between nodes. Thus, between every pair of LA-MPI nodes there are two TCP connections—every node sends messages to the peer node on one connection (*send-connection*), and receives from the peer on the other connection (*receive-connection*. Note that bidirectional connections can be used between communicating MPI nodes just as well. However, unidirectional connections enable a simpler implementation. The TCP connections between nodes are set up only on a demand basis, that is, only when there is a message to be sent on that connection. The TCP path checks for connected TCP sockets before attempting to send a message to a peer LA-MPI node. In case there are no established connections, a connection request is sent to the message destination. The TCP path on the source node then waits for the arrival of a connection acceptance notification from the destination node. On the destination node, the operating system informs the TCP path's progress routine of an incoming connection request (by the `select` system call), which in turn dispatches a callback function to handle connection acceptance. Upon accepting the connection, the destination sends its process rank to the source on this newly created connection.

This serves as the connection acceptance notification for the source LA-MPI node. The connection is ready for message transmission (and reception by the receiver) when the source node also sends its process rank to the destination node. This library-level handshake is necessary to enable participating processes to form the mapping between established connections and process ranks.

Once a connection is established, it remains open for future message transmissions, unless it is idle long enough that the operating system destroys it. In that case a new connection would be established for future messages between those nodes.

The TCP path of the LA-MPI library requires one TCP connection between every pair of communicating nodes of a MPI application. In a cluster environment consisting of thousands of nodes, each node might communicate with a few hundred other nodes in the system, thereby, requiring every node to support a few hundred open TCP connections. This would lead to an increase in the connection management overhead in the operating system. However, modern operating systems have highly tuned protocol stack implementations which allow them to scale to thousands of open connections with little performance degradation. On the other hand, the UDP path performs no connection management tasks since the UDP protocol is a connection-less protocol, and thus, has no connection management overhead.

3.2.3 Message Transmission

The TCP path starts the send process on a message by first fragmenting it—the first fragment is 16 KB in size, and all subsequent fragments are 64 KB. Each message fragment includes a header to enable the receiving node to identify the fragment. The fragments, along with their headers, are placed on a send queue within the library prior to being sent over the network. A write event for the appropriate connection is then added to the write event list to indicate that there is data in the send queue waiting to be sent. The MPI specification does not strictly require fragmentation, but it helps to limit the necessary buffering on the receiving node. The TCP path, like the UDP path described in Section 3.1, follows two protocols of message transmission depending on its size—*eager* and *rendezvous* (also see Table 2.1). The semantics of these protocols remain identical to those for the UDP path, and only the message size values differ. The TCP path follows the *eager* protocol for messages smaller than or equal to 16 KB in size, and the *rendezvous* protocol for messages larger than that. The message transmission protocols of the LA-MPI SRL are explained in greater detail in Section 3.1.1.

When the TCP progress routine processes the write event (because there is space available in the socket buffer for writing), the first fragment is sent from the send queue, and the write event is removed from the event list. This first fragment serves as a RTS message to the receiver. The sender then adds a read event for this connection to wait for the CTS message, which the receiver sends when it is ready to receive the rest of the message. Note that the use of RTS/CTS is not strictly necessary in TCP for flow-control, as the socket buffer provides implicit flow-control. If the receiver is not ready to receive the message, the receiver's socket buffer will fill up, and TCP flow-control will stop the sender from sending further data. However, TCP does not enable the receiver to control the order in which concurrent messages are processed within the same connection. So the library relies on RTS/CTS messages. Note that TCP enables the receiver to reorder concurrent messages from different senders trivially since they are using different connections. Once the *clear-to-send* has been received, the write event is placed back on the write event list until all fragments in the send queue have been sent. Depending on the size of the message, and the size of the socket buffer, this could require multiple invocations of the appropriate callback routine.

The TCP path marks a send request as complete when all the fragments of that send message have been enqueued on the socket buffer. Since TCP guarantees delivery of the message to the receiver, the TCP path of LA-MPI itself does not provide any additional reliability features, instead relying purely on the reliability mechanisms provided by the TCP protocol. Completion of a send request signals the library to release all of the resources held for this message and to notify the application appropriately.

3.2.4 Message Reception

As in the UDP path, message reception in the TCP path is more complicated than message transmission due to the occurrence of unexpected receives and posted receives. As explained in Section 2.3.2, an unexpected receive is a message that has not been posted by the application in advance. Similarly, a posted receive is a message that has already been posted by the application at the time the message arrives at its destination. The *unexpected receives queue* (URQ) and *posted receives queue* (PRQ), maintained by the MML, are used to handle the two variants of message receives (see Section 3.1.2).

For TCP, message receives also use network events, as in the send case. The receive handler of the TCP path is invoked through the progress engine of the library whenever there is incoming data on any of the established connections. All established connections are always active on the read event queue since messages may arrive at any time. The receive handler first reads the header of the incoming fragment from the socket buffer. Then, the messages on the PRQ are checked to find a match for the incoming fragment. If a match is made, the receive routine directly copies the data from the socket buffer into the user buffer posted by the application.

If the receive has not yet been posted, buffer space for the unexpected fragment is allocated by the TCP path, and the fragment is added to the URQ. Subsequent receives posted by the application are first matched against fragments in the URQ before being posted on the PRQ. The library buffer used to allocate an unexpected receive is freed when a match is made in the URQ, and the data from the buffer has been copied into the user buffer. At that point, the fragment is also removed from the URQ.

For multi-fragment messages, as already explained in Section 3.1.1, the first fragment of the message serves as the RTS message to the receiver. For RTS messages, the receiver does not send a CTS to the sender until the corresponding receive has been posted by the application. So, if a match for an incoming fragment is found on the PRQ, the CTS is sent back immediately. Otherwise, the CTS is delayed until the application posts a receive that matches in the URQ. This ensures that at most only one fragment of any unexpected message gets buffered in the library at a node. Again, this is only necessary to enable the receiver to easily reorder concurrent messages from a single sender, since TCP already provides flow-control for unread messages. The receive message operation is completed when all the fragments of a message have been received and copied into the user buffer.

3.3 Evaluation of the TCP Path of LA-MPI

This section presents an evaluation of the proposed TCP path, and compares it against the UDP path, for the LA-MPI library v1.3.x. The evaluation is performed using both microbenchmarks and application benchmarks. The microbenchmarks test for the message ping latency, and the one-way communication bandwidth of the MPI library. The *ping-pong* latency benchmark and the unidirectional bandwidth benchmark, as described in Section 2.6 are used for this purpose. Two application benchmarks from the NPB suite (IS and LU) and the HPL benchmark are also used in this evaluation (also described in Section 2.6).

The evaluation is performed on a FreeBSD workstation cluster employing up to 4 processing nodes. Each workstation has one AMD Athlon XP 2800+ processor, 1GB DDR SDRAM, and a 64bit/66Mhz PCI bus. Each of the nodes also has at least 40GB of hard drive capacity (none of the benchmarks are disk intensive). Each workstation also has one Intel Pro/1000 Gigabit Ethernet network adapter. The nodes are connected to each other through a 8-port Gigabit Ethernet switch. Each node runs an unmodified FreeBSD-4.7 operating system with various socket and networking parameters tuned in order to provide maximum TCP/IP performance. This includes enabling large window sizes in TCP (RFC1323), setting maximum allowable socket buffer size to 512 KB, and tuning related kernel parameters, such as *nmbclusters*, *maxsockets*, etc.

In the following subsections, the microbenchmark performance and the application benchmark performance of the two versions of the library is presented. For clarity,

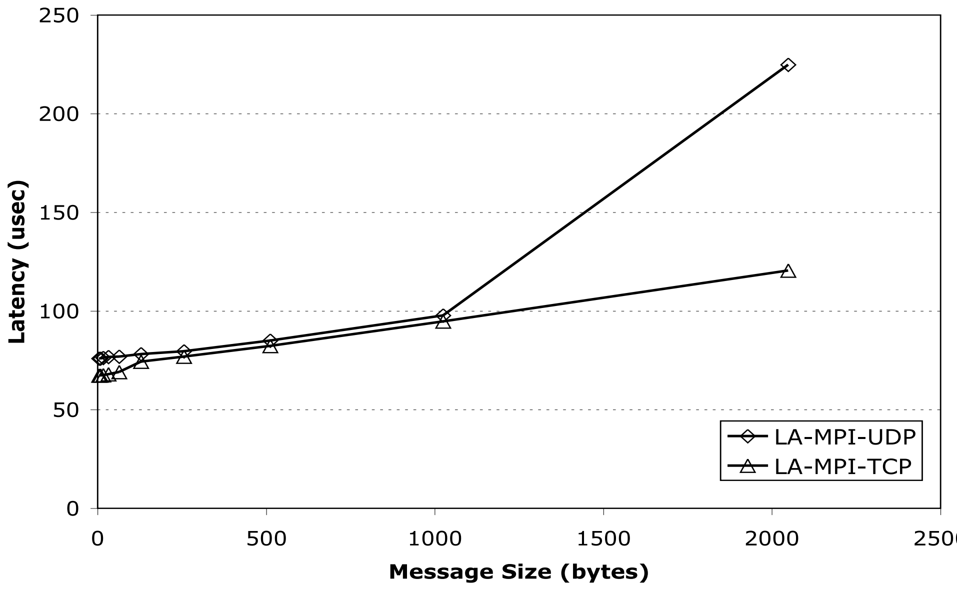

Figure 3.2 Comparison of ping latency between LA-MPI-TCP and LA-MPI-UDP (for 4 B to 2 KB message sizes)

LA-MPI-UDP is used to refer to the LA-MPI library with the UDP path active and LA-MPI-TCP is used to refer to the LA-MPI library with the TCP path active.

3.3.1 Microbenchmark Performance

The message ping latency between two communicating LA-MPI nodes is measured using the *ping-pong* latency test described in Section 2.6. Figure 3.2 shows the ping latency for 4 B to 2 KB message, for both LA-MPI-UDP and LA-MPI-TCP libraries. Figure 3.3 shows the same comparison for 1 KB to 128 KB message sizes. Both the figures show that the message ping latency with LA-MPI-TCP is consistently lower than that with LA-MPI-UDP. The difference in ping latencies between the two libraries is nearly 13% at 4 B message size (approximately 8.6 μsec) and reduces with message size at first to about 3% at 1 KB message size (approximately 3 μsec). Beyond 1 KB messages, the messaging protocol in LA-MPI-UDP shifts from *eager* to *rendezvous*, and this can be seen by the sharp jump in ping latency at 2 KB message

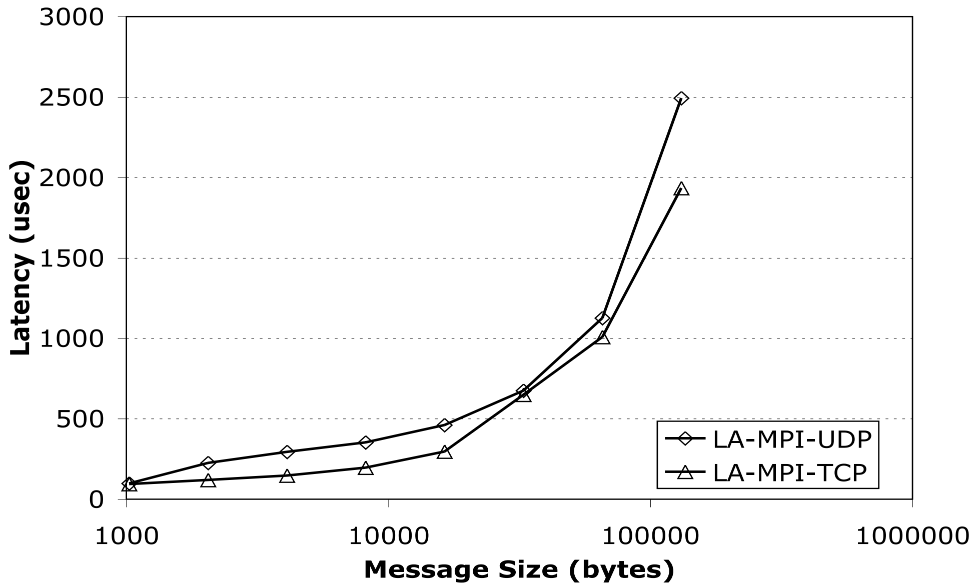

Figure 3.3 Comparison of ping latency between LA-MPI-TCP and LA-MPI-UDP (for 1 KB to 128 KB message sizes)

size of LA-MPI-UDP in Figure 3.2. At this point, the difference in ping latencies between the two libraries is greater than 80%, that is, the LA-MPI-UDP ping latency with 2 KB messages is about 104 μsec higher than LA-MPI-TCP. The switch in messaging protocols for LA-MPI-TCP occurs as message size is increased beyond 16 KB (seen in Figure 3.3), but even then the ping latency with LA-MPI-UDP is more than 4% (26 μsec) higher than that with LA-MPI-TCP. With increasing message sizes beyond 32 KB, the difference in ping latencies increases steadily to about 29% (559 μsec) at 128 KB message size. These results show that the UDP path is actually higher latency than the TCP path. Due to the extra memory copy performed on the send path of the UDP path, with increasing message sizes, the latency gap between LA-MPI-TCP and LA-MPI-UDP grows as it takes progressively greater amount of time for larger memory copies.

Figures 3.4 and 3.5 show the message bandwidths obtained on this benchmark by

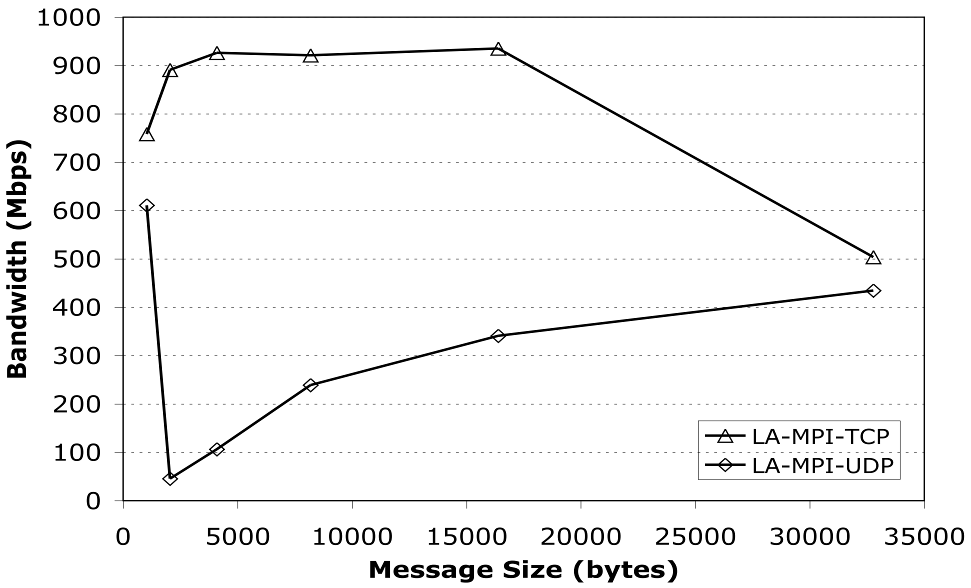

Figure 3.4 Comparison of message bandwidth between LA-MPI-TCP and LA-MPI-UDP (for 4 B to 32 KB message sizes)

the two libraries for 1 KB to 32 KB, and 32 KB to 2 MB message sizes, respectively. Each data point is obtained by iterating over 1000 messages. The figures show that LA-MPI-TCP significantly outperforms LA-MPI-UDP in message bandwidth at all message sizes. However, the most interesting observation is that LA-MPI-UDP is never able to use more than 50% of the available bandwidth beyond 1 KB messages. It performs best with 1 KB messages when it follows the *eager* messaging protocol, and is able to get up to 611 Mbps, or within 24% of the message bandwidth obtained by LA-MPI-TCP. The message bandwidth at 2 KB message size for LA-MPI-UDP falls drastically to about 50 Mbps because of the switch in messaging protocol to *rendezvous*. As message size increases, the message bandwidth with LA-MPI-UDP also increases but never goes beyond 500 Mbps with even 2 MB messages. LA-MPI-TCP on the other hand is able to increase its message bandwidth to almost line-rate with as small as 8 KB message size. LA-MPI-TCP also switches from the *eager* to the

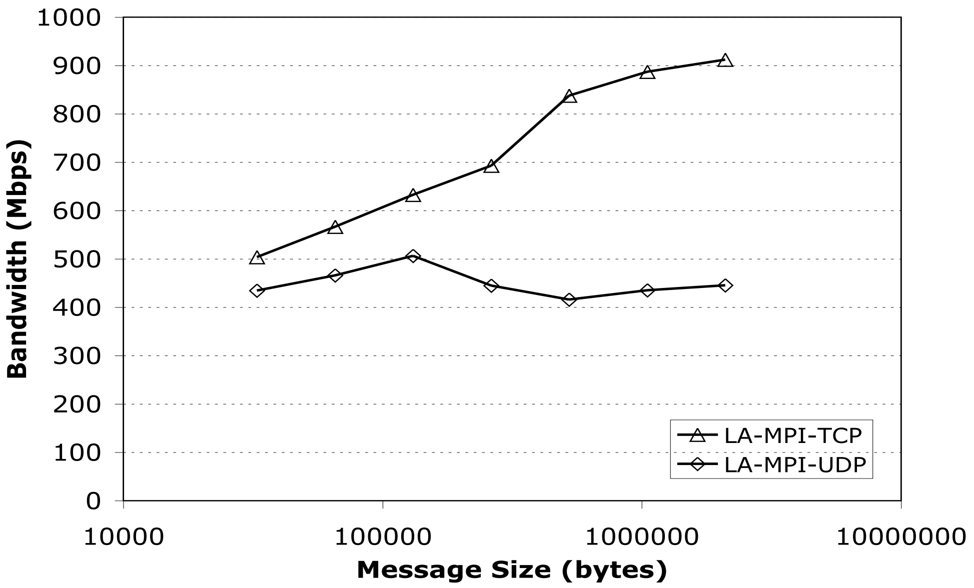

Figure 3.5 Comparison of message bandwidth between LA-MPI-TCP and LA-MPI-UDP (for 4 KB to 2 MB message sizes)

rendezvous messaging protocol beyond 16 KB message size, as seen in the sharp drop in message bandwidth as 32 KB message size. Increasing message sizes beyond 32 KB increases the message bandwidth of the LA-MPI-TCP library steadily, almost getting close to line-rate with 2 MB messages (910 Mbps). Overall, the message bandwidth with LA-MPI-UDP is up to 296% less than that with LA-MPI-TCP, with a minimum difference of 15% at 32 KB message size.

3.3.2 Application Benchmark Performance

The application performance with LA-MPI-UDP and LA-MPI-TCP is compared using the HPL benchmark, and 2 applications from the NAS parallel benchmark suite [1, 12]. These are a more realistic set of benchmarks that include both computation and communication. All of these benchmarks are discussed in greater detail Section 2.6. The two benchmarks from the NPB v2.2 suite used for this evaluation are LU and IS. The LU benchmark uses predominantly short messages for communication.

On the other hand, the IS benchmark utilizes very large messages for communication, and thus, is more bandwidth sensitive than LU. As previously mentioned for the NPB v2.2 suite, each benchmark supports three pre-compiled data-sets—A, B, and C (in increasing order of size). The A and B data-sets are used for this evaluation. All the experiments, HPL, LU, and IS are run on 4 node clusters.

Table 3.1 Application performance comparison between LA-MPI-UDP and LA-MPI-TCP

Benchmark	**LA-MPI-UDP**	**LA-MPI-TCP**
HPL	8.238 GFLOPS	8.338 GFLOPS
LU.A.4	136.54 sec	134.69 sec
LU.B.4	616.48 sec	559.64 sec
IS.A.4	2.54 sec	2.03 sec
IS.B.4	11.49 sec	8.73 sec

Table 3.1 compares the performance of the HPL benchmark, and the two NAS benchmarks achieved with LA-MPI-TCP against LA-MPI-UDP. The HPL performance is reported in terms of the gigaflops (GFLOPS) obtained with the benchmark on the 4-node cluster. The NAS performance is reported in terms of the time taken to run the experiments (seconds), again on the 4-node cluster. The NAS benchmarks are named to reflect the configuration of the experiment as well—the first part of the name gives the benchmark name, the middle part conveys the data-set size, and the last part conveys the number of participating MPI nodes for the experiment.

The table shows that the performance improvement with LA-MPI-TCP varies very widely with benchmark. Thus, the IS benchmark achieves about 20% performance improvement with the A data-set and about 24% improvement with the B data-set. As shown by the data presented in Table 2.2 and Figure 2.8, the IS benchmark is a relatively short benchmark, and uses very large messages for communication among the participating nodes. Thus, the bandwidth advantage of the LA-MPI-TCP library over LA-MPI-UDP at large message sizes (almost 100%) results in a significant performance improvement with LA-MPI-TCP. The other NAS benchmark, LU shows a speedup of about 2.4% with the smaller data-set and 9.3% with the larger data-set. This can also be explained from the message characteristics and message size distribution used by the LU benchmark, and shown in Table 2.2 and Figure 2.5, respectively. The LU benchmark employs a very large number of messages, but a major fraction of them are very small in size. In fact for the B data-set, more than 99% of the messages are less than 1 KB. In this range of message sizes, the LA-MPI-TCP library does not have a very significant bandwidth or latency advantage over LA-MPI-UDP, which translates into a modest performance improvement with LA-MPI-TCP. The HPL benchmark shows a marginal 1.2% speedup with the LA-MPI-TCP library over the LA-MPI-UDP library, again due to the larger number of small messages employed by the benchmark.

The results in Section 3.3.1 show that both message latency and message bandwidth increase with using TCP for LA-MPI message communication. The increase in bandwidth is especially significant. However, the latency and bandwidth gains are not directly translated in to an improvement in application performance in all cases. These results further corroborate the argument in Section 2.6 that a MPI application performance depends upon a lot of different factors, and not just on the attributes that the simple latency and bandwidth microbenchmarks test for. Specifically, the distribution of messages used by the benchmark, and the type of communication employed by it, are both critical to its performance.

3.4 Summary

A common perception in the MPI community is that TCP has an unnecessarily high overhead for MPI communication in a cluster environment. This has prompted some MPI libraries, LA-MPI being one of them, to use UDP datagrams for messaging over Ethernet. However, reliability and flow-control are important concerns for MPI communication as well. The LA-MPI library provides reliability and flow-control mechanisms in the library directly, over a UDP messaging layer. This is achieved in much the same way as the TCP protocol provides reliable communication—through the use of data fragmentation and per-fragment acknowledgments. One of the downsides of LA-MPI-UDP's library-level reliability schemes is the extra memory copy imposed on every outgoing message. This leads to an increase in message latency, and is a big performance bottleneck in the library design. Message bandwidth results show that the UDP path of LA-MPI is never able to utilize even 50% of available bandwidth of Gigabit Ethernet beyond 1 KB messages.

This chapter also introduced a TCP messaging layer for the LA-MPI library. The use of TCP enables switching off reliability and flow-control mechanisms in the library, thereby immediately getting rid of most of the performance bottlenecks. Careful design of the TCP messaging layer of the library also ensures that the per-message library overheads are kept to a minimum. A comparison of message latencies with the UDP path and the TCP path show that TCP messaging is consistently lower latency across a range of message sizes (between 3% and 29%). The TCP path is able to attain line-rate with its message bandwidth for as small as 8 KB messages, and is up to 296% higher bandwidth than the UDP path with the same microbenchmark. Application benchmarks show more modest performance improvements with LA-MPI-TCP signifying that factors other than message latency and bandwidth are also important for overall MPI application performance.

Use of TCP as the messaging layer in LA-MPI enables the library to turn-off library-level reliability and flow-control mechanisms. For detecting bit errors in message transmissions, the TCP protocol utilizes a 16-bit CRC checksum. On the other hand, the reliability mechanism of the LA-MPI library provide it an ability to use a 32-bit CRC checksum. Thus, the probability of an undetected error in a message transmission (or reception) is increased with the TCP path due to the weaker CRC checksum employed by TCP. While for some applications, this weaker checksum might still be sufficient, there could be other applications which require the stronger version. One way to address this issue could be to enhance the TCP protocol in the kernel with a stronger checksum. Another way could be to use a per-fragment 32-bit CRC checksum in the TCP path of the library, and cause retransmissions on detecting an error. However, this second technique would necessarily cause some performance degradation for regular transmissions.

The TCP path utilizes two TCP connections between every pair of nodes participating in a MPI application. On the other hand, the UDP uses just two sockets per node. This naturally implies higher connection management overhead as the number of nodes in the application increases. However, modern operating systems have highly tuned network protocol stacks, which should enable them to cater to hundreds of TCP connections simultaneously with minimal performance degradation. The problem of

TCP scalability is especially important in the domain of networking servers, which have to cater to several thousand clients at once, and has been extensively studied in that context. Such an analysis is outside the scope of this thesis.

Chapter 4
An Event-driven Architecture for LA-MPI

The previous chapter described an efficient TCP path for the LA-MPI library and shows how relying on TCP's reliability and flow-control can improve library performance. Using TCP for communication leads to both a decrease in message latency, and increase in message bandwidth for blocking MPI sends and receives, as shown by the results presented in Section 3.3. However, MPI applications use both blocking and non-blocking communication that are provided by the MPI Standard (Section 2.1). Non-blocking communication is especially important since it offers a way to overlap an application's computation and communication. Under this paradigm, an application initiates a non-blocking message send or receive, and then proceeds to perform some independent computation. The responsibility for making progress on the initiated messages lies with the library progress engine. Most existing MPI libraries (and all freely-available ones) make progress on these pending messages when the application calls into the library. Thus, if a message could not be sent to, or received from another node at the time of the first non-blocking send or receive call, the actual communication could be arbitrarily delayed until the application re-enters the library through the next MPI call. When the application invokes the library, the progress engine attempts to make as much progress on each pending message as possible. On the other hand, absolutely no communication progress takes place between calls into the MPI library by the application.

Such MPI implementations lead to several problems. First, the communication performance is tied to the frequency at which an application makes MPI library calls. If the application calls the library too often, the progress engine wastes resources in determining that there are no pending messages. On the other hand, if the application invokes the MPI library too rarely, communication latency increases leading to a potential drop in application performance. Furthermore, such library implementations also technically violate the MPI standard, which stipulates that communication progress for non-blocking messages must be made even if no other calls are made by the sender or receiver to complete the message send or receive, respectively.

An event-driven communication thread can improve performance by enabling messaging progress in a MPI library, independent of whether or not the application invokes the library layer again after the initial message send or receive call. An asynchronous event thread enables messages to be sent and received concurrently with application execution, thus making the twin tasks of a MPI application—communication and computation, truly concurrent. Such a technique can improve the responsiveness of the MPI library, and can also improve the application performance by decreasing message latency and increasing message bandwidth for non-blocking communication. An implementation of the aforementioned technique is presented in this chapter for the LA-MPI library using TCP-based communication.

The rest of this chapter proceeds as follows. Section 4.1 discusses some of the limitations of the LA-MPI library design. Section 4.2 presents a high-level introduction to the concept of event-driven software architecture, and Section 4.3 shows how an event-driven approach can be applied to the TCP path of the LA-MPI library. An evaluation of the event-driven LA-MPI library is presented in Section 4.4. Finally, Section 4.5 summarizes the chapter and its main contributions.

4.1 Limitations of Existing LA-MPI Design

In the existing LA-MPI library implementation communication tasks are almost entirely governed by the progress engine. However, the progress engine itself can execute only when the application makes a MPI library call. Therefore, the execution of the progress engine is not related to the occurrence of network events, and library communication is intrinsically linked to library invocation. This can be wasteful if library calls occur more often than network events. In that case, the entire progress engine must execute despite the fact that there may be nothing for it to do. The use of callback routines mitigates this cost to some extent, as these handlers will only be called when corresponding events do occur. However, every time the progress engine is executed, it incurs the overhead of calling into the operating system to check for events. On the other hand, if the application makes too few MPI library calls the communication performance would suffer because unless the progress engine is executed pending messages cannot be sent or received by the library. This creates a delicate balancing act for the programmer. If the library is called too frequently, resources will be wasted, and if the library is called too rarely, then messages will not be transferred promptly leading to unnecessary latency increases. From the programmer's perspective, the application should only be required to access the library when messaging functions are required, and not to enable the library to make progress on in-flight messages.

Furthermore, the MPI standard specifies that the library should make progress on non-blocking messages (send/receive), and complete them regardless of whether the application makes a subsequent MPI library call or not. The MPI standard v1.1 progress rule for non-blocking communication states—

> A call to `MPI_WAIT` that completes a receive will eventually terminate and return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is non-blocking, then the receive should complete even if no call is executed by the sender to complete the send. Similarly, a call to `MPI_WAIT` that completes a send will eventually return if a matching receive has been started, unless the receive is satisfied by another send, and even if no call is executed to complete the receive. [14]

The adherence to this progress rule for non-blocking messages is not possible under the auspices of the existing library design since the library does not provide any means for communication to proceed while the application code is being executed. For the library to perform any communication, the application has to make a library call, which in turn causes the progress engine to execute. Thus, in the absence of library invocations from the application, no progress is made on the pending messages in the library.

4.2 Event-driven Software Architecture

An event-driven software architecture consists of independent event handling routines, which only execute in response to events. Such an event-driven application, in its simplest form, consists of one central loop, known as the event loop. The event loop's main function is to detect events, and dispatch the appropriate event handlers when they occur. In the absence of an event, the event loop simply blocks waiting for one. In that case, the occurrence of an event wakes up the event loop, which then dispatches the appropriate event handler.

For example, a web server is an inherently event-driven application. A web server's main function is to accept and respond to client requests, which arrive over the network. So, a web server only executes code when network events occur—either the arrival of a request, or the availability of resources to send responses. In modern web servers, the event loop utilizes one of the operating system event-notification facilities, such as `select`, `kqueue`, or `epoll`, to detect network events [16, 30]. These facilities are centered around a system call that returns a list of pending events. If desired, the system call can block until such an event occurs. So, the event loop of a web server makes one of these blocking system calls to wait for an event. As soon as a client request arrives at the web server, or resources free up to send a response to a previous request (both of which cause a network event), the operating system wakes up the event loop by returning from the system call. The event loop then dispatches the appropriate handler to respond to the event. When the event handler completes, it returns control to the event loop, which either dispatches the next event handler if there were multiple events pending, or it blocks waiting for the next event.

An event-driven software architecture provides an efficient mechanism to deal with the latency and asynchrony inherent in network communication. This design eliminates the need for the application to query repeatedly about the occurrence of an event, making it possible for the application to instead accomplish other useful tasks while it waits for the event to occur. An added advantage is the scalability it offers in terms of the number of network events it can simultaneously monitor without undue performance degradation.

The LA-MPI library uses the notion of read and write events to wait for the availability of resources to send and receive message fragments, respectively. However, the occurrence of these events does not automatically trigger the library to send or receive message fragments. Instead, the library checks for these events only through its progress engine. Even after an interesting event has occurred, the library might not execute the handler until the next invocation of the library by the application, and subsequent execution of the progress engine. Thus, the LA-MPI library architecture is not strictly event-driven, since the tasks of sending or receiving fragments are not performed directly in response to the corresponding events.

4.3 An Event-driven LA-MPI Library (LA-MPI-ED)

An MPI application performs two main tasks: computation and communication. These tasks place different demands on the system, as computation is driven by the control flow of the program, and communication is driven by asynchronous network events. An MPI library provides such applications with two important services. First, the functional interface of the library provides the application a mechanism to transfer messages among nodes. Second, the progress engine of the library performs the communication tasks that actually transfer those messages. Almost all existing MPI libraries combine these tasks by executing the progress engine only when the application invokes the functional interface of the library. This model introduces some limitations in the library as discussed for the LA-MPI library in Section 4.1. This model favors the computation portion of the application, as communication progress is only made when the application explicitly yields control to the MPI library, rather than when network events actually occur. This is a reasonable trade-off as it would be difficult to efficiently handle the computation portion of the application in an event-driven manner. However, as described in Section 4.2, event-driven software architecture provides a convenient technique to perform communication tasks that are completely driven by asynchronous network events. The communication tasks of the MPI library are quite similar to the functions of a web server and thus, match the event-driven architecture model quite well.

This chapter of the thesis presents LA-MPI-ED, an event-driven version of the LA-MPI library. LA-MPI-ED makes use of the TCP path of the library, which has been altered accordingly to facilitate an event-driven approach towards TCP communication. LA-MPI-ED separates the computation and communication tasks of a MPI application. This enables the communication portion of the library to use the event-driven model, and the functional interface of the library to work synchronously with the computation portion of the application. The two tasks of the library occur in separate threads. The main thread (MT) executes the application and the functional interface of the library, as normal. The event thread (ET) executes the communication tasks of the library in an event-driven fashion. When sending messages, the main thread notifies the event thread that there is a new message to be sent, and the event thread sends it. When receiving messages, the event thread accepts the message, and then notifies the main thread that it has arrived when the application invokes the library to receive the message. In this manner, the most efficient software model can be used for both components of the MPI application. This also facilitates greater concurrency between executing the MPI application and executing the progress engine of the MPI library.

During the time the TCP messaging layer for LA-MPI was being developed here at Rice, the LA-MPI team at LANL developed a TCP path for the library independently. This was incorporated in their next major release of the library, version 1.4.5. Their implementation of the TCP path follows the same principles that guided the implementation described in Section 3.2. However, version 1.4.5 of the LA-MPI library also incorporates several performance optimizations in the MML of the library, as well as a significant clean-up of the source code in general. These have no direct bearing on the design of the TCP messaging layer. However, to ensure maximum utility of this research, the LA-MPI-ED development is performed on top of the LA-MPI v1.4.5 library, instead of the earlier version. All subsequent references to the LA-MPI library implicitly refer to version 1.4.5, unless explicitly mentioned otherwise.

4.3.1 Design of LA-MPI-ED

The LA-MPI-ED library utilizes two threads of execution to perform communication tasks and communication tasks of a MPI application independently and concurrently. The main thread (MT) provides the synchronous aspects of a MPI library and executes the core of the library, including the application linked against the library. Thus, the MT is responsible for performing library tasks such as initialization, keeping track of pending messages, posting messages to be sent or received, etc. The event-driven progress engine of the library executes in the context of the event thread (ET) and thus, can run concurrently and independent of the MT. The ET, by virtue of executing the progress engine, is responsible for handling all the asynchronous network events, which affect the state of the library. The MT spawns the ET during library initialization and both threads exist until the end of execution of the MPI application. The ET monitors all connected sockets for incoming data (read events), and the sockets over which messages are being sent for available buffer space (write events). Thus, the ET is ultimately responsible for the transmission and reception of messages whenever resources become available. The threads communicate with each other through the use of shared queues of data structures. Using light-weight threads, such as POSIX threads (pthreads), minimizes the overhead of thread switching, and protected accesses to the shared data. The programming effort of sharing data between the two threads is also minimized since pthreads execute in the same application address space by design. The event-driven library can also support multi-threaded MPI applications with one event thread catering to all the threads of the application. The support for multi-threaded applications in the library is discussed in greater detail in Section 4.3.3.

Conceptually, the MT runs continuously until the application makes a blocking MPI library call, such as `MPI_Send`, `MPI_Recv`, or `MPI_Wait`. A blocking MPI call signifies that the application needs to wait for the library to accomplish a certain task before it proceeds. In this case, the MT blocks on a condition variable using a pthread library call until the pending request has been completed. The ET then gets scheduled by the thread scheduler for execution, and wakes up the MT upon completion of the request. In contrast, the ET runs only when there is a network event of interest to the library. In the absence of an interesting event the ET is kept blocked on the `select` system call. The occurrence of an event causes the operating system to wake up the ET, which then invokes the appropriate event handler to process pending messages. In practice, if both threads have work to be done, the thread scheduler must allow them to share the system's resources. In that case, the responsiveness of the ET can be increased by running it at a higher priority than the

MT.

The TCP path of LA-MPI already utilizes three event lists—read, write and except—to form the components of a blocking `select` call to retrieve events from the operating system. These event-lists are populated by the MT based on user-requests. Some events, such as read events, are always kept active on these lists, since a MPI process can receive a message at any time. The write events are added as and when there are message send requests. The ET uses these event-lists to make the blocking `select` system call. In case the MT updates its interest on some event while the ET is already blocked on the `select` system call, the MT wakes up the ET with the `SIGCONT` signal to enable the ET to use the latest event lists. Thus, the notion of events in the TCP path complements the event-driven architecture of the progress engine very nicely. The event-driven approach in the design of the progress engine also provides an efficient means to exploit even more the event-centric design of the TCP path. Separating the handling of network events into the ET incurs minimal alterations to the structure and functionality of the LA-MPI TCP path. It still performs the tasks, such as connection establishment, message transmission, and message reception exactly as before (Sections 3.2.2, 3.2.3, and 3.2.4, respectively). The only difference with the event-driven progress engine is that execution of these tasks is now split between the MT and the ET, based on whether the task is performed synchronously by the library or happens as a result of an asynchronous network event. The event-driven nature of the progress engine significantly alters only the way the library interacts with the network. Thus, instead of having the progress engine poll for events every time it is invoked, the progress engine waits for the occurrence of an event and processes it immediately.

LA-MPI-ED, like LA-MPI, utilizes events to send or receive message fragments. However, as explained in Section 4.2, LA-MPI is not event-driven since the execution of the library is governed by the MPI application, and not by the occurrence of these events. On the other hand, in LA-MPI-ED, all message communication in the library is performed by the ET in direct (and immediate) response to such events. This makes communication in LA-MPI-ED truly event-driven in nature. The computation tasks of the application are still performed synchronously, and independent to communication, by the MT of the library.

The design of LA-MPI-ED relies heavily on the notion of events in the TCP path. Since the other paths within LA-MPI do not utilize network events, they do not easily fit into the event model. Messages transferred using other paths in the library currently revert to using the MT for all tasks, including communication. To enable specialized networking hardware, such as Quadrics or Myrinet, to utilize the event-driven model, those paths within LA-MPI, possibly including device firmware, would need to be rewritten to use network events for communication. While this is certainly possible and would likely improve communication performance, it is beyond the scope of this thesis.

4.3.2 Example

Figure 4.1 illustrates the operational differences between the two library versions—LA-MPI-TCP and LA-MPI-ED. Subfigure (i) shows a simple MPI request sequence executing at a particular node. Subfigure (ii) shows the progression of this MPI request sequence in both versions of the library. For ease of understanding, the pro-

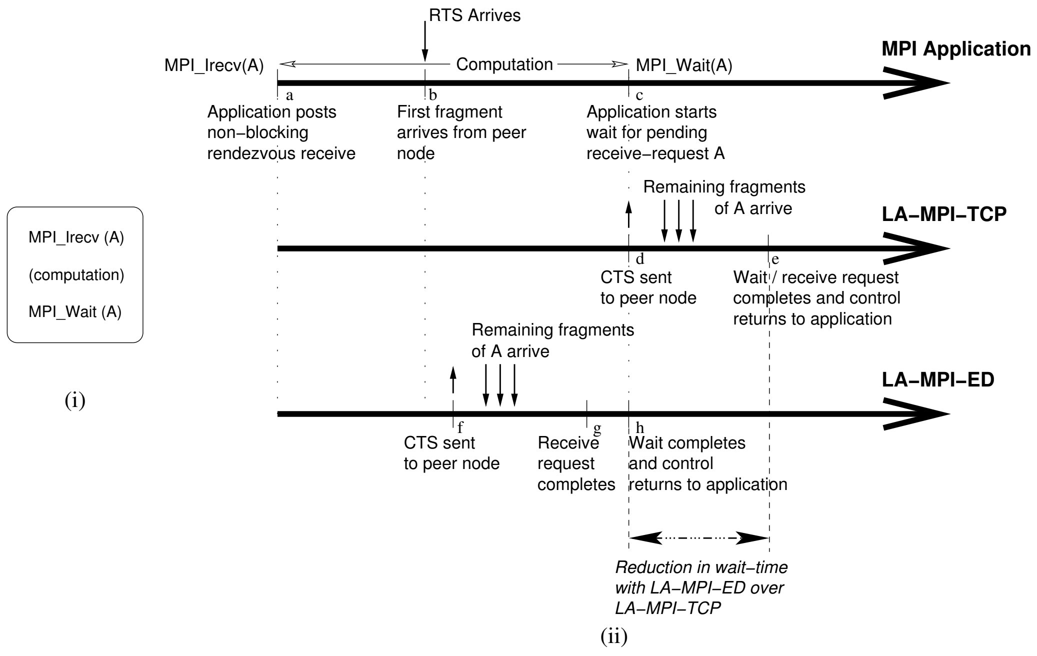
RTS Arrives
MPI_Irecv(A)
Computation
MPI_Wait(A)
MPI Application
a
Application posts non−blocking rendezvous receive
b
First fragment arrives from peer node
c
Application starts wait for pending receive−request A
Remaining fragments of A arrive
LA−MPI−TCP
d
CTS sent to peer node
e
Wait / receive request completes and control returns to application
MPI_Irecv (A)
(computation)
MPI_Wait (A)
(i)
Remaining fragments of A arrive
LA−MPI−ED
f
CTS sent to peer node
g
Receive request completes
h
Wait completes and control returns to application
Reduction in wait−time with LA−MPI−ED over LA−MPI−TCP
(ii)

Figure 4.1 A simple MPI request sequence and its progression in LA-MPI-TCP and LA-MPI-ED libraries

gression is shown on 3 different timelines. The topmost timeline shows the sequence of steps performed by the MPI application and events that are external to the library (such as arrival of the RTS message). The middle and the bottom timeline show the steps as they occur in LA-MPI-TCP and LA-MPI-ED respectively. As shown in the topmost timeline, the MPI application posts a *rendezvous* receive request for message A at time-step a. Then the application performs some computation tasks before invoking the MPI wait call at time-step c to wait for the arrival of message A. At some time-step b, between a and c, the node receives the RTS message from the peer node in the form of the first fragment of the posted receive.

As mentioned in Section 4.3.1, the event-driven progress engine only alters the way in which the library interacts with and responds to network events. The behavior of LA-MPI-TCP and LA-MPI-ED differs when the RTS message arrives from the peer node. The LA-MPI-TCP library takes no action in response to the RTS message arriving at the node until time-step $d\,(= c)$, when the application makes the `MPI_Wait` call. At that point, the RTS is received and the CTS message sent to the peer node. After this, the library spins in the progress engine as it waits for the rest of the fragments of message A to arrive. The entire message is received at this node by time-step e. This completes the receive request along with the wait call and control is returned to the application.

With an event-driven asynchronous progress engine, the LA-MPI-ED library starts to process the RTS message immediately upon receiving it and sends the CTS back to the peer node at time-step $f\,(< c,\ d)$. Note that at this time, the application (and MT) is still busy with its computation tasks, but the ET is free to handle incoming data. This enables the ET to perform communication concurrently with the computation being performed by MT. Finally, at time-step g, the receive of message A is completed and the ET goes back to waiting for the next network event. The MT meanwhile finishes its computation and makes the wait call at time-step h $(= c,\ d;\ > g)$. Since by this time the receive of message A is already complete, the wait call returns back to the application immediately. As a consequence, there is a significant reduction in effective wait-time with the LA-MPI-ED library as pointed out in the figure.

Even though the example only illustrates the operational distinctions between the two versions of the library in performing a non-blocking receive, non-blocking sends also exhibit almost identical behavior. With the event-driven progress engine, the ET can continue sending fragments of a message without requiring the completion of the computation tasks of the MPI application. Collective communication in the

LA-MPI library is layered on top of the point-to-point communication support, and thus, would behave in an identical fashion. As such there is no special provision in the event-driven LA-MPI library to handle collective communication.

The example presented in Figure 4.1 is relatively simple and is only intended to convey a hint about the behavior of an event-driven asynchronous progress engine. In practice, in LA-MPI-ED when computation and communication proceed concurrently, the net time spent for the computation task would increase because the execution of the progress engine would take away compute cycles from the application. As a result, the application would make the wait call later, so time-step h would shift further right along the timeline. However, as long as the overlap of computation and communication more than offsets the increase in effective computation time, this technique provides a performance benefit over the LA-MPI-TCP library.

4.3.3 Performance Implications of LA-MPI-ED

The event-driven LA-MPI library effectively satisfies the objective of separating the synchronous and asynchronous aspects of the library, and handling them independently. An event-driven progress engine is much more responsive to network events, and can much more efficiently transmit and receive messages. As an added advantage the progress engine thread, which runs independent of and concurrent to the core of the library, is now able to process pending requests even in the absence of library invocations by the application.

The thread library does impose a limited amount of overhead for thread switching and thread management tasks. However, for a well designed MPI application, the improvement in performance due to the concurrency of computation and communication usually offsets this overhead. It also relieves the programmer from having to worry about enabling the library to make progress on outstanding messages.

MPI applications can also be very sensitive to message latency, especially for the really short messages, which are frequently used to implement communication barriers. The extra latency introduced by thread switching on the send path can thus potentially hamper application performance. The event-driven LA-MPI library tries to mitigate this shortcoming of its threaded design by a simple optimization—the MT, after posting a message to be send, optimistically tries to send it, as well. If the socket buffer has enough space for the message, the message can be sent immediately without the use of the progress engine. This ensures that whenever there is available space on the socket buffer, which will be the case for most single fragment messages, the message will be sent directly without incurring any thread switching overhead. The event thread will be used to send messages only when there is not enough space in the socket buffer to send the first fragment immediately, or for subsequent fragments of a multi-fragment message. This optimization effectively makes the send side performance of the event-driven and non-event-driven libraries equivalent.

The addition of an asynchronous event-management thread in the event-driven LA-MPI library does not alter the original library's support for multi-threaded MPI applications. Since different threads of a multi-threaded MPI application can be executing the progress engine simultaneously, the correct handling of message communication by the library requires special consideration. The original LA-MPI library ensures correctness by allowing access to message queues only one thread at a time. Thus, multiple threads executing the progress engine iterate through the lists of pending requests one after the other, which ensures their consistency across the different threads. Furthermore, progress on any particular message request is not tied to the thread that initiated this request. Hence, all pending requests in the library are progressed whenever any application thread invokes the progress engine. If there are multiple threads waiting for the completion of different requests simultaneously, the completion of any particular request only causes the corresponding thread to return to the application, while the other threads continue to wait in the progress engine. The event-driven LA-MPI library provides identical behavior for threaded MPI applications by having one event thread to progress requests initiated by all application threads. The only case that requires special consideration is when multiple application threads are sleeping during blocking MPI calls. Since the event thread does not know in advance which request a particular thread is waiting on, all sleeping threads are woken up by the event thread on completion of a request. Each thread re-checks the status of the request immediately after being woken up, and thus, only the thread whose request was completed resumes execution; the remaining threads return to sleep until the event thread awakens them the next time, repeating the process.

In addition to improving the efficiency and responsiveness of communication, an independent messaging thread can also improve the functionality of the MPI library. The event thread could also be used for such tasks as run-time performance monitoring, statistics gathering, and improved run-time library services. Furthermore, in the event of a dropped TCP connection, the ET could also reestablish the connection faster and further improve messaging performance.

4.4 Evaluation of LA-MPI-ED

This section presents an evaluation of the LA-MPI-ED library and compares it against the TCP path of version 1.4.5 of the LA-MPI library. The evaluation is performed using both microbenchmarks and application benchmarks. The microbenchmarks are designed to test for the ping latency, the one-way communication bandwidth of the MPI library, and the ability of the MPI library to overlap communication with computation, all on a gigabit Ethernet network. The application benchmarks are from the NAS set of parallel benchmarks [1]. The microbenchmarks as well as the application benchmarks used in this evaluation are explained in greater detail in Section 2.6.

The evaluation is performed on a FreeBSD workstation cluster employing up to 9 processing nodes. Each workstation has one AMD Athlon XP 2800+ processor, 1GB DDR SDRAM, and a 64bit/66Mhz PCI bus. Each of the nodes also has at least 40GB of hard drive capacity (none of the benchmarks are disk intensive). Each workstation also has one Intel Pro/1000 Gigabit Ethernet network adapter. The nodes are connected to each other through a 24-port Gigabit Ethernet switch. Each node runs an unmodified FreeBSD-4.7 operating system with various socket and networking parameters tuned in order to provide maximum TCP/IP performance. This includes enabling large window sizes in TCP (RFC1323), setting maximum allowable socket buffer size to 512 KB, and tuning related kernel parameters, such as *nmbclusters*, *maxsockets*, etc. LA-MPI-ED uses the default POSIX thread library implementation in FreeBSD (`libc_r`). In this section, as in earlier sections, LA-MPI-ED is refers to the event-driven asynchronous LA-MPI library and LA-MPI-TCP refers to the

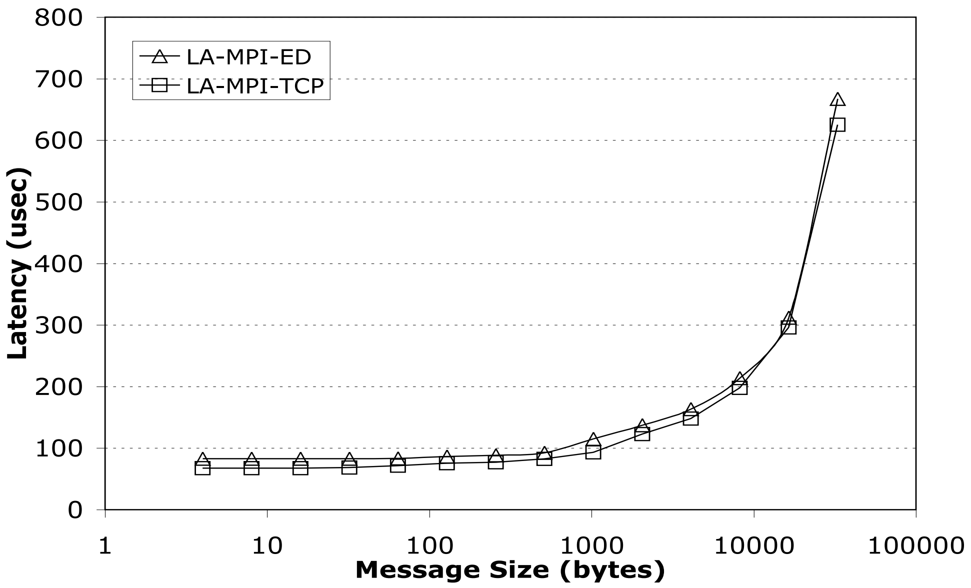

Figure 4.2 Comparison of ping-latency between LA-MPI-TCP and LA-MPI-ED

LA-MPI library with only its TCP path active.

4.4.1 Microbenchmark Performance

The message ping-latency of a MPI library is measured using the *ping-pong* microbenchmark described in Section 2.6. Figure 4.2 shows the ping latency for 4 B-32 KB messages, for both LA-MPI-ED and LA-MPI-TCP libraries. The figure shows that the ping latency of messages with LA-MPI-ED is always slightly higher than with LA-MPI-TCP. The difference in ping latencies between the two libraries is nearly constant at 15 μsec for all message sizes between 4 byte to 16 KB; this difference stems from thread-switching overheads in LA-MPI-ED. Although this is a 23% increase for 4-byte message latency, it is only 5% for 16 KB messages. Beyond 16 KB, the ping latency for both library versions more than doubles as the messaging protocol shifts from *eager* to *rendezvous* and the number of message fragments increases from one to two (see Table 2.1). Again, because of thread-switching overhead on the re-

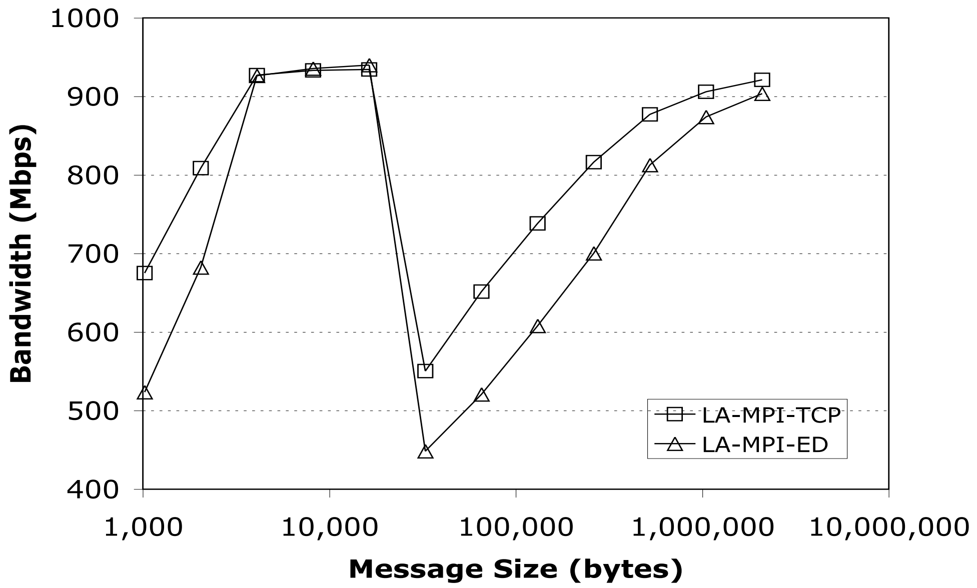

Figure 4.3 Comparison of unidirectional message bandwidth between LA-MPI-TCP and LA-MPI-ED

ceive path, the ping latency in LA-MPI-ED is slightly higher than in LA-MPI-TCP (approximately 42 μsec or 6% difference).

The microbenchmark used to measure the unidirectional bandwidth of a MPI library was also described in Section 2.6. Figure 4.3 shows the unidirectional message bandwidths obtained on this benchmark by the LA-MPI-TCP and LA-MPI-ED libraries for message sizes ranging from 1 KB to 2 MB. Each data point is obtained by iterating the benchmark over 1000 messages. LA-MPI-TCP always outperforms LA-MPI-ED, with performance gaps up to 22% at 1 KB messages and 19% at 32 KB messages (when the rendezvous protocol is first invoked), but with smaller gaps for larger messages within each sending protocol. Besides the thread library overhead, LA-MPI-ED suffers from a greater number of unexpected message receives, each of which results in extra memory copies. The LA-MPI-TCP library, which executes its progress engine synchronously with calls into the library, first posts the message

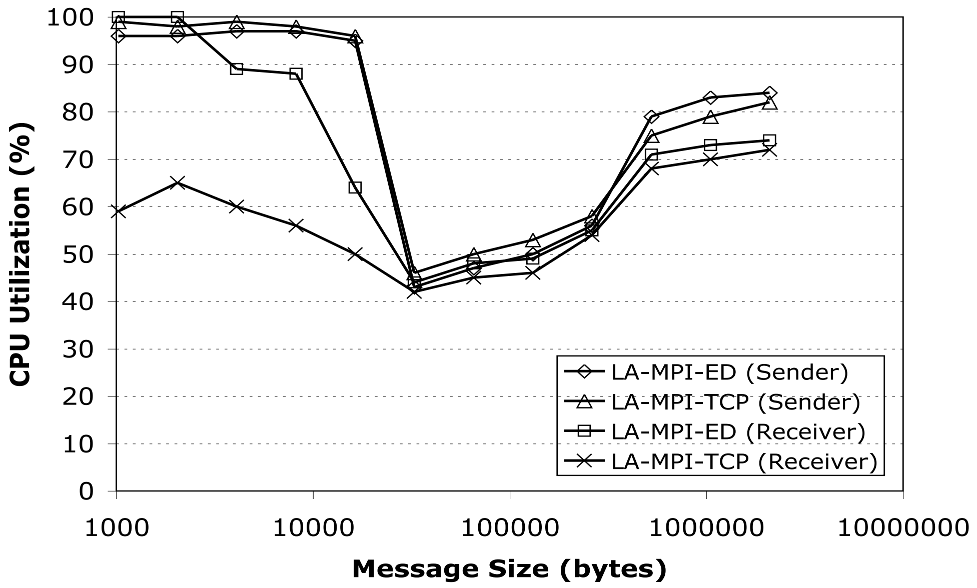

Figure 4.4 Comparison of CPU utilization for unidirectional bandwidth test between LA-MPI-TCP and LA-MPI-ED

receive, and then reads the incoming message by invoking the progress engine. Even if the message arrives earlier, the library would only read the message from the socket at this time, and hence encounters all expected receives, which are read directly into the posted buffer. On the other hand, the LA-MPI-ED library receives a message as soon as it arrives on the socket and thus, invariably ends up receiving the message into a library buffer before the actual application buffer has been posted. The simplicity of the benchmark causes this drop in performance as a side effect to the increased responsiveness of LA-MPI-ED. Under the rendezvous protocol, only the first fragment of any message is received unexpected. Thus, the performance difference between LA-MPI-TCP and LA-MPI-ED drops with message size and is only 2% for 2 MB messages.

Figure 4.4 shows the CPU utilization, of both the sender and receiver, for the unidirectional bandwidth test with the LA-MPI-TCP and LA-MPI-ED libraries, respectively. The base LA-MPI library performs blocking MPI library calls by constantly spinning in the progress engine, repeatedly iterating over all pending requests and trying to make progress on all active paths in the library. This results in 100% CPU utilization with any MPI application as the progress engine remains active whenever the library (or application) is not performing other tasks. However, for comparing the effective CPU utilization (time spent by CPU doing useful work) of LA-MPI-TCP and LA-MPI-ED, the base library is modified suitably to block on the `select` system call while it waits for the completion of an incomplete request. Note that in general such a modification compromises library correctness, since it assumes that there is only one pending request active in the library at any given time. However, for the simple bandwidth benchmark employed here, this assumption is valid, and does not affect library functionality in any way. Figure 4.4 shows that the sending side, with either version of the library, has almost equal CPU utilization across the range of message sizes shown. At 32 KB message size, the sender's CPU utilization with the LA-MPI-ED library is about 4% below that of the LA-MPI-TCP library. At this message size the *rendezvous* protocol comes into effect. The receiver receives the first fragment as an unexpected fragment and delays the transmission of the CTS until the corresponding receive is posted. This causes lower CPU utilization on the sending side with the event-driven library as the library waits for the CTS message before proceeding with the remaining fragments. With increasing message sizes, the relative fraction of this additional waiting-time diminishes, resulting in converging CPU utilization of the sender with either versions of the library. For the receiver, at small message sizes the thread context-switch overhead is substantial and results in considerably higher CPU utilization with the event-driven library. However, the effect of this additional overhead is largely marginalized with the *rendezvous* protocol, and consequently for message sizes of 32 KB and beyond the CPU utilization plots of the two libraries are almost coincident.

Figures 4.5 and 4.7 show the same bandwidth comparison between LA-MPI-TCP and LA-MPI-ED for the cases where all receives are expected and unexpected, respectively. Both these graphs show that the performance of the two library versions is almost matched except at small message sizes. When message size is small, the thread library overhead is a higher percentage of the total communication time and causes a 8-10% drop in bandwidth performance with LA-MPI-ED. A comparison of the graphs also reveals that unexpected receives can cause up to 20% drop in library bandwidth as compared to expected receives. Figure 4.6 and 4.8 show the comparison of CPU utilization between LA-MPI-TCP and LA-MPI-ED, on both the sender and the receiver, for the above two variants of the basic unidirectional bandwidth test. These figures also show that the thread library overhead manifests itself most clearly for small sized messages at the receiving node. With increasing message sizes, the impact of this overhead on CPU utilization is increasingly reduced, and LA-MPI-TCP and LA-MPI-ED achieve almost the same CPU utilization.

One of the main advantages of the event-driven LA-MPI library is its ability to overlap computation and communication, both of which are essential to MPI applications. However, this advantage cannot be seen in microbenchmarks that only perform communication. Figure 4.9 presents pseudo-code for a microbenchmark that highlights the benefits of LA-MPI-ED by carefully balancing computation and com-

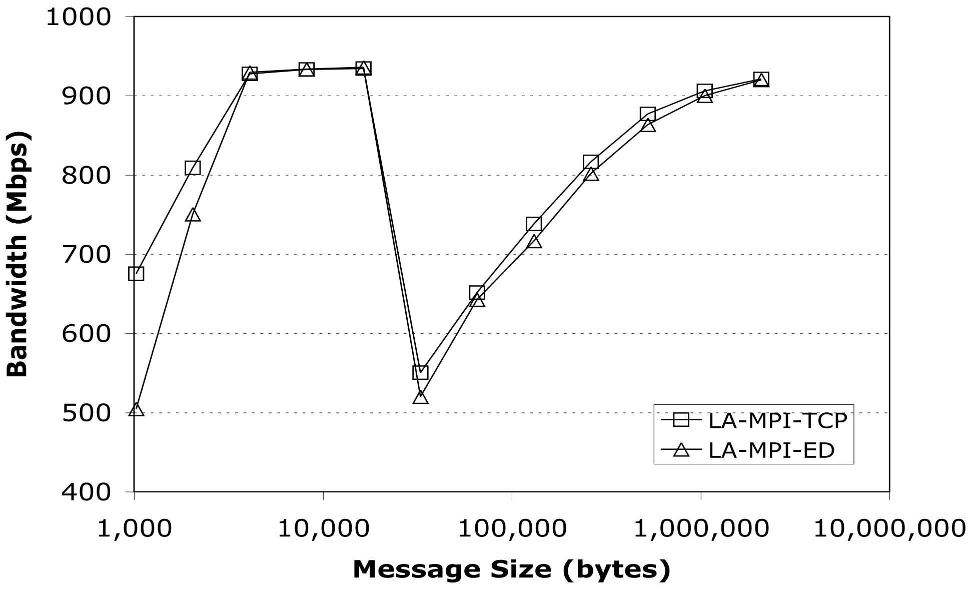

Figure 4.5 Comparison of message bandwidth between LA-MPI-TCP and LA-MPI-ED with all expected messages at the receiver

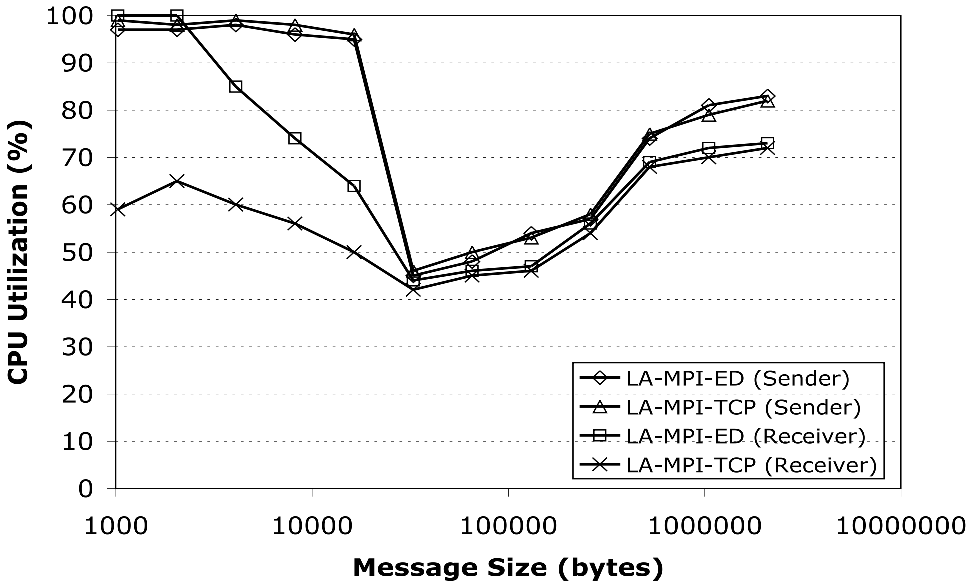

Figure 4.6 Comparison of CPU utilization for unidirectional bandwidth test (all expected receives) between LA-MPI-TCP and LA-MPI-ED

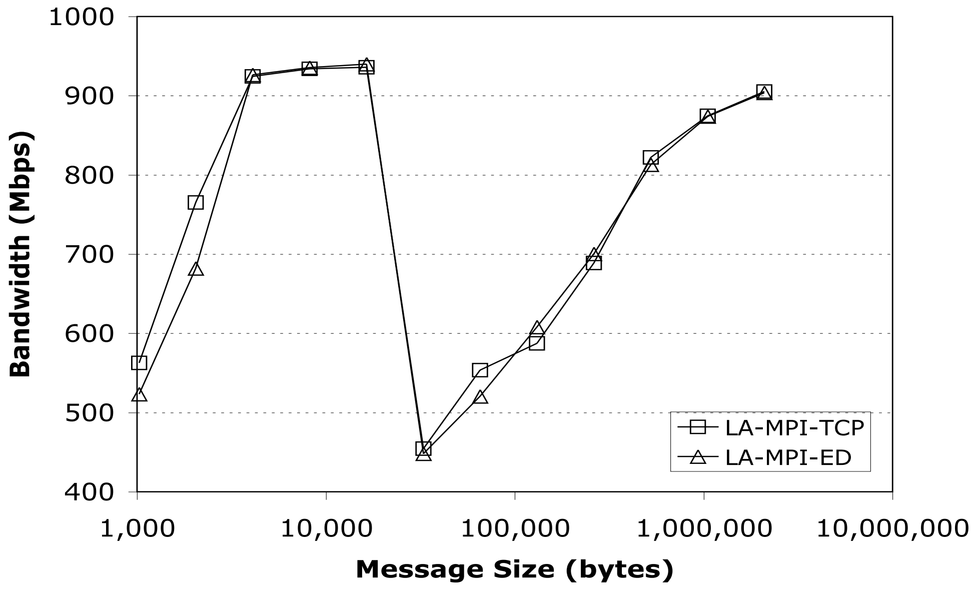

Figure 4.7 Comparison of message bandwidth between LA-MPI-TCP and LA-MPI-ED with all unexpected messages at the receiver

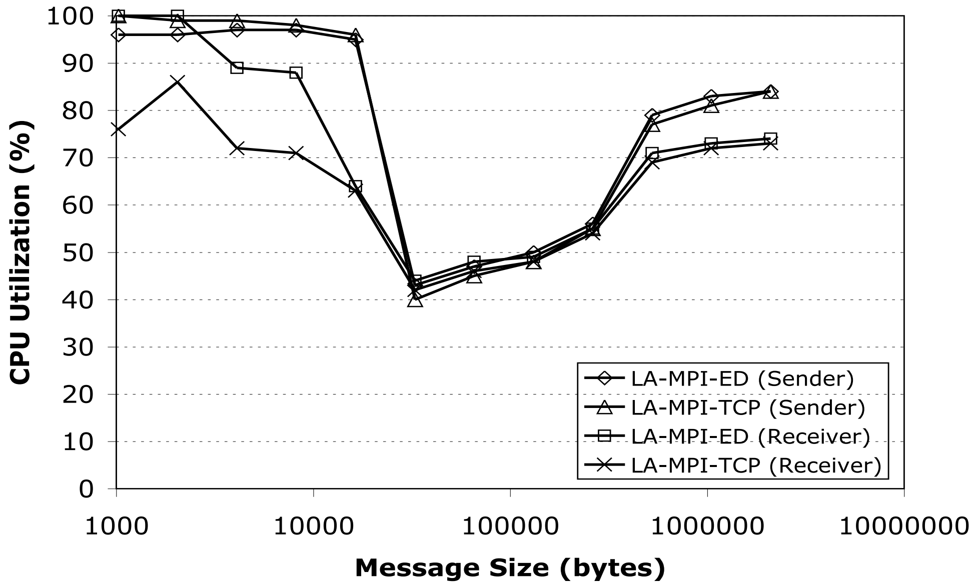

Figure 4.8 Comparison of CPU utilization for unidirectional bandwidth test (all unexpected receives) between LA-MPI-TCP and LA-MPI-ED

```
Sender
i = 0;
while (i++ < NUM_TESTS)
{
   MPI_Send(A);

   MPI_Recv(B);
}
```

```
Receiver
i = 0;
while (i++ < NUM_TESTS)
{
   MPI_Irecv(A);

   /* computation */

   MPI_Wait(A);
   MPI_Send(B);
}
```

Figure 4.9 Pseudo-Code for the microbenchmark used to evaluate the advantages of LA-MPI-ED over LA-MPI-TCP

munication. The microbenchmark involves two MPI nodes—one sender and one receiver. The sender posts a message of user specified size to be sent to the receiver with the blocking MPI send message routine. The receiver posts the corresponding receive through a non-blocking MPI receive message call, and then proceeds to perform computation. The amount of computation performed is user specified in the form of integer increments. After the computation is completed, the receiver waits for receive request to complete through the MPI wait call. Finally, it sends a 4 byte message back to the sender which enables the sender and the receiver to synchronize their iterations through this microbenchmark loop. Statistics are collected at runtime from the library at the receiver node. Specifically, it collects (i) the duration of the wait call (wait-time), and (ii) total time from the receive being posted to the completion of wait for that particular receive request (total-receive-time). Both the wait-time and total-receive-time values are averaged over the number of iterations of the microbenchmark.

Figure 4.10 shows the percentage change in wait-time with increasing message

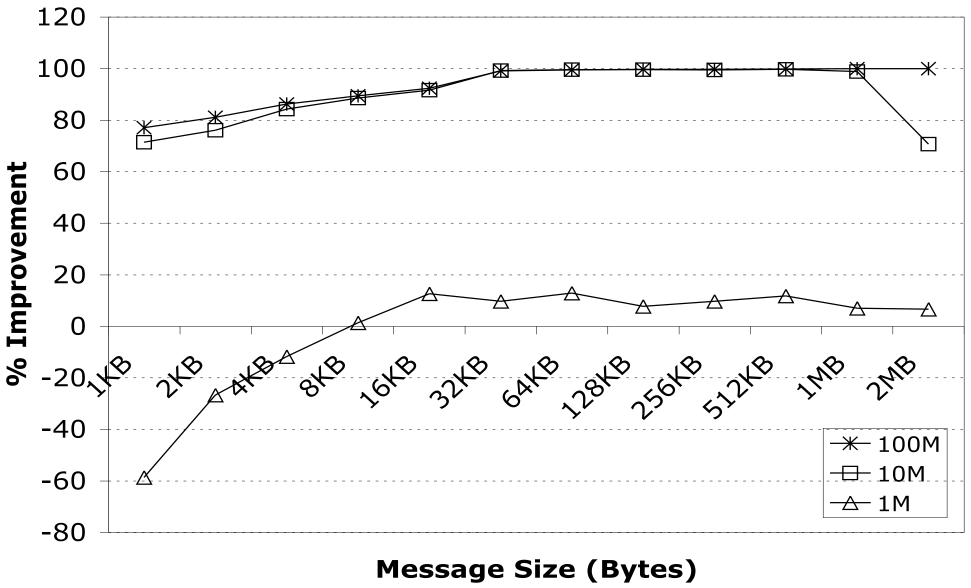

Figure 4.10 Percentage improvement in wait-time of the microbenchmark with LA-MPI-ED

sizes for LA-MPI-ED over LA-MPI-TCP. The three plots shown in the graph are obtained for different computation amounts of 1 million increments (1M), 10 million increments (10M) and 100 million increments (100M). The 1M plot shows a degradation for small message sizes because the wire latency for these messages is itself comparable to the computation times. Thus, the message arrives at the receiver too late to effectively overlap with computation, exposing the thread switching overhead. However, with increasing message sizes, the communication time increases and message receive is able to overlap with computation more. This results in the reduction in wait-time for message sizes beyond 8 KB for LA-MPI-ED over LA-MPI-TCP. This trend continues until message size reaches 64 KB, at which point the overlap between communication and computation is maximized. Beyond this, the communication time is larger than the computation time, so the overlap between them gets limited to the amount of computation, and percentage improvement of wait-time falls a bit. For

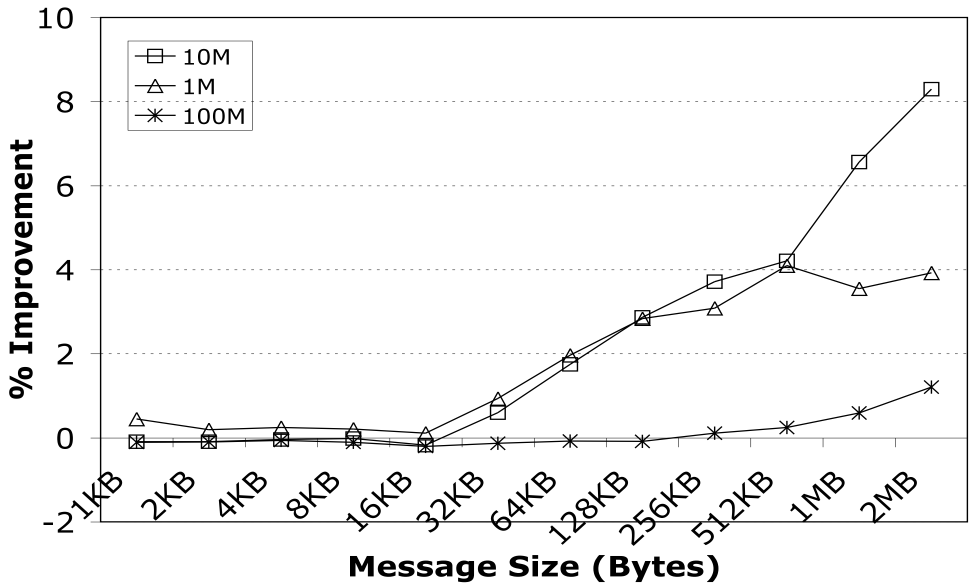

Figure 4.11 Percentage improvement in total-receive-time of the microbenchmark with LA-MPI-ED

the 10M and 100M plots, the computation time is large enough to overlap almost all of the communication time. This results in almost complete elimination of the wait-times with LA-MPI-ED. However, even for the 10M plot there is a drop in the percentage improvement at the highest message size (2 MB). At this message size the communication time exceeds the computation time and hence, the percentage wait-time improvement of LA-MPI-ED over LA-MPI-TCP falls. The 100M plot would show the same characteristic but for a much larger message, whose communication time would be comparable to the time required for 100 million increments.

Figure 4.11 shows the percentage improvement in the total-receive-time of LA-MPI-ED over LA-MPI-TCP, again as a function of message sizes. The three plots in the graph again correspond to varying amounts of computation. Again, as in Figure 4.10, the benefits of the event-driven library become apparent only when the amount of computation is high enough to overlap the communication time ef-

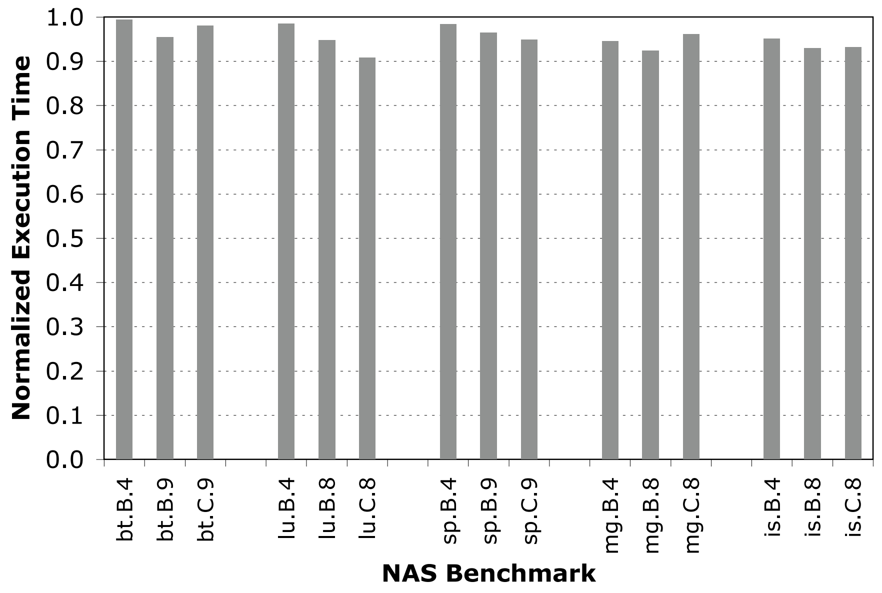

Figure 4.12 Execution times of NAS benchmarks with LA-MPI-ED normalized to LA-MPI-TCP (bars shorter than 1 indicate a speedup for that particular benchmark with LA-MPI-ED over LA-MPI-TCP)

fectively. For a particular amount of computation, the percentage improvement of total-receive-time increases with increasing message sizes. However, with sufficiently large messages, the communication time eventually becomes a magnitude greater than the computation time causing the plot to flatten out. This is seen for the 1M plot beyond 256 KB message size. The other plots do not show this behavior because the range of message sizes is never large enough to reach this stage. The 100M plot shows significantly lower improvement than the others as the computation time dominates and even a large reduction in the wait-time only has a very limited effect on the total-receive-time.

4.4.2 NAS Benchmarks

As described in Section 2.6, the NAS parallel benchmarks (NPB) is a set of 8 benchmarks, comprised of five application kernels and three simulated computation fluid dynamics (CFD) applications [1, 2]. They are a more realistic set of benchmarks than the microbenchmarks used on the previous subsection, which include both computation and communication. This particular evaluation of LA-MPI-ED, and comparison against LA-MPI-TCP uses five benchmarks from the NPB version 2.2 suite—BT, SP, LU, IS, and MG. The first three benchmarks among these five are simulated CFD applications and the latter two are smaller application kernels. As previously mentioned for the NPB version 2.2, each benchmark supports three pre-compiled data-sets, A, B, and C (in increasing order of size). These benchmarks are run on a 4, 8 or 9 node cluster, depending upon whether the benchmark requires a squared or power-of-2 number of nodes. The 4-node experiments are all run with the mid-size data-set (B). The 8-node (or 9) experiments are run for both the B data-set and the larger C data-set.

Figure 4.12 compares the performance of the five NAS benchmarks achieved with LA-MPI-ED against LA-MPI-TCP. Each bar of the graph corresponds to one particular configuration of the benchmark and is named accordingly—the first part of name gives the benchmark name, the middle part conveys the data-set size used for that particular experiment, and the last part conveys the number of participating MPI nodes for the experiment. Each bar of the graph depicts the normalized execution time of the corresponding benchmark using LA-MPI-ED relative to LA-MPI-TCP. Thus, a bar of height less than 1.0 indicates a performance gain with the event-driven library. Each data point for the graph is generated by averaging the execution times over 10 runs of the experiment.

Figure 4.12 shows that the benchmarks obtain varying degrees of performance improvement with the LA-MPI-ED library over LA-MPI-TCP. Each of the 15 benchmark configurations show an improvement with LA-MPI-ED over LA-MPI-TCP, ranging from 0.5% for BT.B.4 to 9.2% for LU.C.8, yielding an average speedup of 4.5%. Overall, the IS benchmark is the best performer with the event-driven library, registering an average speedup of 6.2%. The other 4 benchmarks achieve average speedups in the range of 2.3–5.6%. Among each individual benchmark, for the same data-set size, running the experiment over a larger cluster improves the relative performance of LA-MPI-ED to LA-MPI-TCP. Moving to a larger cluster for the same benchmark with the same data-set size increases the communication component of the application with respect to the computation performed at any given node in the cluster. Thus, the event-driven library, with its more efficient communication mechanism, reaps greater benefits. On the other hand, going to a bigger data-set for the same benchmark, increases the computation component of the application more than the communication component. Thus, as shown in Figure 4.12, the relative performance improvement of LA-MPI-ED compared to LA-MPI sometimes increases and sometimes decreases when moving to the bigger data-set, depending on the particular application.

The results in Section 4.4.1 show that the benefits of the event-driven library depend on the amount of overlap between computation and communication. IS achieves greater benefits than the other NAS codes because its communication consists primarily of very large messages that are pre-posted before the benchmark goes into a computation phase, enabling effective overlap (see Table 2.2 and Figure 2.8). While the other benchmarks do not overlap communication with computation as much, they still do so to varying degrees, yielding noticeable speedups. The results also show that the performance advantage of the event-driven library over LA-MPI-TCP should increase with an increasing number of nodes in the message-passing cluster, making the event-driven library more scalable.

4.5 Summary

MPI libraries perform two different tasks on behalf of MPI applications—computation and communication. Computation is driven completely by the control flow of the program, and communication is driven by asynchronous network events. Consequently, these two tasks place significantly different demands on the system. Most MPI libraries (and all freely available ones) tie the two functionalities together and attempt to solve both through a synchronous computation model. While this technique is simple in its implementation, it is not efficient for communication performance. This single approach to solve two separate problems also leads to greater demands on the MPI application programmer, who must now enable the library to perform communication by engineering his application suitably. In addition, this approach violates the MPI standard, which stipulates that progression of pending messages in the library should be independent of library invocation by the MPI application.

This chapter presents a technique to separate the concerns of computation and communication in the library, and apply the best approach for each of them. Thus, computation is performed synchronously through the functional interface of the library. The communication portion of the library is handled by an event-driven independent thread in the library. This enables the communication in the library to occur in response to asynchronous network events rather than being tied to synchronous library calls. This increases communication responsiveness of the library and reduces message latencies of non-blocking MPI messages. At the same time the threaded approach also enables the MPI library to perform computation and communication concurrently.

The current implementation of the event-driven LA-MPI library focuses on providing the event-driven software model to handle MPI communication using TCP over an Ethernet network. For this purpose the LA-MPI-ED library utilizes TCP network events delivered by the operating system. This threaded implementation of the TCP path of the LA-MPI library dramatically improves library responsiveness to network events, almost eliminating the wait-times for non-blocking receives. The event-driven approach towards handling MPI communication results in an improvement in the average execution times of all 5 NAS benchmarks, with a peak improvement of 7.3%. The results also indicate that the greater the overlap between communication and computation the better is the improvement in application performance by the use of LA-MPI-ED.

Currently the event-driven model is only supported for the TCP path of LA-MPI because the operating system already provides for the delivery of network events to the user-level library as and when they occur. Specialized networking hardware, such as Quadrics and Myrinet, which typically use user-level communication protocols, would require a significant rewriting of those protocols as well as of the device firmware to use the event-driven communication model. While this is cumbersome to do, it is certainly possible, and the event-driven approach towards communication is certainly likely to improve communication performance on these hardware as well.

Chapter 5
Comparing Ethernet and Myrinet for MPI Communication

The previous chapter presented a technique for efficiently handling TCP messaging over Ethernet in a MPI library, which results in higher performance on MPI applications. However, traditionally specialized networks, such as Myrinet and Quadrics, which provide lower latency and higher bandwidth than commodity Ethernet, have been used to facilitate high-performance MPI computing [4, 33]. These specialized networks utilize custom user-level communication protocols which enable them to bypass the operating system completely, and communicate with the network directly from the application-space. As a consequence, these custom networking solutions avoid expensive memory copies between user-space and kernel-space, and network I/O interrupts. TCP-based solutions utilizing the socket interface to the operating system incur interrupt processing overhead within the TCP stack of the operating system, and currently copy data on both the send and receive path, both resulting in higher messaging latencies.

The common perception is that because specialized networks provide significantly lower message latencies and higher message bandwidths than TCP over Ethernet, they will automatically deliver higher application performance. While this might have been true when the disparity between them and Ethernet was substantial, it is no longer necessarily true with the advent of 1–10 Gbps Ethernet. TCP is a carefully developed network protocol that gracefully handles lost packets and flow control. User-level protocols can not hope to achieve the efficiency of TCP when dealing with these issues. Furthermore, there has been significant work in the network server domain to optimize the use of TCP for efficient communication. This chapter of the thesis evaluates the performance differences between 1.2 Gbps Myrinet and TCP over 1 Gbps Ethernet as a communication medium for the LA-MPI library. To ensure a fair comparison, the Myrinet port of the MPICH MPI library v1.2.5..12 is also used for this evaluation. For both libraries, Myrinet communication is provided by a user-level Myrinet communication library and accompanying firmware for a Myrinet network interface card. LA-MPI also provides reliability for Myrinet communication within the library itself. While Myrinet ports exist for a number of publicly available MPI libraries, MPICH was chosen because its Myrinet support was developed by Myricom, Inc., the company which develops the Myrinet technology, and thus, is highly tuned for performance.

Besides these two library versions, the event-driven version of LA-MPI for TCP messaging over Ethernet (LA-MPI-ED), described in Chapter 4, is also used for comparison. Since LA-MPI-ED makes use of TCP in a much more efficient manner using techniques derived from the state-of-the-art in the network server domain, it is much more indicative of the performance possibilities of TCP over Ethernet.

The rest of this chapter proceeds as follows. Section 5.1 compares the message ping latency and unidirectional message bandwidth with the various MPI libraries mentioned above. Section 5.2 evaluates the various MPI libraries on the NAS benchmark suite. Finally, Section 5.3 summarizes the results of the evaluations described in this chapter.

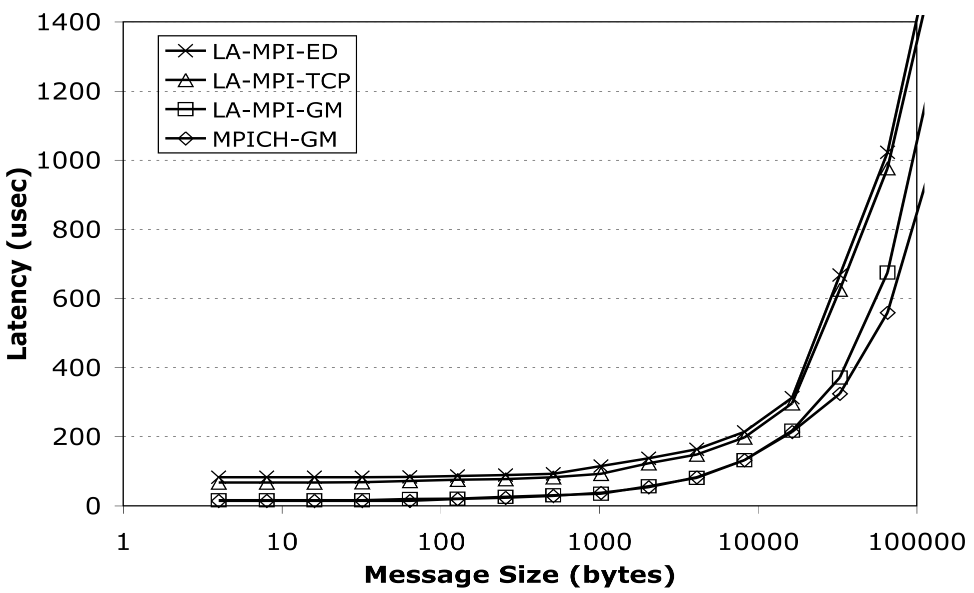

Figure 5.1 Comparison of message ping latencies for Myrinet and TCP over Ethernet messaging

5.1 MPI Communication Latency and Bandwidth

As mentioned in Section 2.6, MPI libraries are typically evaluated using simple latency and bandwidth microbenchmarks, as these are perceived to be the characteristics most indicative of application performance. The microbenchmarks used for the measurement of MPI library message ping latency and unidirectional message bandwidth are described in detail in Section 2.6.

Figure 5.1 shows message latency as a function of message size for LA-MPI and MPICH using both Myrinet and TCP over Ethernet as a communications medium. These plots show that both libraries have similar latencies when using Myrinet (LA-MPI-GM and MPICH-GM), especially for small message sizes. The difference between the two is about 2 μsec for 4 B–2 KB messages and beyond this increases to approximately 16 μsec for 64 KB messages. In comparison, LA-MPI using TCP, LA-MPI-TCP, consistently has a significantly higher latency; about 50 μsec higher than

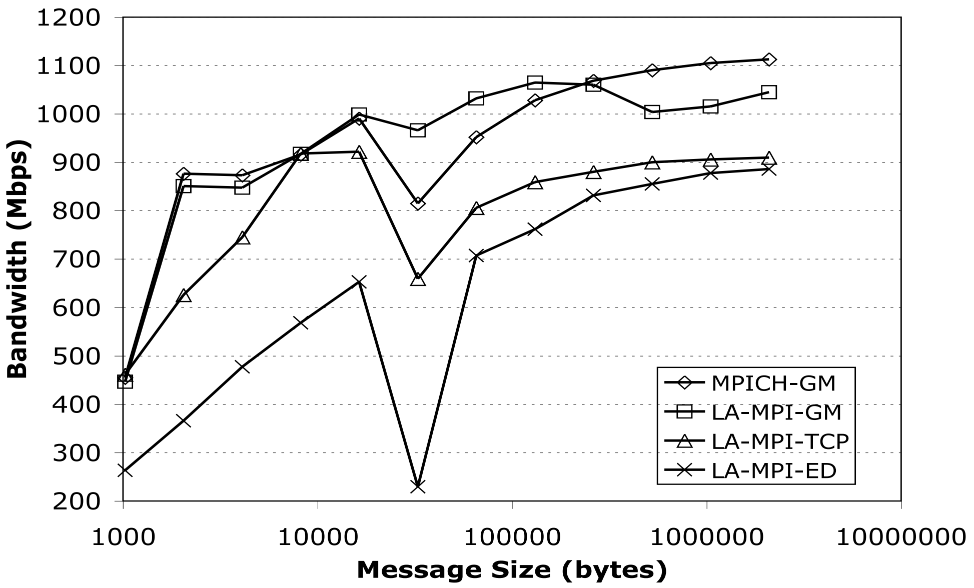

Figure 5.2 Comparison of unidirectional message bandwidths for Myrinet and TCP over Ethernet messaging

MPICH-GM for 4 B messages and increasing steadily to about 420 μsec for 64 KB messages. When TCP is used in an event-driven fashion, LA-MPI-ED, there is a further latency increase of 15 μsec above LA-MPI-TCP. This latency increase is the result of thread switching overhead between the main thread and the event thread within the library.

All of the library versions use the *eager* message transfer protocol for messages up to 16 KB, and then switch to the *rendezvous* protocol for larger messages. This results in as much as a two-fold increase in message latency for 32 KB messages compared to 16 KB messages, as shown in Figure 5.1. These message transfer protocols are explained in Table 2.1.

Figure 5.2 shows messaging bandwidth as a function of message size for LA-MPI and MPICH using both Myrinet and TCP over Ethernet as a communications medium. The figure shows that the achieved messaging bandwidth for 1 KB messages is low for all of the library versions. For such small messages, the per-message overhead of both the MPI library, and the communication substrate limit the achievable performance. As the message size increases, however, Myrinet enables the messaging bandwidth to increase faster than TCP over Ethernet. TCP messaging bandwidth saturates at around 930 Mbps, whereas the higher theoretical peak bandwidth of Myrinet enables it to achieve messaging bandwidths up to around 1100 Mbps. There is a sharp drop in messaging bandwidth in all cases as the message size increases from 16 KB to 32 KB, again because of the switch in the message transfer protocol. For messages larger than 32 KB, all library versions show a consistent increase in bandwidth with message size. MPICH-GM achieves the highest peak bandwidth of about 1100 Mbps, followed by LA-MPI-GM which peaks at about 1050 Mbps. The TCP versions of LA-MPI, however, achieve slightly lower maximum bandwidths of around 900 Mbps, with LA-MPI-ED about 20 Mbps lower than LA-MPI-TCP. LA-MPI-ED consistently sustains less bandwidth than LA-MPI-TCP because of thread switching overhead, and because the increased responsiveness of LA-MPI-ED leads to 100% unexpected receives on this simple benchmark (see Section 4.4.1). However, as the figure shows, the difference in performance between the two consistently decreases with increasing message size.

5.2 NAS Benchmark Performance

The simple microbenchmarks of the previous section show that Myrinet consistently enables lower latency and higher bandwidth MPI messaging. However, these microbenchmarks do not necessarily translate into overall application performance since they simply measure communication performance in isolation. In any MPI

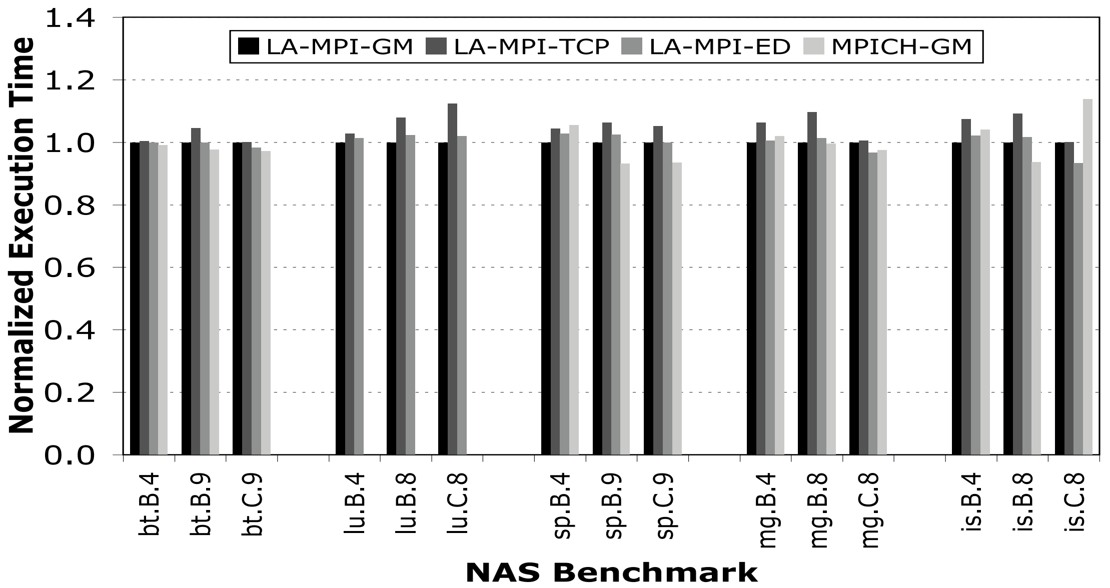

Figure 5.3 Execution times of NAS benchmarks with different MPI library versions normalized to LA-MPI-GM (bars shorter than 1 indicate a speedup for that particular benchmark over LA-MPI-GM).

application, communication occurs in parallel with computation, and most applications are written using non-blocking library calls to maximize the overlap between communication and computation.

The NAS parallel benchmarks (NPB), described in Section 2.6, are a more realistic set of benchmarks that include both computation and communication [1]. This section evaluates the various MPI library versions using five benchmarks from the NPB v2.2 suite—BT, SP, LU, IS, and MG. The experimental setup and the methodology used is exactly similar to the one followed in Section 4.4.2. The one difference in this case is that each workstation of the cluster is also equipped with a Myrinet LANai 7.2 network adapter, and is connected to a 16-port Myrinet switch.

Figure 5.3 compares the performance of different execution configurations of the five NAS benchmarks, achieved with LA-MPI-TCP, LA-MPI-ED, LA-MPI-GM, and

MPICH-GM. Each set of four bars of the graph corresponds to one particular configuration of the benchmark and is named accordingly—the first part of name gives the benchmark name, the middle part conveys the data-set size used for that particular experiment, and the last part conveys the number of participating MPI nodes for the experiment. Each bar of the graph depicts the normalized execution time of the corresponding benchmark using a particular library version relative to LA-MPI-GM. Thus, a bar of length less than 1.0 indicates a performance gain over the LA-MPI-GM library. Each data point for the graph is generated by averaging the execution times over 10 runs of each experiment.

Figure 5.3 shows that LA-MPI using TCP over Ethernet (LA-MPI-TCP) consistently performs worse than LA-MPI using Myrinet (LA-MPI-GM) on all of the 5 NAS benchmarks. Overall across the 15 benchmark configurations, LA-MPI-TCP is more than 5% slower than LA-MPI-GM. However, when TCP is used in a more efficient manner (LA-MPI-ED), it reduces the overall performance advantage of LA-MPI-GM across the 15 benchmark configurations to just over 0.3% (down from 5.2%). More interestingly however, LA-MPI-ED is able to beat LA-MPI-GM on several of the benchmarks, with a peak speedup of 7% on IS.C.8. Also, moving to a bigger cluster, or a larger data-set improves LA-MPI-ED with respect to LA-MPI-GM. Thus, as the communication component of a MPI application increases, a properly designed application is able to extract greater benefits from the increased concurrency of communication and computation, that the event-driven library provides. The two Myrinet MPI libraries—LA-MPI-GM and MPICH-GM—are almost evenly matched (on an average) in performance on this set of benchmarks. Some of the benchmark configurations show better performance with MPICH-GM, whereas others perform better with LA-MPI-GM. This shows that the library support for different interconnect technologies (or protocol) varies widely amongst different MPI library implementations, causing a substantial application performance impact. The LU benchmark results are not reported with MPICH-GM since the benchmark's internal verification tests failed with this library.

These results are especially interesting because they show that the LA-MPI-ED library using TCP messaging is able to outperform (or at least come very close to) both Myrinet library versions on several of the benchmarks, in spite of the significant latency and bandwidth advantage that Myrinet has over Gigabit Ethernet. LA-MPI-ED is able to outperform the other libraries because it is able to effectively overlap communication and computation better than the other library versions. Longer messages using non-blocking MPI communication provide greater scope for an effective overlap, and thus the applications which use more of these messages show performance improvements with LA-MPI-ED. However, when messaging latency effect predominates in the overall performance of the MPI application, the Myrinet MPI libraries perform better owing to their significantly lower message latencies.

5.3 Summary

This chapter shows that while specialized networks may have lower latency and higher bandwidth than commodity Ethernet when measured in isolation, this does not necessarily translate into better application performance. As the NAS benchmarks show, the library's ability to enable overlapping communication and computation is equally as important as raw latency and bandwidth. With comparable networking technologies, 1 Gbps Ethernet and 1.2 Gbps Myrinet, several (6 out of 15) of the NAS benchmarks run as fast or faster (by as much as 7%) when the MPI library efficiently uses TCP over Ethernet for communication rather than Myrinet. While the most recent Myrinet networks provide up to 2 Gbps of bandwidth, multiple Gigabit Ethernet links should provide competitive performance. Furthermore, the pending arrival of 10 Gbps Ethernet should extend the performance advantage of TCP over Ethernet.

Even if individual message latency is a factor, as in some of the NAS benchmarks used for this evaluation, TCP implementation of existing MPI libraries still has significant room for improvement. The latency gap between Ethernet and Myrinet is largely due to the memory copies required to move data between the application and the kernel, and the overhead of interrupt processing. The user-level communication protocols employed by specialized networks, including Myrinet and Quadrics, avoid these copies by directly transferring data between the network interface and application buffers, resulting in lower communication latencies. However, these same techniques can be integrated into commodity protocols like TCP, to make TCP messaging an inexpensive, portable, and reliable alternative to the specialized networks.

Chapter 6
Related Work

Chapter 4 presented a technique to handle message communication in a MPI library with an event-driven software model. This model, even though new in the MPI domain, has been extensively used to provide high performance in communication bound applications, such as network servers. A threaded software architecture is another mechanism extensively used by communication intensive applications, and is designed to exploit concurrency of processing requests in the application to provide high performance. The event-driven LA-MPI library, presented in Chapter 4 uses threading for a similar purpose—increasing concurrency between a MPI application's computation and communication tasks. Together with the event-driven model for handling TCP messaging in the library, the threaded LA-MPI library is shown to improve messaging performance over Ethernet quite substantially. This technique uses completely generic operating system features over commodity Ethernet to enable high-performance messaging. On the other hand, previous research has primarily focused on developing custom protocols and/or interconnect technologies to improve the performance of message-passing applications.

This chapter is organized as follows. Section 6.1 discusses some network server applications, and their use of the event-driven architecture model and the threaded architecture models to support high-performance communication using TCP over Ethernet. Section 6.2 discusses in brief the communication and message progression mechanisms found in other MPI libraries, both free and commercial ones. Section 6.3

presents a brief discussion on some of the other research efforts aimed at improving MPI library performance over Ethernet. Finally, Section 6.4 presents a high-level overview of some of the custom interconnect technologies that have been developed for high-performance message-passing.

6.1 Network Servers

6.1.1 Event-driven

The chief responsibility of a network server application is to accept and respond to user requests, which almost always takes the form of network communication. Thus, the server receives requests from client processes over the network, reads the request off the network, processes it, and sends back an appropriate response to the client. The communication handled by the network server is inherently asynchronous because it depends entirely on the client. An event-driven architecture model is an efficient means of handling asynchronous communication, and at the same time satisfy the high concurrency requirements that a large number of simultaneous client requests impose on a network server application. This has led to the development of a number of high-performance network servers based on this model.

thttpd (tiny/turbo/throttling HTTP server) is a small single-process, single-threaded web server built on the event-driven software architecture model [34]. It works exactly like the web server described in the example presented in Section 4.2. Thus, it implements the processing of HTTP requests as a state machine, where each step is triggered by the occurrence of an event, such as arrival of new data on the sockets being monitored by the server application, or the availability of space on the socket to send a response to the client, etc. The central event-loop of the thttpd web server monitors for these events, and invokes the appropriate event-handler on their occurrence. Since the number of events that the event-loop can simultaneously monitor is large, the web server can effectively interleave the processing of many HTTP requests. The entire server process executes in the context of a single process and a single thread of control, thereby avoiding the overheads of context switching and thread synchronization present in multi-process or multi-threaded web servers.

Flash is another web server developed on the event-driven architecture model, but it differs from thttpd because it is multi-threaded as well [32]. thttpd performs disk I/O operations as a part of the event-handlers for network events, and thus might block if a file I/O operation cannot be satisfied from the memory. This almost always causes a performance degradation of the web server. Flash performs disk I/O operations through the use of independent helper threads. Similar to thttpd, the main server thread continually waits for events, and dispatches the respective event-handlers on their occurrence. However, Flash also regards filesystem accesses as events, and handles these events by dispatching them to independent threads via IPC. These helper threads block on disk I/O requests, and return an event to the main server process on completion. This technique results in slightly higher overhead from multiple threads, but is much more tolerant of disk intensive workloads. The *Harvest* web cache has an architecture very similar to thttpd or Flash, but employs just a single thread since most of its workload can be kept in main memory, and thus file accesses are ensured not to block [8].

The event-driven approach to handle MPI communication, as introduced in Section 4.3, is based on exactly the same principles as these network servers. Since communication aspects of the MPI library are also inherently asynchronous, as in the network server, they are most efficiently handled by an event-loop, which monitors for these events, and dispatches appropriate event-handlers on their occurrence. In a MPI library all communication is limited to messages sent from one parallel process to another, and thus guaranteed to fit in the main memory of a node. Thus, blocking disk I/O operations are never performed by the event-handlers in LA-MPI-ED and consequently there is no performance degradation in that respect. However, LA-MPI-ED employs two threads of execution, unlike thttpd or harvest, since the computation tasks of a MPI application cannot be performed efficiently by an asynchronous event-driven software model. This necessitates the use of the second thread in the library to handle the synchronous computation requirements of the MPI application. At the same time, the use of this second thread leads to a certain amount of context switching and thread synchronization overhead in the LA-MPI-ED library.

6.1.2 Threaded

Threading offers another mechanism to exploit task concurrency, and handle asynchronous network I/O very efficiently in network servers. The requests to a network server are mostly independent of each other, and can be processed concurrently. Hence, network servers are well suited for a thread-per-request model, where the server spawns a thread to handle each request independently. Some examples of such applications include DCOM, RPC packages, and Java Remote Method Invocation [10, 26, 27]. The threading model of software architecture is also well supported by modern languages and programming environments. In this model too there is one central dispatcher loop, which waits for requests to arrive, and then spawns a new thread to process each request. The operating system (or the thread scheduler) overlaps computation and I/O operations, and enables the processing of multiple requests concurrently by transparently switching among threads.

The thread model, although simple to program, suffers from some overheads associated with cache and TLB misses, thread scheduling, and synchronization operations with locks and mutexes. The degradation of application performance because of these overheads can increase rapidly as the number of threads employed by the application grows. Thus, in applications where the number of requests is large and concurrency requirements quite high, the thread-per-request model has only had limited success. One technique adopted by web servers to get around this scalability problem of threading has been to use bounded thread pools. This provides a coarse form of load conditioning solution where the size of the thread pool is strictly bounded to ensure minimum throughput degradation of the network server with an increasing number of concurrent requests. This approach has been used by several popular web servers including Apache, IIS, and Netscape Enterprise Server [11, 17, 28]. Under this technique the web server processes only as many requests concurrently as there are threads in the pool, and rejects additional requests until some of the threads have completed their requests and been returned to the pool. With this technique, however, some requests can suffer arbitrarily high response times when the web server is operating under high load.

The thread model employed by LA-MPI-ED differs significantly from the conventional threaded architecture for a network server, especially in not following the thread-per-request model. Thus, the library never spawns threads dynamically during the execution of the MPI application—the event thread gets spawned by the main thread during library initialization, and both threads run concurrently until library termination (see Section 4.3.1). Thus, the threading overhead associated with LA-MPI-ED is much reduced as compared to that in typical threaded network servers. Also, this overhead is almost constant since threads are not spawned dynamically, and thus unlike network servers, LA-MPI-ED does not suffer latency degradation with an increase in the number of concurrent message requests.

6.2 Messaging in MPI Libraries

Communication performance is a big determinant of the overall performance of any MPI library. It is dependent both on the interconnect technology used, and the level of support in the library for that technology. However, the model of communication employed by the library is just as crucial for performance, and is especially important in understanding the performance differences between various libraries on any particular interconnect. The model of communication followed by a MPI library dictates how the library handles progression of pending requests in the library. Thus, it exerts considerable influence on the overall performance of the library too.

The most common approach across a range MPI libraries, both publicly available and proprietary, is to use a synchronous model for communication, tying it with the computation aspects of the library. This model was explained in Chapter 3 in the context of the LA-MPI library. Two other commonly used, freely available MPI libraries—MPICH and LAM/MPI—both employ some form of a synchronous progress engine to handle progression of pending requests [6, 20, 38] (Section 2.3). Both of these libraries support TCP messaging over Ethernet. However, none of them utilize the efficient event mechanisms made available by the operating system to handle TCP communication, and thus, are not particularly efficient over Ethernet. The progress mechanism for TCP in both of these libraries uses a `select`-based polling approach similar to the technique employed by the LA-MPI library (Section 3.1). This necessarily ties the progression of requests in the library to the invocation of the library by the MPI application, and limits the amount of concurrency that can be exploited between computation and communication. Most significantly, they place the burden of exploiting this concurrency solely on the application programmer. Even though these libraries incorporate the notion of events (by using `select`) to detect incoming messages over a TCP connection, they are not event-driven, and thus, are not efficient in their communication over Ethernet using TCP. In contrast, the LA-MPI-ED library enables communication and computation to proceed concurrently in a MPI application by the use of independent threads to perform these two tasks. Moreover, the event-driven communication employed by LA-MPI-ED to handle TCP messaging makes the library more responsive to network events, and reduces the latency of non-blocking messages significantly.

None of the research efforts on MPI communication using TCP utilizes the well-known techniques in the network server domain of event-driven software architecture and threading. To the best of my knowledge, all free MPI libraries use similar synchronous progress engines to handle non-blocking communication. Thus, they fail to exploit the concurrency between communication and computation efficiently. More significantly, they also fail to implement the MPI standard correctly (see Section 4.1). However, there are some commercial MPI libraries, which claim to provide a correct (and MPI standard compliant) asynchronous message progression mechanism, and thus, enable greater concurrency between the communication and computation tasks of a MPI application.

Three of these commercial libraries are the MPI/PRO, ChaMPIon/PRO, and HP MPI v2.0 library. The MPI/PRO and ChaMPIon/PRO are developed by MPI Software Technology, Inc., and enable message progression irrespective of whether the application makes MPI library function calls, or is executing in some computationally intensive section of code [24, 25]. This enables these libraries to overlap communication and computation on platforms that support such asynchronous I/O, and reduce CPU overhead. These two libraries, however, do not advertise how they achieve asynchronous message progression and I am not aware of how they do it.

The third commercial MPI library, HP MPI v2.0, does mention the technique it uses to enable independent message progression. This library uses the `SIGALRM` signal to timeout at fixed intervals, and progresses pending messages at those instances [31]. The timeout period can be modified by the user to match hardware and application requirements, and in this way, the library aims to achieve reasonable concurrency in performing computation and communication.

Even though this technique is effective for achieving independent message progress in a MPI library, its not completely analogous to the event-driven communication thread employed by the LA-MPI-ED library (Section 4.3). Most importantly, the HP MPI v2.0 library is not event-driven. The alarm timeouts only partially overcome the drawbacks of a synchronous progress engine that were discussed in Section 4.1. Even though it enables the progression of pending requests in the library independently of library invocations by the user application, the progression does not occur on network events. Thus, when these timeouts do occur, there is only a probabilistic chance of it coinciding with a network event to progress pending requests suitably. An event-driven MPI library, on the other hand, ensures that request progression is performed immediately when a network event occurs. This enables efficient TCP communication in the library, and is as such more effective in enabling asynchronous MPI request progression than the HP MPI library.

6.3 High-performance MPI Research

A subset of research into high-performance MPI libraries has proposed the development of custom communication protocols for messaging over Ethernet, instead of using generic protocols provided as a part of the operating system protocol stack. Shivam et al. proposed a message layer called Ethernet Message Passing for enabling high-performance MPI [36]. EMP implements custom messaging protocols directly in the network interface, and bypasses the entire protocol stack of the operating system. EMP supports reliable messaging and provides zero-copy I/O to user MPI applications for pre-posted receives. Experimental results show that EMP provides significantly lower message latency than TCP over Ethernet. In a later work, Shivam et al. also extended EMP to parallelize the receive processing to take advantage of a multiple CPU network interface and, showed further gains in MPI performance [37].

TCP splintering is another research work aimed at improving TCP MPI performance over Ethernet [18]. This work focuses on the limitations of TCP's congestion-control and flow-control policies in a cluster environment, and proposes offloading parts of the protocol stack to the network interface to provide improved MPI performance. Specifically, it targets congestion control and acknowledgment generation for offloading to the network interface in order to reduce the CPU overhead of message-passing, and improve message latency.

Both EMP and TCP splintering argue against the existing TCP protocol over Ethernet as a messaging medium, and suggest custom messaging protocols or custom alterations to the TCP protocol stack. However, these custom modifications have limited portability across different hardware systems and/or operating systems, and thus, are quite restricted in their applicability. Furthermore, custom messaging protocols are difficult to implement correctly, and even more difficult to debug. On the other hand, the enhancements to MPI libraries proposed in Chapter 4 rely purely on MPI library-level techniques to improve messaging efficiency using existing communication protocols, such as TCP. Since they do not alter the TCP/IP protocol stack of the operating system, or make any implicit assumptions about the hardware platform, these enhancements can be applicable over a range of different message-passing environments. Even though currently these enhancements only benefit TCP messaging over Ethernet, both TCP/IP support in operating systems, and Ethernet are ubiquitous, making the enhancements truly generic in their scope and widely applicable.

TCP's congestion-control and flow-control policies increase the operating system overhead of messaging using TCP, and increase message latency. However, reliability of communication is essential even for message-passing applications. As Chapter 3 showed, it is often better to use the highly tuned reliability mechanisms of the TCP protocol, rather than provide these mechanisms at the library-level. Also, offloading

TCP processing or significant parts of it to the network interface is orthogonal to the MPI library enhancements proposed in this thesis, and can only serve to improve TCP messaging performance further.

6.4 Specialized Interconnects for Message-Passing

A more common approach to high-performance MPI is to use specialized interconnect technologies, such as Myrinet or Quadrics [4, 33]. Myrinet is a high-performance, packet-communication, and switching technology that has gained considerable acceptance as a cluster interconnect [4]. It uses Myricom LANai programmable network interfaces, and user-level communication protocols to provide a low-latency high-bandwidth communication medium. Similarly, the Quadrics interconnect technology uses programmable network interfaces called Elan, along with special-purpose network switches and links to provide a high-performance messaging substrate on clusters [33]. The Quadrics network achieves high-performance by creating a global virtual-address space through hardware support to encompass the individual nodes' address spaces. User-level communication protocols and messaging libraries are implemented on the network interface itself, and enable the Quadrics network to provide even higher bandwidths and lower message latencies than Myrinet.

While these specialized networks once outperformed Ethernet, with the advent of 1–10 Gbps Ethernet, it is no longer clear that there is a significant performance benefit in network speed [23]. These specialized user-level networks offer other advantages towards high-performance messaging by reducing copies in data transmission, avoiding interrupts, and offloading message protocol processing onto a network interface. However, many of these same techniques can be integrated into commodity networks using TCP and Ethernet. For example, recent work looks to offload all or part of TCP processing onto an Ethernet network interface, reducing acknowledgment-related interrupts as a side-effect [3, 22]. The use of zero-copy sockets can improve TCP performance further by avoiding extraneous copies for message sends and posted message receives [9, 15, 35]. Copy avoidance for unexpected receives could be provided through network interface support, while pre-posting library buffers to the operating system in conjunction with a system such as Fast Sockets would allow the reduction of one extra copy from the socket buffer to the library buffers [35]. When used together with the event-driven communication model described in Chapter 4, copy avoidance, interrupt avoidance, and TCP offloading could significantly improve messaging performance with TCP on Ethernet, bringing it closer to the specialized technologies.

Chapter 7
Conclusions

7.1 Conclusions

MPI libraries provide a rich and powerful set of APIs for parallel programming on a cluster environment using the message-passing paradigm. These libraries have to perform essentially two tasks on behalf of the message-passing applications—computation and communication. Thus, communication performance of a MPI library is a big factor influencing the overall performance of a MPI application. The traditional approach towards high-performance MPI has been through the use of custom protocols and/or interconnect technologies, such as Myrinet and Quadrics [4, 33]. These technologies use user-level communication protocols, and powerful programmable network interfaces to provide a low latency, high bandwidth communication substrate for messaging. On the other hand, commodity protocols, such as TCP/UDP on Ethernet, use the protocol stack of the operating system for communication, and are generally considered unfit for high-performance MPI applications owing to their high communication overhead and low performance. A part of the reason for the messaging performance gap between Ethernet and the specialized networks is the inefficiency of existing support in MPI libraries for Ethernet messaging. With the use of efficient communication mechanisms, which utilize the full potential of the operating system support for TCP communication on Ethernet, this performance gap can be reduced, and Ethernet can be a viable alternative to the specialized networks.

A common perception in the MPI community is that TCP as a messaging layer performs worse than UDP because of the substantial operating system overhead associated with TCP communication. This has prompted several MPI library developers to offer messaging support over Ethernet with UDP datagrams. In these libraries, the reliability of communication is provided by library-level reliability and flow-control mechanisms. Chapter 3 shows that these mechanisms add greater overhead for messaging than using a TCP messaging layer, leading to poorer performance. The chapter proposes the design of an efficient TCP messaging layer for the LA-MPI library. The results show that the message bandwidth capability of the LA-MPI library almost doubles with the TCP messaging layer as compared to the UDP messaging layer. With the TCP messaging layer, the LA-MPI library can attain line-rate message bandwidth for as small as 8 KB messages, whereas, with the UDP messaging layer, the message bandwidth of the library never increases beyond 500 Kbps for message sizes greater than 1 KB. Message latency also shows a consistent improvement with the TCP messaging layer, with improvements ranging from 3–29% across a range of message sizes. These factors also lead to an improvement in the overall MPI application performance with TCP messaging. Thus, TCP, even with its higher operating system overhead as compared to UDP, is a better messaging alternative than the latter.

Chapter 4 then proposes a technique to separate the concerns of computation and communication in a MPI library, and apply the best approach for each of them. Through this technique the computation component of a MPI application is performed synchronously through the functional interface of the library, and the communication portion is handled by an independent event-driven thread in the library. This enables communication in the library to occur in response to asynchronous network events rather than being tied to synchronous library calls. This increases communication responsiveness of the library, and improves the performance of non-blocking MPI communication. The event-driven approach to handle communication in the MPI library is currently provided for the TCP messaging layer by utilizing the network events provided by the operating system. The results with this technique show significant improvement in the performance of non-blocking MPI communication primitives, sometimes even eliminating wait-times of non-blocking messages completely. Overall MPI application performance also improves with this technique, depending upon the design of the application itself, and how much of an overlap of communication and computation it is designed to exploit. Among 5 benchmarks from the NAS parallel benchmark suite, the event-driven LA-MPI library shows an average performance improvement of 4.5% over LA-MPI-TCP, with a peak of 6.2% (for the IS benchmark). The performance of the event-driven LA-MPI library relative to LA-MPI-TCP also improves as the number of nodes employed for the NAS benchmark experiments is increased.

Chapter 5 provides a comparison of TCP messaging over Ethernet and Myrinet messaging. 1.2 Gbps Myrinet has a significantly lower message latency and higher message bandwidth than TCP over 1 Gbps Ethernet. However, these attributes of the Myrinet network do not necessarily translate into considerable application performance advantage as compared to Ethernet messaging. The results from the execution of the NAS benchmarks show that the library's ability to enable overlapping communication and computation is as important as raw latency and bandwidth. Thus, up to

6 of the 15 benchmark configurations run as fast as or faster than Myrinet when the MPI library efficiently uses TCP over Ethernet for communication. Also, when the MPI library efficiently uses TCP over Ethernet for communication, the performance gap between Ethernet messaging and Myrinet messaging is brought down from 5% to just over 0.3%.

This thesis shows that TCP communication over Ethernet can be handled efficiently in MPI libraries to provide a performance comparable to that with more expensive specialized interconnect technologies. Thus, Ethernet can be a low-cost, portable, and reliable communication substrate for high-performance MPI.

7.2 Future Work

Message latency is an important consideration for MPI applications, especially those that are not written to fully exploit the non-blocking MPI communication primitives. These applications perform worse on Ethernet than on Myrinet or Quadrics owing to the higher message latency of Ethernet. However, the TCP protocol implementation in the operating system has significant room for improvement. The latency gap between Ethernet and the specialized networks is largely due to the memory copies required to move data between the application and the kernel, and the overhead of interrupt processing. The user-level communication protocols employed by the specialized networks avoid these copies by directly transferring data between network interface and application buffer. They also avoid the protocol processing and interrupt overheads by implementing a significant portion of the communication protocol on their programmable network interface. However, these same techniques can also be integrated into commodity protocols, such as TCP. Extraneous copies for message sends and posted receives can be eliminated by using various zero-copy socket alternatives [35, 15]. The CPU overhead of protocol processing and interrupt processing can also be reduced by offloading all or part of TCP processing onto an Ethernet network interface [22]. These TCP optimizations would enable even higher MPI performance with TCP messaging over Ethernet. Moreover, these TCP enhancements would simultaneously benefit all networking applications that rely on TCP for communication.

The performance of TCP messaging over Ethernet can be further improved by maximizing the bandwidth utilization of Gigabit Ethernet. Section 4.4 shows that the *rendezvous* protocol is responsible for the sharp drop in unidirectional message bandwidth as message size increases beyond 16 KB. As already mentioned, the *rendezvous* protocol is not necessary in TCP messaging for flow-control, since TCP already provides that for unread data in the socket buffers. However, adherence to this protocol is necessary to enable a receiver to control the order in which concurrent messages from the same sender are processed; the TCP protocol does not permit this. In fact, it is the socket interface to the TCP protocol stack, which lacks an appropriate mechanism to reorder concurrent messages from the same sender. The existing socket interface forces data to be read sequentially from the head of its buffer. An enhancement to this interface that allows data to read from arbitrary positions in the socket buffer can eliminate the need for the *rendezvous* protocol entirely. Such a scheme could enable reading data from the middle of a socket buffer, without being required to copy all earlier data from the buffer. With such a mechanism, TCP flow control would suffice for even reordering different messages from the same sender. It would also eliminate the extra memory copy required for most unexpected receives. Only when the amount of incoming data in the socket buffer exceeds a certain high-water mark, messages would need to be copied out of it, and into a temporary library buffer. Thus, such a technique could significantly improve Ethernet bandwidth utilization of the MPI library.

The existing implementation of the event-driven LA-MPI library adds a thread context switch on the message path on at least the receiver. This results in some additional message latency, as seen in the results presented in Section 4.4.1. Even though this overhead is effectively hidden for asynchronous MPI communication, the performance of blocking MPI primitives is adversely affected. The use of lightweight threads keeps the thread switching time to a minimum, but it would be helpful to eliminate this overhead altogether, or at least reduce it further. One possible way of achieving this could be to utilize processors that enable multiple thread contexts for hyper-threading or simultaneous multithreading [13, 29]. Although these systems see some inter-thread communication overheads, those should be low as the communication is only of metadata in shared memory regions that may be cached. A comprehensive evaluation of these architectures for the threaded LA-MPI-ED library would enable identifying and reducing the elements responsible for inter-thread communication in the library.

References

1. D. H. Bailey, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinki, R. S. Schreiber, H. D. Simon, V. Venkatakrishnam, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

2. D. H. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. NASA Technical Report NAS-95-020, NASA Ames Research Center, December 1995. http://www.nas.nasa.gov/Software/NPB/.

3. Hrvoje Bilic, Yitzhak Birk, Igor Chirashnya, and Zorik Machulsky. Deferred Segmentation for Wire-Speed Transmission of Large TCP Frames over Standard GbE Networks. In *Hot Interconnects IX*, pages 81–85, August 2001.

4. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE MICRO*, 15(1):29–36, February 1995.

5. C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest Information Discovery and Access System. *Computer Networks and ISDN Systems*, 28(1–2):119–125, 1995.

6. Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

7. Ralph M. Butler and Ewing L. Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System. *Parallel Computing*, April 1994.

8. A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrel. A Heirarchial Internet Object Cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 153–163, Jan 1996.

9. Jeffery S. Chase, Andrew J. Gallatin, and Kenneth G. Yocum. End-system Optimizations for High-Speed TCP. *IEEE Communications, Special Issue on High-Speed TCP*, 39(4):68–74, April 2001.

10. Microsoft Corporation. *DCOM Technical Overview*, 1996.

11. Netscape Corporation. Netscape Enterprise Server. http://wp.netscape.com/enterprise/v3.6/index.html.

12. Jack Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Computer science technical report number cs - 89 - 85, University of Tennessee, Knoxville, July 2004.

13. Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stam, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5), 1997.

14. The Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

15. Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *Proceedings of 1999 USENIX Technical Conference*, pages 109–120, June 1999.

16. Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and Evaluating `epoll`, `select`, and `poll` Event Mechanisms. In *Proceedings of the Ottawa Linux Symposium*, July 2004.

17. Dean Gaudet. Apache Performance Notes. http://httpd.apache.org/docs/misc/perf-tuning.html.

18. Patricia Gilfeather and Arthus B. Macabe. Making TCP Viable as a High Performance Computing Protocol. In *Proceedings of the Third LACSI Symposium*, Oct 2002.

19. Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A Network-Failure-Tolerant Message-Passing System for Terascacle Clusters. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing*, June 2002.

20. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

21. William D. Gropp and Barry Smith. Chameleon Parallel Programming Tools Users Manual. Techinical Report ANL-93/23, Argonne National Laboratory, March 1993.

22. Yatin Hoskote, Bradley A. Bloechel, Gregory E. Dermer, Vasantha Erraguntla, David Finan, Jason Howard, Dan Klowden, Siva G. Narendra, Greg Ruhl, James W. Tschanz, Sriram Vangal, Venkat Veeramachaneni, Howard Wilson, Jianping Xu, and Nitin Borkar. A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 38(11):1866–1875, November 2003.

23. Justin Gus Hurwitz and Wu chun Feng. End-to-end Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro*, 24(1):10–22, 2004.

24. MPI Software Technology Inc. ChaMPIon/Pro. ChaMPIon/Pro product sheet, April 2003.

25. MPI Software Technology Inc. MPI/Pro for Linux: Standards-based Message Passing Middleware for Clusters and Multiprocessors. MPI/Pro product sheet, April 2003.

26. Sun Microsystems Inc. RPC: Remote Procedure Call Protocol Specification Version 2. Network Working Group RFC1057, June 1988.

27. Sun Microsystems Inc. Java Remote Method Invocation Specification. Revision 1.7, Java 2 SDK, Standard Edition v13.0, December 1999.

28. Bill Karagounis. Web and Application Server Infrastructure - Performance and Scalability. Microsoft Corporation, April 2003.

29. Kevin Krewell. Intel's Hyper-Threading Takes Off. *Microprocessor Report*, 20(4):547–564, 2002.

30. Jonathan Lemon. Kqueue: A Generic and Scalable Event Notification Facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

31. Hewlett-Packard Development Company L.P. HP MPI User's Guide, December 2003. Eighth Edition.

32. Vivek S. Pai, Peter Druschel, and Willy Zwaenpoel. FLASH: An Efficient and Portable Web Server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

33. Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The QUADRICS Network: High-Performance Clustering Technology. *IEEE MICRO*, Jan 2002.

34. Jef Poskanzer. *thttpd - tiny/turbo/throttling HTTP server*. Acme Laboratories, February 2000. http://www.acme.com/software/thttpd.

35. Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-Performance Local Area Communication With Fast Sockets. In *Proceedings of the 1997 USENIX Technical Conference*, pages 257–274, January 1997.

36. Piyush Shivam, Pete Wyckoff, and Dhabaleshwar Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Proceedings of SC2001*, Nov 2001.

37. Piyush Shivam, Pete Wyckoff, and Dhabaleshwar Panda. Can User Level Protocols Take Advantage of Multi-CPU NICs? In *Proceedings of IPDPS2002*, Apr 2002.

38. Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

39. W. Richard Stevens. *TCP/IP Illustrated, Volume 1. The Protocols*. Addison-Wesley Professional, January 1994.