Message Passing Using Raw Ethernet Frames In Parallel Computing

A Project Report

submitted in partial fulfillment of the requirements
for the award of the degree of

Bachelor of Technology

in

AEROSPACE ENGINEERING

by

Deepak Sarda

under the guidance of

Dr. P. Sriram

DEPARTMENT OF AEROSPACE ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.
June 2003

Certificate

This is to certify that the project entitled **Message Passing Using Raw Ethernet Frames in Parallel Computing** submitted by **Deepak Sarda** in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Aerospace Engineering, is a bona fide record of work carried out by him under the supervision and guidance of **Dr. P. Sriram** during the academic year 2002-2003 at the Department of Aerospace Engineering, Indian Institute of Technology Madras.

Dr. S. Santhakumar
Professor and Head
Dept. of Aerospace Engineering
IIT Madras

Dr. P. Sriram
Associate Professor
Dept. of Aerospace Engineering
IIT Madras

Acknowledgements

This work would not have been possible without the help, guidance and support of several people, Prof. Sriram being the foremost among them. His encouragement and advice saw me through several difficult periods. His cheerful demeanour and never-say-die attitude are infectious; things I'll always remember him for.

My heartfelt thanks go to Neville Clemens whose short *funda* sessions saved me several hours on more than one occasion and helped finish this project in time. To all my other sixth-wing mates and some non-sixth-wingers: it's been wonderful knowing each one of you. My four years at IIT Madras wouldn't have been so great without you all.

I cannot even begin to thank my parents who have always stood by me, supported me through all my endeavours and made me into the person I am today.

Deepak Sarda

Abstract

The recent past has witnessed a great success in building inexpensive high performance computing platforms. This has been achieved by designing a parallel computing platform using commodity off-the-shelf components and free, open-source software. Such systems, termed Beowulf clusters, have now made high performance computing affordable for many people.

A Message passing system is the programming model of choice for such clusters. To derive the maximum performance from such a system, it becomes imperative that the inter-process communication infrastructure works at the maximum efficiency possible. The present work looks at communication methods currently in used and their shortcomings in the context of cluster computing. An alternative communication framework is presented and preliminary results of performance measurements against existing methods are provided. The results obtained indicate promise in developing the proposed framework further.

Table of Contents

List of Figures

CHAPTER 1

Introduction

1.1 High Performance Computing

High performance computing has been gaining importance in recent times. We start by introducing different classes of these high performance machines. This classification is based on the method of manipulating instructions and data streams and comprises four main architectural classes.

SISD machines SISD stands for Single Instruction Single Data. These are the conventional systems that contain one CPU and hence can accommodate one instruction stream that is executed serially. Nowadays many large mainframes may have more than one CPU but each of these execute instruction streams that are unrelated. Therefore, such systems still should be regarded as (a couple of) SISD machines acting on different data spaces. Examples of SISD machines are for instance most workstations like those of DEC, Hewlett-Packard, and Sun Microsystems.

SIMD machines SIMD stands for Single Instruction Multiple Data. Such systems often have a large number of processing units, ranging from 1,024 to 16,384, designed such that all may execute the same instruction on different data in lock-step. So, a single instruction manipulates many data items in parallel. Examples of SIMD machines in this class are the CPP DAP Gamma II and the Alenia Quadrics.

MISD machines MISD stands for Multiple Instruction Single Data. Theoretically, in these machines multiple instructions should act on a single stream of data. As yet no practical machine in this class has been constructed nor are such systems easy to conceive.

MIMD machines MIMD stands for Multiple Instruction Multiple Data. These machines execute several instruction streams in parallel on different data. The difference between the multi-processor SISD machines mentioned above and this kind lies in the fact that the instructions and data are related because they represent different parts

of the same task to be executed. MIMD systems may run many sub-tasks in parallel in order to shorten the time-to-solution for the main task to be executed.

There is another important distinction between the classes of systems :

Shared memory systems Shared memory systems have multiple CPUs all of which share the same address space. This means that the knowledge of where data is stored is of no concern to the user as there is only one memory accessed by all CPUs on an equal basis. Shared memory systems can be both SIMD or MIMD.

Distributed memory systems In this case each CPU has its own associated memory. The CPUs are connected by some network and may exchange data between their respective memories when required. In contrast to shared memory machines, the user must be aware of the location of the data in the local memories and will have to move or distribute data explicitly when needed. Again, distributed memory systems may be either SIMD or MIMD. The first class of SIMD systems mentioned which operate in lock step, all have distributed memories associated to the processors. As we will see, distributed-memory MIMD systems exhibit a large variety in the topology of their connecting network. The details of this topology are largely hidden from the user which is quite helpful with respect to portability of applications.

Although the difference between shared and distributed memory machines seems clear cut, this is not always entirely the case from user's point of view. Virtual shared memory can also be simulated at the programming level: A specification of High Performance Fortran (HPF) was published in 1993 [1] which, by means of compiler directives, distributes the data over the available processors. Therefore, the system on which HPF is implemented in this case will look like a shared memory machine to the user. Other vendors of Massively Parallel Processing systems (sometimes called MPP systems), like HP and SGI/Cray, also are able to support proprietary virtual shared-memory programming models due to the fact that these physically distributed memory systems are able to address the whole collective address space. For the user such systems have one global address space spanning all of the memory in the system. In addition, packages like TreadMarks [2] provide a virtual shared memory environment for networks of workstations(NOWs).

Another trend that has came up in the last few years is distributed processing. This takes the Distributed Memory - MIMD concept one step further. Instead of housing many integrated processors in one or several boxes, several independent workstations, mainframes, etc., are connected by (Gigabit) Ethernet, FDDI or other high speed networks and set to work concurrently on tasks in the same program. Conceptually, this is not different

from Distributed Memory - MIMD computing, but the communication between processors is often orders of magnitude slower. Many packages to realise distributed computing are available. Examples of these are PVM (Parallel Virtual Machine), and MPI (Message Passing Interface). This style of programming, called the "message passing" model has become so widely accepted that PVM and MPI have been adopted by virtually all major vendors of distributed-memory MIMD systems and even on shared-memory MIMD systems for compatibility reasons. In addition there is a tendency to cluster shared-memory systems, for instance by HiPPI channels, to obtain systems with a very high computational power. Example., the NEC SX-5, and the SGI/Cray SV1 have this structure. So, within the clustered nodes a shared-memory programming style can be used while between clusters message-passing could be used.

1.2 The Beowulf Class of Cluster Computers

In the summer of 1994 Thomas Sterling and Don Becker, working at CESDIS under the sponsorship of the ESS project, built a cluster computer consisting of 16 DX4 processors connected by channel bonded Ethernet. They called their machine Beowulf. The machine was an instant success and their idea of providing COTS (Commodity off the shelf) base systems to satisfy specific computational requirements quickly spread through NASA and into the academic and research communities. The development effort for this first machine quickly grew into a what we now call the Beowulf Project. Some of the major accomplishment of the Beowulf Project will be chronicled below, but a non-technical measure of success is the observation that researchers within the High Performance Computer community are now referring to such machines as "Beowulf Class Cluster Computers." That is, Beowulf clusters are now recognized as genre within the High Performance Computing community.

The exact configuration of a balanced cluster will continue to change and will remain dependent on the size of the cluster and the relationship between processor speed and network bandwidth and the current prices of each of the components. An important characteristic of Beowulf clusters is that changes in the configuration of the cluster do not change the programming model. Therefore, users of these systems can expect to enjoy more forward compatibility then we have experienced in the past.

Another key component to forward compatibility is the system software used on Beowulf. With the maturity and robustness of Linux, GNU software and the "standardization" of message passing via PVM and MPI, programmers now have a guarantee that the programs they write will run on future Beowulf clusters - regardless of who makes the proces-

sors or the networks. A natural consequence of coupling the system software with vendor hardware is that the system software must be developed and refined only slightly ahead of the application software. The historical criticism that system software for high performance computers is always inadequate is actually unfair to those developing it. In most cases coupling vendor software and hardware forces the system software to be perpetually immature. The model used for Beowulf system software can break that rule.

In the taxonomy of parallel computers, Beowulf clusters fall somewhere between MPP (Massively Parallel Processors, like the nCube, CM5, Convex SPP, Cray T3D, Cray T3E, etc.) and NOWs (Networks of Workstations). The Beowulf project benefits from developments in both these classes of architecture. MPPs are typically larger and have a lower latency interconnect network than an Beowulf cluster. Programmers are still required to worry about locality, load balancing, granularity, and communication overheads in order to obtain the best performance. Even on shared memory machines, many programmers develop their programs in a message passing style. Programs that do not require fine-grain computation and communication can usually be ported and run effectively on Beowulf clusters. Programming a NOW is usually an attempt to harvest unused cycles on an already installed base of workstations in a lab or on a campus. Programming in this environment requires algorithms that are extremely tolerant of load balancing problems and large communication latency. Any program that runs on a NOW will run at least as well on a cluster.

A Beowulf class cluster computer is distinguished from a Network of Workstations by several subtle but significant characteristics. First, the nodes in the cluster are dedicated to the cluster. This helps ease load balancing problems, because the performance of individual nodes are not subject to external factors. Also, since the interconnection network is isolated from the external network, the network load is determined only by the application being run on the cluster. This eases the problems associated with unpredictable latency in NOWs. All the nodes in the cluster are within the administrative jurisdiction of the cluster. For example, the interconnection network for the cluster is not visible from the outside world so the only authentication needed between processors is for system integrity. On a NOW, one must be concerned about network security. Another example is the Beowulf software that provides a global process ID. This enables a mechanism for a process on one node to send signals to a process on another node of the system, all within the user domain. This is not allowed on a NOW. Finally, operating system parameters can be tuned to improve performance. For example, a workstation should be tuned to provide the best interactive feel (instantaneous responses, short buffers, etc.), but in cluster the nodes can be tuned to provide better throughput for coarser-grain jobs because they are not interacting directly with users.

The future of the Beowulf project will be determined collectively by the individual organizations contributing to the Beowulf project and by the future of mass-market COTS. As microprocessor technology continues to evolve and higher speed networks become cost effective and as more application developers move to parallel platforms, the Beowulf project will evolve to fill its niche.

An unabridged version of the above account of Beowulf history is available online at: http://www.beowulf.org/beowulf/history.html

CHAPTER 2

Parallel Programming

Exploiting the full potential of a parallel computer requires a cooperative effort between the user and the language system. There is clearly a trade-off between the amount of information the user has to provide and the amount of effort the compiler has to expend to generate optimal parallel code. At one end of the spectrum are languages where the user has full control and has to explicitly provide all the details while the compiler effort is minimal. This approach, called *explicit parallel programming*, requires a parallel algorithm to explicitly specify how the processors will cooperate in order to solve a particular problem. At the other end of the spectrum are sequential languages where the compiler has full responsibility for extracting the parallelism in the program. This approach, called *implicit parallel programming*, is easier for the user because it places the burden of parallelization on the compiler. Clearly, there are advantages and disadvantages to both, explicit and implicit, parallel programming approaches. Many current parallel programming languages for parallel computers are essentially sequential languages augmented by a set of special system calls (primitives). A parallel program on these computers is a collection of cooperating *processes* which execute concurrently, sychronizing and sharing data among the processors making use of these primitives. The lack of standards in these programming languages makes parallel programmes difficult to port to different parallel computers. Parallel programming libraries address this issue by offering vendor-independent primitives.

There are, in general, three (explicit) parallel programming models:

- Message Passing Programming
- Shared Memory Programming
- Data Parallel Programming

The first of these is the model of choice for developing programs for any Beowulf class machine.

2.1 Message Passing Programming

In message passing programming, programmers view their programs as a collection of cooperating processes with private variables. The only way for an application to share data among processes is for the programmer to explicitly code commands to move data from one process to another. The message passing programming style is naturally suited to Beowulf class machines in which some memory is local to each processor but none is globally accessible.

Message passing programming implementations need only two primitives in addition to normal sequential language primitives. These are *Send* and *Receive* primitives. These primitives are used to exchange data/instructions among processes. Both these primitives come in two variants - the *blocking* and the *non-blocking* calls. In a blocked send call, after a message is sent, program execution is suspended till the destination process receives the message; execution is resumed thereafter. This is also called *synchronous* send. In the non-blocking version of send, program execution continues after the message has been sent, irrespective of whether it has been delivered to its destination. This is also called *asynchronous* send.

There are two public domain message passing systems which are used widely - *Message Passing Interface (MPI)* and *Parallel Virtual Machine (PVM)*. MPI is now a well accepted standard and is widely used. Although not a formal standard, PVM is also quite popular. For the present work, we chose the MPI approach.

2.2 Message Passing Interface

MPI, developed by the *MPI Forum* [3], is a standard specification for a library of message passing functions. MPI specifies a public-domain, platform independent standard [4] of a message library, thereby achieving portability. This specification is OS, platform and vendor neutral but encourages development of optimized implementations of the standard for various platforms.

MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI includes *point-to-point* message passing and *collective* (global) operations, all scoped to a user-specified group of processes. Furthermore, MPI provides abstractions for processes at two levels. First, processes are named according to the rank of the group in which the communication is being performed. Second, virtual topologies allow for graph or Cartesian naming of processes that help relate the application semantics to the message passing

semantics in a convenient, efficient way. Communicators, which house groups and communication context (scoping) information, provide an important measure of safety that is necessary and useful for building up library-oriented parallel code.

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

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

Several open source implementations [5] of the MPI library have been developed. Some of them are:

MPICH The Argonne National Laboratory/Mississippi State University open source implementation. [6]

LAM/MPI Originally from the Ohio Supercomputing Center, currently maintained by Indiana University. [7]

Vendor Provided Several vendors such as IBM, SGI, Cray Research and HP provide MPI libraries for their platforms, affirming the stature of MPI as *the* message passing standard.

2.3 MPICH

MPICH is an open-source, portable implementation of the Message-Passing Interface Standard. The current stable version (1.2.5) contains a complete implementation of version 1.2 of the MPI Standard and also significant parts of MPI-2, particularly in the area of parallel I/O. The upcoming MPICH2 release [Section 7] will provide a complete implementation of MPI-2. The “CH” in MPICH stands for “Chameleon,” symbol of adaptability to one’s environment and thus of portability.

The central mechanism for achieving the goals of portability and performance is a specification called the *abstract device interface (ADI)* [8]. All MPI functions are implemented

in terms of the macros and functions that make up the ADI. All such code is portable. Hence, MPICH contains many implementations of the ADI, which provide portability, ease of implementation, and an incremental approach to trading portability for performance. One implementation of the ADI is in terms of a lower level (yet still portable) interface called the *channel interface* [9]. The channel interface can be extremely small (five functions at minimum) and provides the quickest way to port MPICH to a new environment. Such a port can then be expanded gradually to include specialized implementation of more of the ADI functionality.

2.3.1 Abstract Device Interface

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

Although MPI is a relatively large specification, the device-dependent parts are small. By implementing MPI using the ADI, code is made available that can be shared among many implementations. Efficiency is obtained by platform-specific implementations of the abstract device. For this approach to be successful, the semantics of the ADI do not preclude maximally efficient instantiations using modern message-passing hardware.

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

graphics device. MPICH uses the same approach in defining its message-passing ADI.

A message-passing ADI must provide four sets of functions: specifying a message to be sent or received, moving data between the API and the message-passing hardware, managing lists of pending messages (both sent and received), and providing basic information about the execution environment (e.g., how many tasks are there). The MPICH ADI provides all of these functions; however, many message-passing hardware systems may not provide list management or elaborate data-transfer abilities. These functions are emulated through the use of auxiliary routines, described in [8].

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

The routines at the lowest level in the ADI hierarchy can be in terms of a vendor's own existing message-passing system or new code for the purpose or can be expressed in terms of a further portable layer, the *channel interface*.

2.3.2 Channel Interface

At the lowest level, what is really needed is just a way to transfer data, possibly in small amounts, from one process's address space to another's. Although many implementations are possible, the specification can be done with a small number of definitions. The channel interface, described in more detail in [9], consists of only five required functions. Three routines send and receive envelope (or control) information and two routines send and receive data. Others, which might be available in specially optimized implementations, are defined and used when certain macros are defined that signal that they are available. These include various forms of blocking and nonblocking operations for both envelopes and data.

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

MPICH includes multiple implementations of the channel interface:

Chameleon Perhaps the most significant implementation is the Chameleon version. By

implementing the channel interface in terms of Chameleon [10] macros, portability is provided to a number of systems at one stroke, with no additional overhead, since Chameleon macros are resolved at compile time. Chameleon macros exist for most vendor message-passing systems, and also for *p4* [11], which in turn is portable to very many systems including Beowulf clusters.

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

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

Currently, on Beowulf class systems, MPICH utilises the p4 channel interface to provide message passing functionality. The p4 interface provides communication functionality across heterogeneous systems by using the TCP/IP protocol suite which, as is explained in Section 3.3, provides seamless, reliable communication across different platforms.

The basic idea behind the current work is to determine methods by which the communication performance of the current MPICH implementation on Beowulf class clusters can be improved. An understanding of the basics of data communication over networks is a prerequisite for working towards this goal. The following chapter provides a brief overview of data networks and the TCP/IP protocol suite.

CHAPTER 3

Computer Data Networks

Computer networks have revolutionized our use of computers. Computer systems used to be stand-alone entities. Each computer was self contained and had all the peripherals and software required to do a particular job. It was the need to *share* information and resources that changed this scenario. Today, probably the Internet is the most famous example of a communication network; but the range of features and services offered by data networks is impressive. To offer a few examples:

- Electronic mail has already revolutionized the way we communicate.
- File transfer across systems, either in the same room or continents apart, is a routine affair.
- Sharing of peripherals such as printers and scanners helps optimize resources.
- Remote login and remote application execution provide great ease in several situations.
- The ability to move data across a range of devices like cellphones, laptops and workstations provides unparalled power and convenience.

Communication networks can be divided into two basic types: *circuit-switched* and *packet-switched*. The classic example of a circuit-switched network is the original public telephone system (the current system is a frame relay based network, we shall not discuss that network here). When a telephone call was placed, a dedicated circuit was established by the telecom switching centres (telephone exchanges) from your telephone to the other telephone. There might have been a number of switching centres in between you and the other end, but once this circuit was established you were guaranteed exclusive access to it and the only delay in communication was the time it took for the signal to travel from one end to the other.

An internet (not *the* Internet, just any collection of interconnecting networks), on the other hand, typically uses packet-switching techniques.

3.1 Packet Switching

Unlike circuit-switched networks where a dedicated communication line is established between one party and the other, all parties which are on the network share communication links. The information to be transmitted is divided into pieces and each piece is transmitted on its own through the connection of networks. These pieces are called *packets*.

A packet is the smallest unit that can be transferred through the networks by itself. A packet must contain the address of its final destination, so that it can be sent on its way through the internet. With a circuit-switched network, we are guaranteed that once a circuit is established, we can use the full capacity of the circuit. With a packet switched network, however, we are sharing the communication bandwidth with other computers.

When a packet travels through an internet, it is *routed* through the various interconnecting entities that make up that particular internet. Since there is usually no predetermined path from source to destination for the packet to follow, there is no telling which route will it eventually take. Hence, the time taken for a packet to finish its journey is a function of the route it takes and the current volume of traffic on that route.

Packet-switching, by its very nature, gives rise to a number of issues which must be dealt with in order to facilitate data transfer. Some of the issues are discussed in the subsequent sections.

3.1.1 Gateways

When we speak of internets, we are talking about two or more networks connected by entities called *gateways*. The networks in question may be homogeneous running the same *protocols* [Section 3.2] and utilizing similar networking hardware or they may be different in terms of the protocols and/or the hardware used. A packet making its way through a homogeneous network is in easy waters but the one making its way through a heterogeneous network has to be translated as it crosses boundaries so that it can be understood by the network trying to read its destination information in order to route it towards its terminus. It's the gateways which provide this functionality, among other things.

3.1.2 Fragmentation

Most networks have a maximum packet size that they can handle, based on the characteristics of the physical medium over which packets are being sent. This is called the network's *maximum transmission unit* or *MTU*. For example, part of the Ethernet standard

[Section 4.2] is that the data portion of the frame cannot exceed 1500 bytes. The maximum for another kind of network called the token-ring network is typically 4464 bytes, while some networks have smaller MTUs. Consider a situation where a packet is starting from host *a* on network *A* and is destined for host *b* on network *B*. The MTU on the medium used in networks *A* and *B* is 1500 bytes but the MTU on the link joining *A* and *B* is 128 bytes. Obviously, something has to be done when packets are sent from *a* to *b*.

Fragmentation is the breaking up of a data stream into smaller pieces. Some networks use the term *segmentation* instead of fragmentation. The reverse of fragmentation is *reassembly*. Using these techniques, the problem of variable MTU can be overcome.

3.1.3 Modes of Service

When we speak of the kind of service provided by any network implementation, we generally talk of the following parameters:

- connection-oriented or connectionless
- sequencing
- error control
- flow control

A *connection-oriented* service requires that the two application programs establish a logical connection with each other before communication can take place. There is some overhead involved in establishing this connection. This type of service is often used when more than one message is to be exchanged between two peer entities.

The converse of this is a *connectionless* service, also called a *datagram* service. In this type of service, messages are transmitted from one system to the other, independent of other messages. This means that every message must contain all information required for its delivery. Section 4.3 provides more details on this.

Sequencing describes the property that the data is received by the receiver in the same order as it is transmitted by the sender. As mentioned earlier, in a packet-switched network, it is possible for two consecutive packets to take different routes from the source computer to the destination computer, and thus arrive at their destination in a different order from the order in which they were sent.

Error Control guarantees that error-free data is received at the destination. There are two conditions that can generate errors: the data gets corrupted (modified during transmis-

sion) or the data gets lost. The network implementation has to provide for recovery from both these situations.

Flow control assures that the sender does not overwhelm the receiver by sending data at a rate faster than the receiver can process the data. This is also called pacing. If flow control is not provided, it is possible for the receiver to lose data because of lack of resources.

Many of these parameters occur together. For example, sequencing and error control are usually provided together and the protocol is then termed *reliable*. It is also unusual to find a reliable protocol that does not provide flow control. It is unusual to have a connectionless protocol that provides sequencing, since the messages in such a protocol are usually unrelated to previous or future messages.

3.1.4 Addressing

The end points for communication are two user processes, one on each computer. With an internet, the two systems could be located on different networks, connected by one or more gateways. This requires three levels of addressing.

- A particular network must be specified.
- Each host on a network must have a unique address.
- Each process on a host must have a unique identifier on that host.

Again, a network implementation has to address this addressing issue, which can be quite complex if one thinks about the fact that there are millions on computers on the Internet and every process running on every connected computer must have a unique way to define its position on the Internet.

3.2 Of Standards and Stacks

3.2.1 Layering

The computers in a network use well-defined *protocols* to communicate. A protocol is a set of rules and conventions between the communicating participants. As we saw in the preceding sections, the complexity involved in implementing these protocols over a packet-switched network is quite daunting. This complex task could be solved by a single,

monolithic system or by breaking up the task into pieces and solve each piece individually. Experience has shown that the second approach leads to a better and more extensible solution.

It is possible that part of the solution developed for a file transfer program can also be used for a remote printing program. Also, if we are writing the file transfer program assuming the computers are connected with an Ethernet, it could turn out that part of this program is usable for computers connected with a leased telephone line.

In the context of networking, this approach of breaking down the task into simpler subtasks is called *layering*. We divide the communication problem into pieces (layers) and let each layer concentrate on providing a particular function. Well-defined interfaces are provided between layers. Very enough, these layers stacked one on top of the other are collectively called the *networking stack*.

3.2.2 OSI Model

The starting point for describing the layers in a network is the International Standards Organization (ISO) *open systems interconnection* model (OSI) for computer communication. This model, shown in Figure 3.1, was developed between 1977 and 1984 and is intended to serve as a guide and not a specification. It provides a framework in which standards can be developed for the services and protocols at each layer. In fact, several networks such as TCP/IP were developed before the OSI model. A study of the TCP/IP layering model will reveal that although it doesn't implement the 7-layer OSI model exactly, it provides similar functionality by a similar layering approach.

7	Application	*message*
6	Presentation	*message*
5	Session	*message*
4	Transport	*message*
3	Network	*packets*
2	Data Link	*frames*
1	Physical	*bits*

Figure 3.1: OSI 7-layer model with units of information exchanged at each layer.

One advantage of layering is to provide well-defined interfaces between the layers, so that a change in one layer doesn't affect an adjacent layer. It is important to understand that protocols exist at each layer. A *protocol suite*, or a *protocol family* as it is sometimes

called, is a collection of protocols from more than one layer that forms the basis of a useful network. TCP/IP is one such protocol suite.

3.3 TCP/IP - The Internet Protocols

3.3.1 Introduction

The Internet was first proposed by the US Advanced Research Projects Agency, as a method of testing viability of packet switching networks, and was later developed by the Defense Advanced Research Projects Agency.

In the early years, the purpose and usage of the Arpanet network was widely discussed, leading to many enhancements and modifications as the users steadily increased and demanded more from the network. User requests included the capability of transferring files from one university to another, as well as being able to perform remote logins and perform tasks as if the user were actually there on the premises.

As time passed many enhancements were made to the existing Network Control Protocol but by 1973 it was clear that NCP was unable to handle the volume of traffic passing through it. TCP/IP, developed by Kahn and Cerf in 1974, and gateway architecture were proposed as solutions. This protocol was to be independent of the underlying network and computer hardware as well as having universal connectivity throughout the network. This would enable any kind of platform to participate in the network. In 1981 a series of request for comments was issued, standardising the TCP/IP version 4 for the Arpanet.

Within 12 months the TCP/IP protocol had succeeded in replacing the NCP as the dominant protocol of the Arpanet and was connecting to machines across the United States. Today, it is the only protocol in use by the millions of computers which connect to the Internet.

TCP/IP became popular primarily because of the work done at the Berkeley university. Berkeley had been a leader in the unix development arena over the years and in 1983 they released a new version that included the TCP/IP networking stack as an integral element. That 4.2BSD version was made available to the world as public domain software. An optimised TCP implementation followed in 1988 and practically every other version of TCP/IP available today has its roots in the Berkeley version.

There are several interesting points about TCP/IP:

- It is not vendor-specific.

- It has been implemented on everything from handheld devices to supercomputers.
- It is used in all kinds of networks - small networks connecting systems in an office to global networks connecting systems across continents.

3.3.2 Layering in TCP/IP

Although the protocol family is referred to as TCP/IP, there are more members in this family than just TCP and IP. Figure 3.2 shows the relationship of the protocols in the protocol suite along with their approximate mapping into the OSI model described in section 3.2.2.

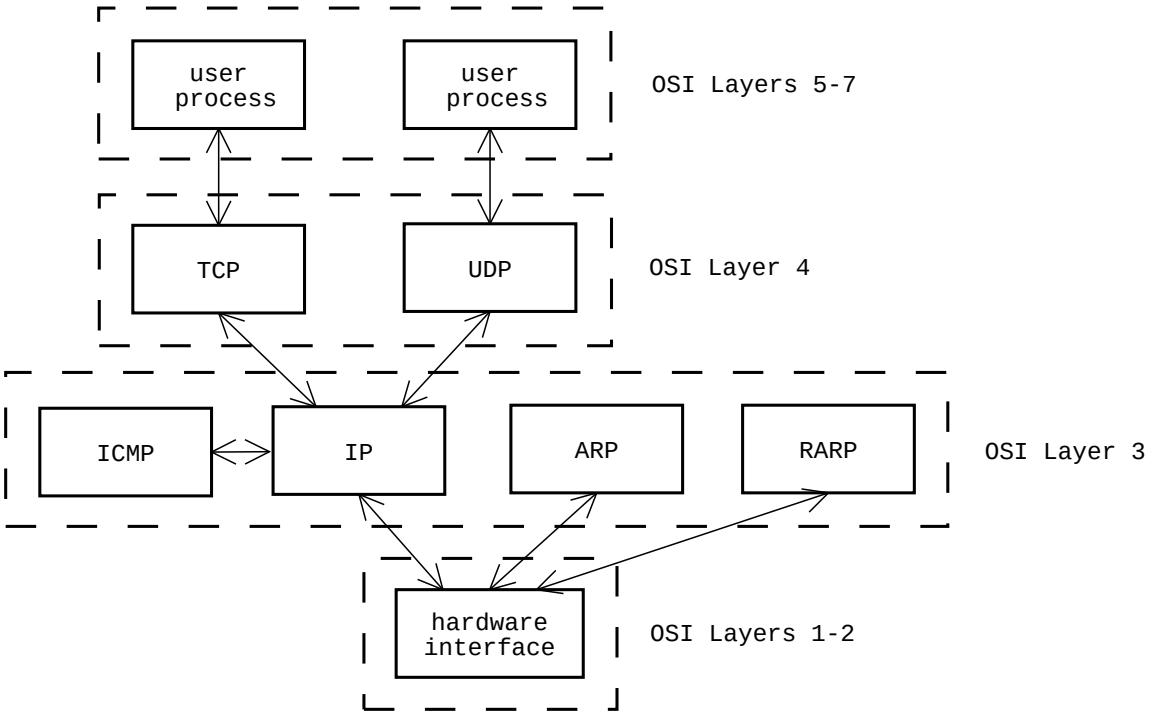

Figure 3.2: Layering in the TCP/IP protocol suite. [Adapted from Stevens [12]].

The TCP/IP protocol suite, with the services it provides at each layer provides solutions to all the issues discussed in Section 3.1. Also, as the growth of the Internet has demonstrated, it provides a robust and scalable implementation for a packet-switched network. Stevens [12] provides a detailed description of how TCP/IP handles all the problems described earlier.

TCP *Transmission Control Protocol*. A connection-oriented protocol that provides a reliable (page 16), full-duplex byte stream for user processes.

UDP *User Datagram Protocol*. A connectionless, not-reliable, protocol.

ICMP *Internet Control Message Protocol*. The protocol to handle error and control information between gateways and hosts.

IP *Internet Protocol*. IP is the protocol that provides the packet delivery service for TCP, UDP and ICMP.

ARP *Address Resolution Protocol*. The protocol that maps an Internet address into a hardware address.

RARP *Reverse Address Resolution Protocol*. The protocol that maps a hardware address into an Internet address.

3.4 System Area Networks

When the Beowulf concept was described in Section 1.2, we said it consists of a set of individual computing nodes interconnected by a high speed data network. Although this data network is similar to a Local Area Network, it would be incorrect to say it *is* a LAN. In fact, a better term to describe this kind of network would be *System Area Network* or SAN. This distinction is important because it is this distinction which forms the basis of the present work.

Since Beowulfs come in all shapes and sizes, it would be difficult to come up with a catch-all definition which describes all the SANs deployed on all the Beowulfs. A broad, mostly applicable definition would describe a SAN as a *high speed*, *switched* network connecting a cluster of computing nodes. Typical high-speed setups would be

Fast Ethernet Capable of 100Mb/sec transmission rate. Refer page 24.

Gigabit Ethernet Capable of 1Gb/sec transmission rate. Refer page 24

Myrinet This is a proprietary high performance network developed with clusters in mind [13].

Switched here refers to the way in which all the nodes are linked together. In typical LAN environments, a hub is used to link all the nodes on the LAN. Use of a hub means everyone is sharing the available bandwidth of the network. *Latency* is also high because any node which wishes to transmit has to wait till there are no other packets travelling on the wire. This also means that the network operates only in half-duplex mode - a node can either send data or receive data, not both at the same time.

Switches solve these problems associated with hubs by providing a high speed, temporary, one-to-one connection between any two nodes which wish to communicate. This

results in each node having the full bandwidth of the underlying medium at its disposal. It also results in low latencies and more importantly, permits full-duplex mode of operation - each node can send *and* receive data simultaneously.

3.4.1 TCP/IP - an overkill?

The question we raise here is: For a network such as the System Area Network described in Section 3.4, does the TCP/IP protocol suite provide an overkill of features? These unnecessary features - would their removal improve network performance?

Consider the former question first. To answer it, let us go back to the issues discussed in Section 3.1 and see how they hold up in the context of the SAN.

Gateways Since the system area network consists of a single network with no crossing of boundaries into another, possibly non-similar network, there is no need to provide translation services for migrating packets.

Fragmentation Again, since the packets always move on the same network, there is no question of having differing values of MTUs in the packets journey from source to destination. So fragmentation and reassembly are also non-issues.

Sequencing In a high speed, switched network it is extremely improbable that packets arrive at the destination out of sequence. A check implemented at user level for the minute probability of this occurrence, would be more efficient.

Error Control Error control for packet corruption is already implemented at the data-link layer (OSI layers 1 & 2). The reliability of a SAN doesn't warrant implementing this error checking functionality at all layers of communication. Only functionality making recovery of lost packets possible, need be provided.

Flow Control Typically, the compute nodes in the cluster have adequate buffers for the kind of data that is expected to be communicated. If ever a situation arises where a destination cannot process all incoming packets, it will start dropping packets. A simple polling technique at the source node will resend the dropped packets. Identical to what UDP packets do.

Addressing Since there is just one network, we can overlook network level address resolution. Also, in a SAN, the configuration of all nodes is know a priori. We know the MAC addresses of all the Network Interface Cards connected to the network. This means there is no need for address resolution at the host level. The process identification at each host can be implemented in user space or OS level, as the need be.

Thus, we can see that most of the features provide by TCP/IP are superfluous in the context of a SAN and a much simpler protocol can do the job.

3.4.2 Raw Ethernet Frames - A better approach?

In the context of Message Passing Systems which use TCP/IP for message passing, we have shown that the use of TCP/IP may not be the best possible approach. We now propose an alternative model for implementing communications in a message passing system.
The proposed model, depicted in Figure 3.3, uses just three layers for implementing a communication stack.

User Process	OSI Layers 5-7
Direct	OSI Layers 3-4
Hardware Interface	OSI Layers 1-2

Figure 3.3: Proposed 3 layer approach with approximate OSI model mapping.

The Direct layer will actually be implemented as a channel device for MPICH as described in Section 5.2. This layer will process the MPI messages to be sent on the network and prepare a raw ethernet frame in the appropriate form [Section 5.3.1] and pass it on to the Hardware Interface layer. This layer will add the error-control information and other information as required by the Ethernet protocol for a raw ethernet frame [Section 4.2]. Any MPI messages received by the Hardware Interface layer will be handed over to the Direct layer where it will be processed and the received data sent into the appropriate User Process layer buffers.

Note that this protocol is designed with just MPI in mind, where the best possible performance is sought. All other network services are best handled by TCP/IP since there are legions of network programs and tools which work with it.

Now we come to the second question raised above: would the removal of the nonessential features of TCP/IP improve performance? As can be seen, the alternative three layered approach outlined above does not provide for any of these features. Will this lead to improved performance? The answer to that is in the remainder of this document.

CHAPTER 4

Using Ethernet Frames

4.1 Introduction and Brief History

Ethernet is the most widely used local area network (LAN) communication technology. The original and most popular version of Ethernet supports a data transmission rate of 10 Mb/s. Newer versions of Ethernet called "Fast Ethernet" and "Gigabit Ethernet" support data rates of 100 Mb/s and 1 Gb/s (1000 Mb/s). An Ethernet LAN may use coaxial cable, special grades of twisted pair wiring, or fiber optic cable. "Bus" and "Star" wiring configurations are supported.

The first experimental Ethernet system was developed in the early 1970s by Bob Metcalfe and David Boggs of the Xerox Palo Alto Research Center (PARC). It interconnected Xerox Alto computers and laser printers at a data transmission rate of 2.94 Mb/s. In July 1976, Metcalfe and Boggs published their landmark paper entitled "Ethernet: Distributed Packet Switching for Local Computer Networks" in the Communications of the Association for Computing Machinery (ACM).

In 1979, Digital Equipment Corporation (DEC), Intel, and Xerox came together for the purpose of standardizing an Ethernet system that any company could use. In September 1980 the three companies released Version 1.0 of the first Ethernet specification called the "Ethernet Blue Book", or "DIX standard" (after the initials of the three companies). It defined the "thick" Ethernet system (10Base5), based on a 10 Mb/s CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol. It is known as "thick" Ethernet because of the thick coaxial cable used to connect devices on the network. The first Ethernet controller boards based on the DIX standard became available about 1982.

In 1983, the Institute of Electrical and Electronic Engineers (IEEE) released the first IEEE standard for Ethernet technology. It was developed by the 802.3 Working Group of the IEEE 802 Committee. The formal title of the standard was IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.

In 1985, IEEE 802.3a defined a second version of Ethernet called “thin” Ethernet, “cheapernet”, or 10Base2. It used a thinner, cheaper coaxial cable that simplified the cabling of the network. Although both the thick and thin systems provided a network with excellent performance, they utilized a bus topology which made implementing changes in the network difficult, and also left much to be desired in regard to reliability.

In 1990, a major advance in Ethernet standards came with introduction of the IEEE 802.3i 10Base-T standard. It permitted 10 Mb/s Ethernet to operate over simple Category 3 Unshielded Twisted Pair (UTP) cable. The widespread use of UTP cabling in existing buildings created a high demand for 10Base-T technology. 10Base-T also permitted the network to be wired in a “star” topology that made it much easier to install, manage, and troubleshoot. These advantages led to a vast expansion in the use of Ethernet.

In 1995, IEEE improved the performance of Ethernet technology by a factor of 10 when it released the 100 Mb/s 802.3u 100Base-T standard. This version of Ethernet is commonly known as “Fast Ethernet”.

In 1997, the IEEE 802.3x standard became available which defined “full-duplex” Ethernet operation. Full-Duplex Ethernet bypasses the normal CSMA/CD protocol to allow two stations to communicate over a point to point link. It effectively doubles the transfer rate by allowing each station to concurrently transmit and receive separate data streams. For example, a 10 Mb/s full-duplex Ethernet station can transmit one 10 Mb/s stream at the same time it receives a separate 10 Mb/s stream. This provides an overall data transfer rate of 20 Mb/s. The full-duplex protocol extends to 100 Mb/s Ethernet and beyond.

In 1998, IEEE once again improved the performance of Ethernet technology by a factor of 10 when it released the 1 Gb/s 802.3z 1000Base-X standard. This version of Ethernet is commonly known as “Gigabit Ethernet”.

The current IEEE 802.3 standards may be obtained online [14].

4.2 Frame Format

The Figure 4.1 illustrates the format of an Ethernet frame as defined in the original IEEE 802.3 standard.

Preamble A sequences of 56 bits having alternating 1 and 0 values that are used for synchronization. They serve to give components in the network time to detect the presence of a signal, and begin reading the signal before the frame data arrives.

Start Frame Delimiter A sequence of 8 bits having the bit configuration 10101011 that indicates the start of the frame.

Preamble	Start Frame Delimiter	Dest. MAC Address	Source MAC Address	Type	MAC Client Data	Pad	Frame Check Sequence
7 bytes	1 byte	6 bytes	6 bytes	2 bytes	0 - *n* bytes	0 - *p* bytes	4 bytes

Figure 4.1: Ethernet Frame Format

Destination & Source MAC Addresses The Destination MAC Address field identifies the station or stations that are to receive the frame. The Source MAC Address identifies the station that originated the frame. A Destination Address may specify either an *individual address* destined for a single station, or a *multicast address* destined for a group of stations. A Destination Address of all 1 bits refers to all stations on the LAN and is called a *broadcast address*.

Type The Type field indicates the nature of the MAC client protocol (protocol type). A list of Ethernet protocol type assignments is made available by IEEE [15].

MAC Client Data This field contains the data transferred from the source station to the destination station or stations. The maximum size of this field is 1500 bytes. If the size of this field is less than 46 bytes, then use of the subsequent "Pad" field is necessary to bring the frame size up to the minimum length.

Pad If necessary, extra data bytes are appended in this field to bring the frame length up to its minimum size. A minimum Ethernet frame size is 64 bytes from the Destination MAC Address field through the Frame Check Sequence.

Frame Check Sequence This field contains a 4-byte cyclical redundancy check (CRC) value used for error checking. When a source station assembles a MAC frame, it performs a CRC calculation on all the bits in the frame from the Destination MAC Address through the Pad fields (that is, all fields except the preamble, start frame delimiter, and frame check sequence). The source station stores the value in this field and transmits it as part of the frame. When the frame is received by the destination station, it performs an identical check. If the calculated value does not match the value in this field, the destination station assumes an error has occurred during transmission and discards the frame.

A more detailed discussion about the technical specifications of the Ethernet standard is available at http://www.techfest.com/networking/lan/ethernet.htm .

4.3 Sockets

On UNIX and derivative systems (including Linux), network communications take place through operating system abstractions called sockets. A socket, simply put, creates an endpoint for communication. Sockets come in two varieties: *connected* and *connectionless*. In the connected mode, both parties taking part in a communication open sockets at their respective ends. Once these two sockets are registered at both ends to be 'connected', data passed to a socket on one end will flow to the other end and vice versa. In the connectionless mode, when data is to be transferred, a socket is opened locally and data is written to it - unmindful of whether the receiving end has opened a corresponding receive socket. This method is typically used in setups where short bursts of data are sent and the overhead of setting up a connected socket connection doesn't justify using that mode.

A typical example of connected sockets in action would be when surfing the Internet. When a browser makes a request for a file from a web server, it is actually establishing a connected socket communication method between the local client machine and the remote web server. Once a connection is established, data starts flowing back and forth with no further negotiations necessary (unless, of course, the connection is broken). Tools such as **ping** and services such as *NFS* and *NIS* make use of connectionless sockets.

What makes sockets special is the fact that even a socket, like many other UNIX entities - a disk device, a pipe or a normal file - is just a file descriptor and hence, can be used as such. So one can simply establish a socket connection and use the standard **read** and **write** calls to achieve data transfer seamlessly over a network. Although it is possible to do that, it is rarely done, since there are better methods to do it. Several functions are available which make the process of reading and writing to sockets much more convenient and provide great flexibility and control.

The **socket** call is used to open a socket. The prototype of this call [16] is:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

The *domain* parameter specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in `<sys/socket.h>`. Some of the frequently used families are:

Name	Purpose
PF_UNIX, PF_LOCAL	Local communication
PF_INET	IPv4 Internet protocols
PF_INET6	IPv6 Internet protocols
PF_NETLINK	Kernel user interface device
PF_PACKET	Low level packet interface

The socket has the indicated *type*, which specifies the communication semantics. Some of the currently defined types for Linux 2.4.x are:

SOCK_STREAM Provides sequenced, reliable, two-way, connection-based byte streams.

SOCK_DGRAM Supports datagrams which are connectionless, unreliable messages of fixed maximum length.

SOCK_RAW Provides raw network protocol access.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the 'communication domain' in which communication is to take place.

The aim of the current work is to send and receive messages at the data-link layer, making use of raw ethernet frames. Hence, the *domain* to be used is **PF_PACKET** and the *type* is **SOCK_RAW**.

For this communication domain, the *protocol* is same as the **Type** field of the ethernet frame as described in Section 4.2. As mentioned earlier, the protocol numbers are assigned by IEEE and they are available as system defined constants in `<linux/if_ether.h>`. A sample listing of the file is as follows:

```
/*
 *      These are the defined Ethernet Protocol ID's.
 */
#define ETH_P_LOOP      0x0060     /* Ethernet Loopback packet     */
#define ETH_P_IP        0x0800     /* Internet Protocol packet     */
```

```
#define ETH_P_ARP       0x0806     /* Address Resolution packet    */

#define ETH_P_IEEEPUP   0x0a00     /* Xerox IEEE802.3 PUP packet   */

#define ETH_P_DEC       0x6000     /* DEC Assigned proto           */

#define ETH_P_RARP      0x8035     /* Reverse Addr Res packet      */

#define ETH_P_IPV6      0x86DD     /* IPv6 over bluebook           */

...

...
```

For the present purpose, any of the above ID's can be used, with the exception of ETH_P_LOOP which is reserved for communication local to the machine. We choose the protocol id as ETH_P_IPV6 since it provides a convenient filter. When the receive function [Section 5.3.3] is appropriately set, only packets bearing this protocol will be passed by the kernel to the receive function. Since no services on the SAN in consideration currently use this protocol, it is guaranteed that any frame bearing this protocol id has originated from the Send function [Section 5.3.2] of our program.
Thus, the socket call made in the send and receive functions will look like this:

```
sock_handle = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IPV6));
```

The **htons** function is to convert from host byte order to network byte order. More information on byte ordering can be obtained from "UNIX Network Programming" [12]. Once a socket has been opened, data can be written to it using the **sendto** [17] function and data can be read from it using the **recvfrom** [18] function. More information about using the low level packet interface can be obtained from `packet` man page [19].

CHAPTER 5

Message Passing Using Raw Ethernet Frames

5.1 Goal

The goal of the present work is to implement a few of the MPI point-to-point communication calls using raw ethernet frames and benchmark them against the standard MPICH calls which use TCP/IP for communication. The benchmarks will measure such variables as latency and bandwidth [Section 6.1], especially for small message sizes which are typical of many message passing algorithms in use.

5.2 Initial Approach - the channel device

As described in Chapter 2, the MPICH implementation of MPI uses a powerful layered approach and provides the *channel interface* which consists of the simplest point-to-point communication primitives based on which all other message passing routines are implemented.

Messages are sent in two parts: a *control part* containing information on the MPI message tag, size and communicator as well as about the message itself, and the *data part* containing the actual data. There are separate routines to send and receive the control and data parts, along with a routine to check to see if any messages are available.

A bare minimum interface [9] must provide at least the following five functions:

MPID_ControlMsgAvail Indicates whether a control message is available.

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

MPID_SendControl Sends a control message.

MPID_RecvFromChannel Receives data from a particular channel.

MPID_SendChannel Sends data on a particular channel.

Although conceptually the task of implementing a new *channel* device is a simple one, the lack of good documentation coupled with the poorly documented source codes of existing *channel* devices made this task very difficult. After many failed attempts, this approach was abandoned in favour of a standalone implementation which does not make use of the MPICH framework.

5.3 Standalone Implementation

After the inability to create a *channel* device, it was decided to implement a standalone framework which would provide simple send and receive MPI message passing calls using raw ethernet frames. Certain assumptions were made in designing this framework.

- All MPI processes are part of the same communicator:
  the default MPI_COMM_WORLD.
- Messages are short, less than 1488 bytes [Section 5.3.1].
- Messages are of type MPI_byte. The size of this datatype is one byte.
- The kernel provides adequate buffers for the raw sockets in use.

The rationale for making these assumptions becomes clear if we take a look at the goal of the current work [Section 5.1]. These assumptions, while making the task of implementing the framework simpler, in no way affect the quality or integrity of our benchmarks.

5.3.1 Ethernet Frame Format chosen

With the basic framework in place, we next define the frame format to be used for the raw ethernet frames. A basic blocking MPI send call has the following prototype:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

where

buf initial address of send buffer

count number of elements in send buffer

datatype datatype of each send buffer element

dest rank of destination

tag message tag

comm communicator

In our chosen framework, **datatype** and **comm** remain constant across all messages passed and hence need not be communicated explicitly.

A basic blocking MPI receive call has the following prototype:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

where

buf initial address of receive buffer

count maximum number of elements in receive buffer

datatype datatype of each receive buffer element

source rank of source

tag message tag

comm communicator

status status object

Again, **comm** and **datatype** can be ignored. The **status** is a structure which provides some information about the completed receive call to the programmer. It is not strictly necessary and can be ignored.

Thus, a receive call can sort through all received messages (raw ethernet frames) and return the one requested if it can determine the **tag**, the rank of the sender (**source**) and the size of the data received (**count**) for each ethernet frame it reads.

Based on this requirement, we defined the raw ethernet frame format as shown in Figure 5.1.

Note that the *Preamble*, *Start Frame Delimiter*, *Pad* and *Frame Check Sequence* bit sequences [Section 4.2] have not been shown here. This is because these bit sequences are added by the ethernet device (the Ethernet card) to every frame received by it from the

Dest. MAC Address	Source MAC Address	Protocol Id	Message Tag	Source Rank	Count	Client Data
6 bytes	6 bytes	2 bytes	4 bytes	4 bytes	4 bytes	0 - 1488 bytes

Figure 5.1: New Ethernet Frame Format Chosen

layer above it, the OSI Layer 2 [Section 3.2]. When a frame is written to a raw socket, it is enough if it contains the *Destination MAC*, the *Source MAC*, the *Protocol id* and the *data*. Also of interest is that the largest payload this frame can carry is 1488 bytes.

5.3.2 The Send Call

To prepare a raw frame conforming to the above described format, the Send function needs the following information:

- Destination MAC address
- Source MAC address
- Protocol Id
- The message tag
- The rank of the source process
- The number of bytes to be sent (*count*)
- The data to be sent

From the discussion on page 30 we know that the last four of the above are readily available from the calling process. The Protocol Id can be chosen from one listed in `<include/linux/if_ether.h>`. We choose the protocol id as ETH_P_IPV6 for reasons cited on page 28 of Section 4.3.

For the first two items on the above list, we need a mapping between the rank of a process and the MAC address of its ethernet interface. The **MPI_Get_processor_name** function provides a convenient way to get this mapping. This function returns the hostname of the node on which the calling process is running. Since we know the MAC address for the ethernet interface on each node (running `/sbin/ifconfig` will give the MAC address), we can map every process's rank to the hostname and hence the MAC address of the corresponding node. This information is propagated to all other participating processes by a call to the **MPI_AllGather** function.

With this mapping information, we have all the data needed to build a raw frame. The prototype of the Send function is as follows:

```
int send_direct(int dest_node, int source_node, int tag, int sender_rank,
                int count, void *data)
```

The **send_direct** function returns a 0 on error, 1 otherwise. The pseudo code for the Send function is as follows:

```
if socket not open
    open raw socket
    set socket parameters
get MAC addresses for source and destination nodes
prepare the raw frame
write frame to socket
end
```

5.3.3 The Receive Call

In our discussion on page 31, we concluded that the Receive function needs to sort through incoming frames based on the following parameters:

- The message tag
- The source process's rank
- The number of bytes sent

If an incoming frame matches these parameters as defined by the calling function, then the data part of this frame is returned in the calling function assigned receive buffer. The prototype for the Receive function is as follows:

```
int recv_direct(int tag, int sender_rank, int count, void *data)
```

The **recv_direct** function returns a 0 on error, 1 otherwise. The pseudo code for the Receive function is as follows:

```
if socket not open
    open raw socket
    set socket parameters
while required message not received
    read frame from socket
    if frame incomplete
        discard frame and read next frame
    if frame matches required parameters
        return data to calling function in receive buffer
    else
        read next frame
end
```

CHAPTER 6

Benchmarks

Benchmarking of parallel computing systems provides a method to compare the various high performance computing platforms available. Because of the high portability of MPI code and the fact that MPI implementations are available on almost all HPC platforms, benchmarking MPI performance often provides the best metrics to compare these systems.

Several benchmarks such as the NAS Parallel Benchmarks [20] from NASA Ames Research Center and the HPL benchmark [21], a distributed memory version of the popular Linpack benchmark, are available for measuring MPI performance. For the present work, we chose MPBench [22], part of the LLCbench [23] suite of benchmarks. From the MPBench website:

> ... a program to measure the performance of some critical MPI functions. By critical we mean that the behavior of these functions can dominate the run time of a distributed application.

6.1 MPbench

MPBench is a benchmark to evaluate the performance of MPI on Beowulf style clusters, Massively Parallel Processing systems and clusters of workstations. It uses a flexible and portable framework to allow benchmarking of any message passing layer with similar send and receive semantics. This makes it easy for us to use the same code to benchmark both MPI send-receive calls and the direct versions of these calls explained in Section 5.3. It generates two types of reports, consisting of the raw data files and Postscript graphs. The program does not provide any interpretation or analysis of the data,such analysis is left entirely to the user.

6.1.1 How it works

MPBench currently tests eight different MPI calls. The following functions are measured. The default number of processes used can be specified.

Benchmark	*Units*	*# Processes*
Bandwidth	Megabytes/sec	2
Roundtrip	Transactions/sec	2
Application Latency	Microseconds	2
Broadcast	Megabytes/sec	user defined value
Reduce	Megabytes/sec	user defined value
AllReduce	Megabytes/sec	user defined value
Bidirectional Bandwidth	Megabytes/sec	2
All-to-All	Megabytes/sec	user defined value

All tests are timed in the following manner.

1. Set up the test.
2. Start the timer.
3. Loop of operations over the message size as a power of two and the iteration count.
4. Verify that those operations have completed.
5. Stop the timer.
6. Compute the appropriate metric.

By default, MPBench measures messages from 4 bytes to 2^{16} bytes, in powers of two for 100 iterations. For the present work, we modified the test to run over message sizes from 4 bytes to 2^{10} bytes. Each test is run a single time before testing to allow for cache setup and routing. The cache is then flushed before each repetition and before each new message size is tested. The cache is not flushed however between iterations on the same message size, which are averaged.

In MPBench, calls to the timer around every operation are avoided, because this often results in the faulty reporting of data. Some of these operations take so little time, that the accuracy and latency of accessing the system's clock would significantly affect the reported data. Thus it is only appropriate that timing operations are performed *outside the loop*.

Only the following tests which measure the point-to-point performance parameters were run for the present work. In these tests there are two tasks, termed master and slave.

6.1.2 Bandwidth

MPBench measures bandwidth with a doubly nested loop. The outer loop varies the message size and the inner loop measures the send operation over the iteration count. After the iteration count is reached, the slave process *acknowledges* the data it has received by sending a four byte message back to the master. This informs the sender when the slave has completely finished receiving its data and is ready to proceed. This is necessary, because the send on the master may complete before the matching receive does on the slave. This exchange does introduce additional overhead, but given a large iteration count, its effect is minimal.

The master's pseudo code for this test is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    send(message size)
  recv(4 bytes)
  stop timer
```

The slaves' pseudo code is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    recv(message size)
  send(4 bytes)
  stop timer
```

6.1.3 Roundtrip

Roundtrip times are measured in much the same way as bandwidth, except that, the slave process, after receiving the message, echoes it back to the master. This benchmark is often referred to as *ping-pong*. Here the metric used is transactions per second, which is a common metric for database and server applications. No acknowledgment is needed with this test as it is implicit given its semantics.

The master's pseudo code for this test is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    send(message size)
    recv(message size)
  stop timer
```

The slaves' pseudo code is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    recv(message size)
    send(message size)
  stop timer
```

6.1.4 Application Latency

Application latency is something relatively unique to MPBench. This benchmark can properly be described as one that measures the time for an application to issue a `send` and con-

tinue computing. The results for this test vary greatly given how the message passing layer is implemented. For example, PVM will buffer all messages for transmission, regardless of whether or not the remote node is ready to receive the data. MPI on the other hand, will not buffer messages over a certain size, and thus will block until the remote process has executed some form of a `receive`. This benchmark is the same as bandwidth except that there is no acknowledgement of the data and results are reported in units of time.

The master's pseudo code for this test is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    send(message size)
  stop timer
```

The slaves' pseudo code is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    recv(message size)
  stop timer
```

6.2 Results

The tests were run on *Sruthi* [24], the Beowulf cluster run by the Aerospace Engineering Department, IIT Madras. Runs were carried out on two diskless nodes, and not on the main server and a node, to ensure maximum parity between the communicating entities. This way, we could ensure that the load on each client is (more or less) identical. Also, no other process was utilizing the network, save for the NFS calls which are used to transfer the benchmark code to the nodes from the server.

6.2.1 Bandwidth

The metric reported for this test is KB/sec and the results are shown in Figure 6.1. The increasing trend with size of messages is expected; an increase in message size will mean more KB/sec since the processing time for larger frames doesn't increase at the same rate.

Of interest in the chart is the widening gap between the two cases. Also, it is expected that both curves reach a plateau for higher message sizes. This is also consistent with the results of this benchmark run on other platforms. The chink in the TCP/IP curve at message size of 512 bytes is unexplained but it is suspected that it is due to MPICH switching the mode of the send call.

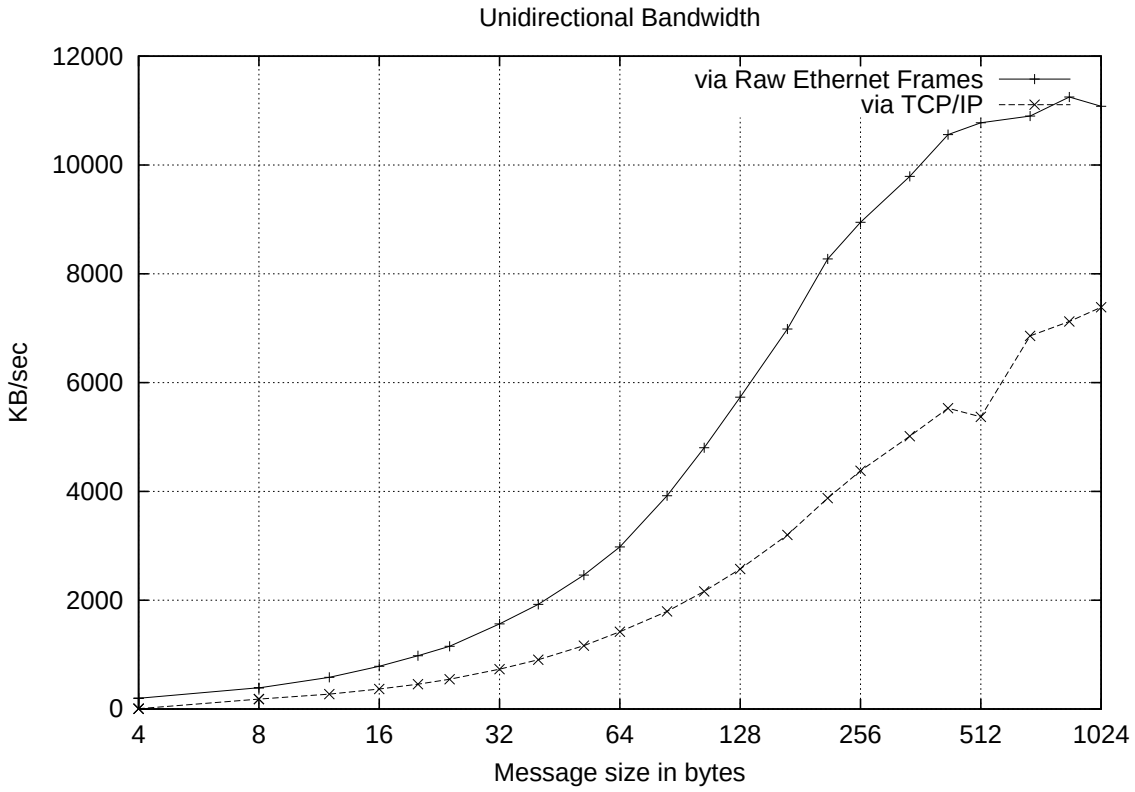

Figure 6.1: Benchmark: Bandwidth

6.2.2 Roundtrip

This test presents results, shown in Figure 6.2, in terms of transactions carried out per second. The value for the raw ethernet case is roughly twice that for the TCP/IP case for

small message sizes. For small messages, roundtrip time is largely dominated by protocol overheads and the means to access the network. This is the reason for the large difference in values for small messages for the two cases; the advantage that the raw ethernet implementation enjoys in terms of a light protocol diminishes with increase in message size.

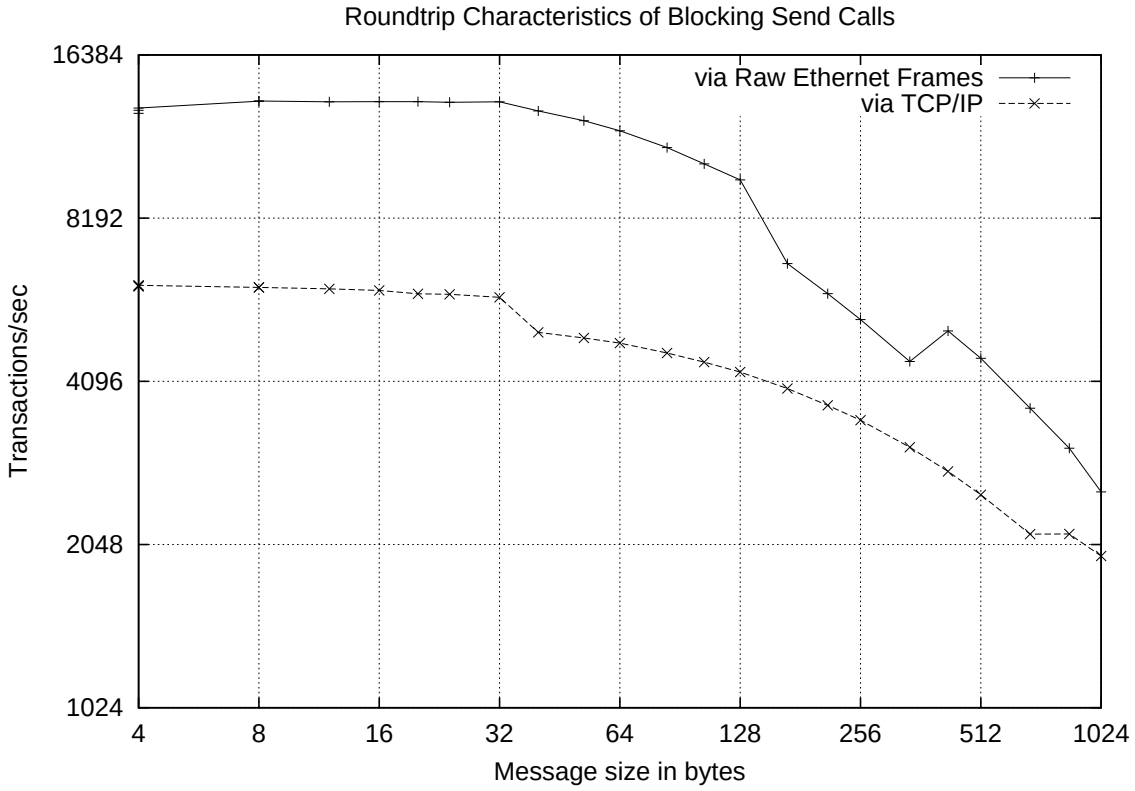

Figure 6.2: Benchmark: Roundtrip

6.2.3 Application Latency

The latency times for this test are reported in microseconds (Figure 6.3). Again, as can be seen, the light weight protocol provides substantial gains vis-à-vis the TCP/IP protocol. The steep values for the MPICH case for 4 byte messages are unexplained. These steep values kept recurring in various runs with different configurations.

Of interest in all these benchmark results is the almost constant values for the raw ethernet frames case when the message sizes were less than 32 bytes. This is easy to explain. The Ethernet frame specification [Section 4.2] allows for a 1500 byte *Client Data*

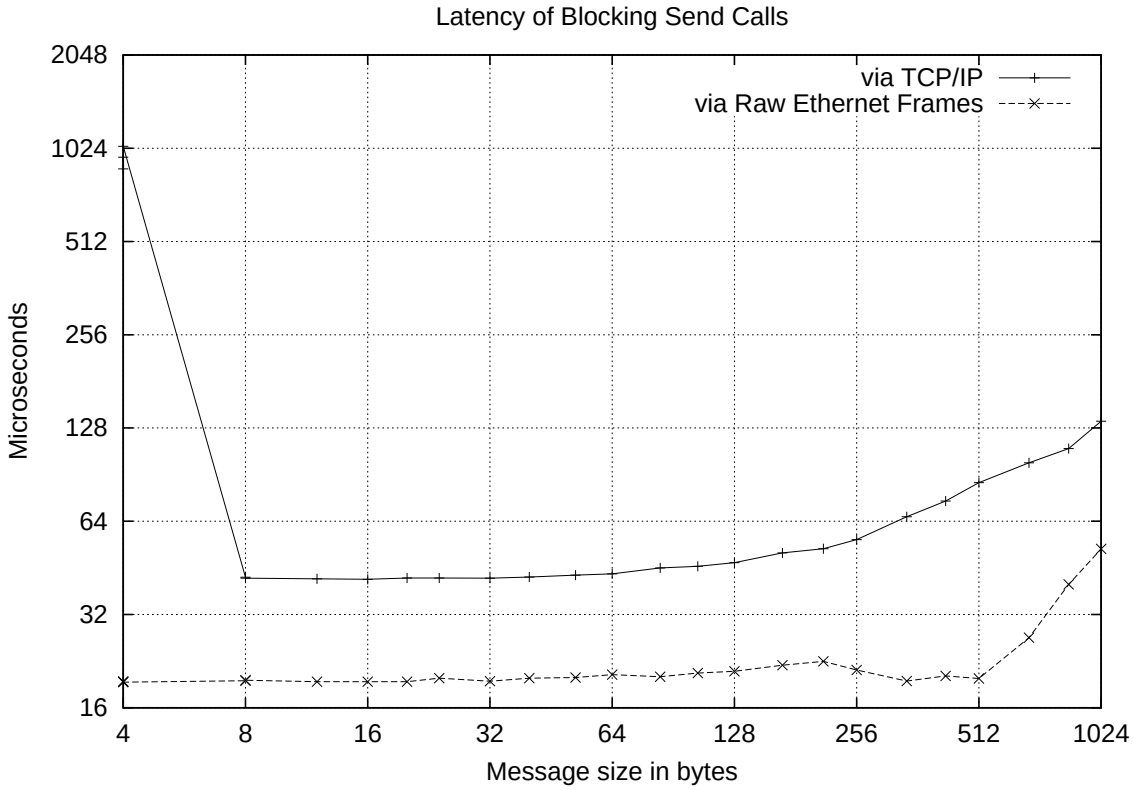

Figure 6.3: Benchmark: Application Latency

field, the minimum for this field being 46 bytes. As explained earlier, if this field is less 46 bytes long, then the subsequent *Pad* field is filled up till the size comes up to the minimum. The frame format adopted for the light protocol [Section 5.3.1] uses up 12 bytes of this Client Data field by way of the *tag*, *rank* and *count* fields. So if the actual message sent is less than 34 bytes, then the Pad field will be filled appropriately. Thus, the outgoing frame size for all message sizes less than 34 bytes is the same, which explains the results.

CHAPTER 7

Conclusions and Scope for Further Work

As has been demonstrated in the previous chapters, the use of raw ethernet frames for message passing has the potential of affecting a significant improvement in the communication performance in Beowulf clusters. The task of implementing a solution based on this concept remains. As explained in Section 2.3, MPICH provides a convenient method to achieve this.

The group at ANL recently released a beta-test version of MPICH2 [25], the next generation implementation of MPICH. MPICH2 provides several enhancements which include complete support for the MPI-2 standard, a redesigned and more powerful abstract device interface - *ADI3*, a new channel interface - *CH3* and a new interface called the *method interface*.

Of particular interest in the new version is a CH3 device implementation of the ADI on TCP. It defines a specific interface to the low level OS TCP operations. This implementation is designed to provide a relatively small interface to port MPICH to new platforms, in a manner similar to the channel interface in the earlier MPICH [Section 2.3.2]. The important change here is the doing away of the p4 layer which was used to provide communication functionality on workstation class implementations. This is expected to make the task of implementing a device which utilizes the raw ethernet frames functionality much more easier and straight-forward.

Also of interest in the new release is the method interface. This defines a set of operations that are needed to implement communication on a single connection. This interface allows multiple communication methods to be used in a single MPI program. This feature could be exploited to provide functionality wherein the user can switch between a raw ethernet frames based *Direct* protocol or any other protocol such as TCP/IP. This will be especially useful during the development life cycle of the Direct implementation. Since the ADI expects implementation of only a few specific calls in the communication device and provides a software emulation of all other MPI calls based on these core calls, it is possible that these software emulations are not as efficient as implementations of these calls at the

device level. If need be, the user can avail of more efficient versions of calls using other devices till such time as the Direct device provides those calls.

The upcoming release of MPICH promises to make the task of implementing a raw ethernet frames based communication framework both easy and attractive. Once such a framework is in place, the preliminary results made available by this work can be verified by real-world tests.

References

[**1**] High Performance Fortran Forum. High Performance Fortran Language Specification. Technical report, Rice University, Dept. Computer Science, October 1992.

[**2**] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. "TreadMarks: Shared Memory Computing on Networks of Workstations". *IEEE Computer*, Vol. 29, No. 2, pp. 18-28, February 1996.

[**3**] Message Passing Interface Forum, http://www.mpi-forum.org/

[**4**] MPI: a Message Passing Interface Standard, MPI Forum, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

[**5**] MPI Implementations, http://www.lam-mpi.org/mpi/implementations/

[**6**] William Gropp, Ewing Lusk, Nathan Doss and Anthony Skjellum. "High-performance, portable implementation of the MPI - Message Passing Interface Standard". *Parallel Computing*, Vol. 22, No. 6, pp. 789-828, September 1996. http://www-unix.mcs.anl.gov/mpi/mpich/

[**7**] LAM / MPI Parallel Computing, http://www.lam-mpi.org/

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

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

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

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

[**12**] W. Richard Stevens. "UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI". Prentice Hall, 1998, ISBN 0-13-490012-X

[**13**] Nanette J. Boden et al. “Myrinet—a Gigabit-per-Second Local-Area Network.” *IEEE-Micro*, 15(1):29–36, 1995. http://www.myri.com/myrinet/

[**14**] IEEE 802 standards: IEEE Local and Metropolitan Area Network standards. http://standards.ieee.org/getieee802/

[**15**] EtherType Field Public Assignments. http://www.standards.ieee.org/regauth/ethertype/type-pub.html

[**16**] Linux Programmer’s Manual. SOCKET(2). ‘`man 2 socket`’

[**17**] Linux Programmer’s Manual. SEND(2). ‘`man 2 send`’

[**18**] Linux Programmer’s Manual. RECV(2). ‘`man 2 recv`’

[**19**] Linux Programmer’s Manual. PACKET(7). ‘`man 7 socket`’

[**20**] NAS Parallel Benchmarks 2. NASA Ames Research Center. Version 2.4-beta, November 2002. http://www.nas.nasa.gov/NAS/NPB/

[**21**] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Version 1.0, September 27, 2000. http://www.netlib.org/benchmark/hpl/

[**22**] MPBench: MPI Benchmark. Philip J. Mucci, Kevin London, John Thurman. http://icl.cs.utk.edu/projects/llcbench/mpbench.html

[**23**] LLCbench: Low-Level Characterization Benchmarks. http://icl.cs.utk.edu/projects/llcbench/

[**24**] Midhun Kumar Allu. “Building and Benchmarking of Beowulf Cluster”. BTech. Dissertation, IIT Madras, May 2002.

[**25**] David Ashton, William Gropp, Ewing Lusk, Rob Ross and Brian Toonen. “MPICH2 Design Document: Draft.” Mathematics and Computer Science Division, Argonne National Laboratory.