A Network Processor Based Message Manager for MPI

by

Chamath Indika Keppitiyagama

B.Sc., University of Colombo, Sri Lanka, 1997

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

August 2000

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date 2000-08-03

Abstract

We have implemented a system called MPI-NP II, which is an MPI specific messaging system for the Myrinet System Area Networks (SAN). It consists of a low-level message manager executing on the LANai processor of the Myrinet Network Interface Card (NIC), a thin host interface layer, and LAM-MPI, a public domain version of MPI.

MPI-NP II is a re-design of MPI-NP that simplifies and improves the performance of the original implementation. MPI-NP differs from other low-level messaging systems in that it off-loads some of the MPI specific communication tasks onto the network processor. In particular, it manages MPI message envelopes and can progress messages asynchronously from the host. It realizes three of the goals stated in the MPI standard, namely; zero-copy messaging, overlap of communication and computation, and off-loading tasks to a communication co-processor. In addition, it greatly simplifies and reduces host/NIC interaction and makes it possible to support broadcasting on the NIC.

The design MPI-NP II introduces the concept of a microchannel, which is analogous to an independent thread on the NIC whose task is to deliver a specific message. The message manager allows for multiple outstanding send/receive requests and guarantees message delivery based on the available envelope resources, independent of the message size.

We achieve these design goals without unduly burdening the slow network processor. MPI-NP II has a minimum message latency of 22 microseconds and a maximum bandwidth of 92MB/s. These values are comparable to other low-level messaging systems but with the added benefit of being able to overlap communication and computation.

Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, Professor Alan Wagner, for his invaluable support and guidance throughout my work. This work would not have been possible without his guidance.

My thanks also go to Professor Norm Hutchinson, the second reader for this thesis, for his valuable comments. Thanks also to Professor Mike Feeley, Matt, Alex, Joon, Shree, Yanping, Yvonne, Ross, Dima, Kobi and rest of the gang at the systems lab for their help on numerous occasions.

I am grateful to Ashley Wijeyeratnam for his valuable support not only for this project but also to adjust to a place far away from my home. Finally I would like to thank my parents and my wife, Prabha, for their continuous support.

CHAMATH INDIKA KEPPITIYAGAMA

The University of British Columbia
August 2000

Chapter 1

Introduction

1.1 Motivation

Parallel processing is no longer restricted to high cost, vendor specific "*Inside the Box*" parallel processors. Low cost personal computers or workstations connected by a local area network have created a new form of computing called *cluster computing.* Several projects have shown that cluster computing is a viable alternative to high performance parallel processors [26, 36, 37]. The cost effectiveness of a cluster as a parallel processing environment can be attributed to the following:

- **Low cost PC's**: Rapid advancement in technology and market place dynamics have decreased the cost of processors and increased the performance. Personal computers, comparable to high performance processors of few years back, have become a household item.

- **Free Open source UNIX**: Linux and FreeBSD are examples of public domain UNIX flavors. These operating systems are free and available with the source code. UNIX has been the preferred operating system for the academic

world for some time. As a result, expertise in installing and configuring is readily available for academic and research institutions. Additionally, networking was part of the UNIX system from its inception.

- **Public domain message passing libraries**: Message passing is a natural programming paradigm for developing parallel programs in clusters. Its importance in parallel computing prompted 40 leading organizations representing both industry and academia to set up a forum to standardize the message passing libraries. The result was the Message Passing Standard (MPI) [24]. MPI is an ideal framework for parallel programming in clusters. The machine and language independent definition of MPI relieves the programmers from intricate details of the network and the operating system. Several public domain MPI implementations became available as soon as the MPI standard was finalized. LAM [9] and MPICH [21] are two public domain MPI implementations that are widely used. Several free distributions of Linux and FreeBSD include MPI implementations.

However the most important component, the network, that glues together the above components was not up to the task of high performance parallel computation. Earlier clusters were constrained by the slow 10Mb/s Ethernets and only coarse grained parallelism was possible because of this bottleneck. The appearance of fast Gigabit/sec commodity networks have made it possible to overcome this bottleneck and to bridge the performance gap between parallel processors and network clusters. Networks like Myrinet [4] are not only fast, they also have an onboard processor and memory. The onboard memory on the Myrinet network interface can be mapped onto the user space. This type of network interfaces are known as *User Level Network*

Interfaces. Although fast networks open the possibility of finer grained parallelism, message passing on top of traditional protocol stacks cannot deliver the performance of the network to the application.

Several projects were able to deliver a good portion of the raw bandwidth available on Myrinet to the host applications (e.g., FM [33], BIP [18], PM [12]). A general purpose Myrinet messaging layer is a salient feature of these implementations. The custom program running on the Myrinet interface is not aware of the nature of the program running on the host. There are several MPI libraries that use these low level Myrinet interfaces. MPI-FM [22] implements the MPI library using the FM Myrinet interface. MPICH-BIP [21] uses BIP as the communication layer.

The MPI standard states that its goals include avoiding memory to memory copying, offloading communication to a communication coprocessor where available and allowing for overlapping of computation and communication. The onboard processor, DMA engines and memory on the Myrinet network interface card can be used to achieve those goals. However MPI libraries implemented on general purpose Myrinet interfaces cannot deliver these goals. Chaussumier et al. [7] discuss the problems of implementing nonblocking MPI communication as asynchronous communication on BIP. Because BIP is not aware of the MPI communication semantics, the host must intervene to handle the control messages and acknowledgements. Chaussumier et al. suggest allowing the network processor to handle protocol issues up to MPI level as one of the solutions to this problem. However they also raise the question of practicability of this approach in the light of the resources available on the NIC.

Let's consider the potential of the Myrinet under the following scenario.

- Cluster is dedicated to parallel processing.

- Parallel programs are written using MPI.

In such an environment, the need for a generic Myrinet messaging layer does not exist. Since all programs are written in MPI, an MPI specific messaging layer on Myrinet can deliver the above mentioned goals of the MPI specification. MPI_NP [41, 1] was a step in this direction. Its goal was to allow overlapping of computation and communication utilizing the NIC processor. Performance results of MPI_NP suggested that the cost of implementing that selected subset of MPI on the NIC was relatively high.

Transparently to the user, the MPI system must perform numerous tasks; buffer management, data conversion, packing non-contiguous data and delivering the message to the correct receiver are some of them. To perform these tasks, the top layers must interpret both data and the *envelope*. The lowest layers only interpret the envelope which is a simple data structure. Correct delivery of the message depends on the matching of the sender's envelope with a receive request according to strict rules defined by MPI. Message matching is a core functionality of an MPI library. The remaining task is simply to transfer data from the send user buffer to the receive user buffer (complex data types must be unpacked at the destination). The low level Myrinet interfaces like BIP and FM do not interpret the MPI envelopes. It is up to the host processor to interpret the envelopes and to match sends with correct receives.

To use the NIC processor as a *communication coprocessor*, it is essential to handle MPI envelopes on the NIC. MPI envelope matching is not a computationally intensive task. Furthermore the storage requirement for an MPI envelope is relatively small (about 4 to 5 integers). Even with the current generation processors on the NIC, this task does not unduly tax the NIC processor. This reasoning directed

us to believe that the relatively high latency shown by MPI_NP was due to issues other than the envelope matching on the NIC.

Envelope handling on the NIC can further simplify the message matching process. NIC's can communicate to match the message and finally transfer the message without any intervention from the host. This cooperation of NIC's can be exploited further to provide a guarantee of message completion to the host application.

1.2 Thesis Statement

An MPI-aware control program on the NIC can reduce the host overhead, simplify the interface of the low level communication system to the MPI library, allow overlapping of computation with communication and can provide an efficient group communication mechanism.

1.3 Methodology

To support the thesis, we implement an MPI library that uses an MPI-aware control program on the NIC. We call the whole system (the MPI library and the control program on the NIC) MPI_NP II. The Myrinet control program is named the *Myrinet Message Manager for MPI* (M4). We implement message matching, envelope resource management, and some of the group communications in the M4. One of the core ideas in this design is that the NIC's should cooperate to reduce the frequency of host to host synchronizations and to provide asynchronous message transfer. Although this implementation is specific to Myrinet, the design can be adapted to any network interface with features similar to Myrinet.

We use LAM/MPI to provide the upper layers of the MPI library. The message manager is interfaced to the LAM/MPI at the RPI layer using another thin layer called the *Host Interface Layer* (HIL). We also wanted to show that the interface of an MPI-aware Myrinet control program is simpler than an interface of a generic control program.

Finally we compare the host overhead, bandwidth and latency of MPI_NP II with MPICH-BIP, which is another MPI implementation for the Myrinet. By taking MPI_NP II as an example we show that an MPI specific control program on the Myrinet reduces host overhead, simplifies the interface, progresses communication asynchronously and provides efficient group communication.

Chapter 2

Background

2.1 Message Passing Interface

Operating system designers have used message passing as a central concept in several popular operating system designs. Its use as a parallel programming paradigm was motivated by the concept of *communicating sequential processes* introduced by Hoare [3]. In this model, processes run in their own separate address spaces and communicate through explicit messages. In a distributed memory environment, message passing can be easily mapped to the hardware. Walker [6] discusses message passing approaches used in parallel computing. These include the use of coordination languages (Occam, Fortran M) and message passing libraries.

In the early 80's programmers had to use vendor-specific interfaces to write message passing applications. One of the major problems with these programs is the inportability. In response to this problem, several non-vendor specific programming interfaces were designed (e.g. PVM). Although the portability problem was solved through machine independent programming interfaces, libraries written using one specification could not be used with another specification.

The Message Passing Interface (MPI) forum was formed to produce a standard application programmers interface to message passing. The result was the MPI [24] standard. The MPI forum was an open forum and participants included representatives from both industry and academia. Rather than accepting an interface that was already popular, the MPI forum defined a new standard that includes features of popular message passing libraries. Hempel et al. [31] note that participation of major parallel computer manufacturers ensured that none of the machines were disadvantaged by the specification.

MPI provides a rich collection of message passing routines. These include point to point communication (send, receive), global communication (broadcast, scatter, gather) and global computation (reduce). The MPI communication universe is divided into several spaces called *communicators*. Communication in one communicator is insulated from communication in another. This communicator abstraction is important for independent development of libraries in MPI. Use of separate communicators in a library ensures that communication within the library is insulated from the communication of the other parts of the application. MPI communication can be viewed as taking place in a typed channel. Both the sender and the receiver have to specify the data type to receive a message correctly. The sender must explicitly specify the receiver, but the receiver can use the wild card ANY_SOURCE to receive messages from an unknown source. Although this can lead to nondeterministic behavior of the application, it improves the expressive power of MPI. The third parameter the messages must synchronize on is called a *tag*. It is an integer value that can be used to filter messages further within one communicator. The wild card ANY_TAG is allowed for the tag. However unlike ANY_SOURCE, use of ANY_TAG does not produce non deterministic code (if we don't use threads).

This happens because of the non-overtaking nature of MPI messages. The expressive power of MPI is enhanced by the rich collection of data types. Furthermore it allows programmers to define new data types. Non-contiguous data types are also allowed. The ability to arrange processes in different topologies allows for program code to be closer to the problem domain.

Message matching is a central feature of MPI. Correct delivery of a message depends on correctly matching the sender's and receiver's *envelopes*. The envelope contains the fields that MPI messages synchronize on (communicator or context, tag and the source on the receiver and the destination on the sender). The MPI standard requires that messages be non-overtaking and that this order must be preserved. Using nonblocking communication and different tags, a process can receive messages in a different order than they were sent (without violating the ordering rule of MPI). An implication of this behavior is that envelopes must be preserved until they are matched and messages delivered. This can result in overflow of envelope buffers on the remote node. Burns et al. [11] have shown that the space for MPI envelopes on the remote host is a limited resource that must be managed.

The MPI library must handle the complexity of the communication transparently from the user. From the user's point of view the communication medium is reliable and error free. If the underlying communication system is not reliable, it is the responsibility of the MPI library to provide reliability on it. An application can use complex user defined data types. Some of these data types may not be contiguous. Lower layers of the MPI system must gather these data in to a contiguous buffer wherever needed. Different data representations in a heterogeneous environment should also be transparent to the user. The MPI system is responsible for converting data to a common format before sending.

One of the contributing factors for the widespread use of MPI is the availability of public domain implementations of MPI. Some of the participants of the MPI forum took the responsibility of developing a library even before the MPI standard was finalized. MPICH [40] was the result. It was developed at the Argone National Laboratory and is available as a public domain software. Another public domain implementation from the Ohio Supercomputing Center, LAM [9], was soon to follow.

Success of the MPI 1.1 standard was soon followed by MPI-2 [25]. The MPI-2 specification added dynamic process creation, input/output and C++ bindings to MPI. Some researchers have also attempted to provide an interface for communication between different implementations of MPI. *Inter-operable MPI (IMPI)* [34] is a specification to provide this functionality. Hempel et al. [31] reflects upon the MPI development process, lessons learned from it and the success of MPI.

2.2 Myrinet

Myrinet[1][4, 29, 41] is a high speed network technology that is currently capable of providing duplex 1.28 GBits/second data rate. Myrinet network interfaces are connected to a crossbar switch with point to point links. Any topology is supported and the network can be scaled to a large number of hosts. Myrinet allows arbitrary length packets and there is no maximum packet size. Switches use wormhole routing to deliver these packets.

Myrinet Network Interface Card (NIC) has a dual context RISC processor (LANai), 3 DMA engines, SRAM and several registers. The host has access to the SRAM and some of the registers over the I/O bus. One of the DMA engines

[1]http://www.myri.com

(hDMA) is for data transfer from host to LANai memory (and vice versa). The hDMA can access the host memory through the EBUS (External bus) and LANai memory through the LBUS (LANai bus). The other two DMA engines (sDMA and rDMA) facilitate data transfer from the wire to LANai memory and vice versa. The Myrinet Control Program (MCP) executing on the LANai processor executes a program that typically schedules these DMA engines. The LBUS is shared by the processor and the DMA engines. The network processor has the lowest priority for the LBUS. On a rising edge of a clock cycle hDMA gets the highest priority followed by the processor. On the falling edge the priority order is rDMA, sDMA and the processor. When all three DMA engines are active the processor cannot access the LBUS and this stops the execution. Note that the LANai does not have a cache.

Myrinet provides an ideal setup up to implement *User-Level Network Interface Protocols* [30], where the operating system is removed from the critical communication path.

2.3 LAM: Local Area Multicomputer

The origin of LAM can be traced back to the Trollius [10] project. Trollius was an operating system for transputers developed at the Ohio Supercomputing Center and the Cornell Theory Center. In the Trollius world, the dedicated computing nodes (transputers) were referred to as *Inside the box computers* (ITB) and the general purpose computers were referred to as *Out of the Box computers* (OTB)[9]. ITB's ran Trollius natively while OTB's executed it on the host operating system. Trollius provided a consistent operating environment to parallel applications running on the both the ITB's and OTB's. The most important function of the Trollius core is synchronizing message passing [5].

LAM inherited the functionality and design of Trollius, however it only supports the OTB computers. Although LAM is not restricted to MPI, currently MPI programs are the main users of LAM [34]. Our discussion about LAM is specific to LAM 6.1 and its MPI personality, but we believe that the newest version of LAM is not significantly different from the LAM 6.1 architecture.

A collection of UNIX daemons (*lamd*) running on the cluster of host computers make the *kernel* of the multicomputer. These daemons schedule the collection of internal processes, and external processes that have attached to the kernel, and provide a multitasking environment. The LAM *kernel* is a rendezvous server and not an executive [8]. The kernel can launch a process but the process may or may not attach itself to the kernel. In the latter case the process runs as an independent process.

2.3.1 Bootstrapping the Multicomputer

The LAM boot process starts with the user invoking the *lamboot* command on the command line. This action will initiate the booting process on nodes specified by the user. *lamboot* uses the *rsh* command (or *ssh* depending on the configuration) to startup a process called *hboot* on each node. Before launching *hboot* on remote nodes, it also creates a server socket and passes the port and the IP address as command line arguments. *hboot* parses a configuration file passed into it as an argument and startup all the programs listed in it on the local node. *lamd* is listed in this configuration file. *hboot* passes the command line it received from the *lamboot* to the *lamd*. *lamd* now has the port and IP address for the server socket of *lamboot*. On startup *lamd* initializes the kernel IO interface which requires creating a named UNIX socket. Clients (the processes that attach to the kernel) use this to

communicate with the kernel. Then *lamd* initializes its process table and starts these processes. These processes are the *pseudo daemons* of LAM that provide different services. It is even possible to start these services as stand alone daemons without affecting the functionality, however, stand alone daemons affect the performance. The main server loop of *lamd* can schedule these pseudo servers. Each server has its own server loop, but without monopolizing the processor they return the control to the main server after performing one task. *lamd* invokes these services to service requests from the client processes that have attached themselves to the kernel.

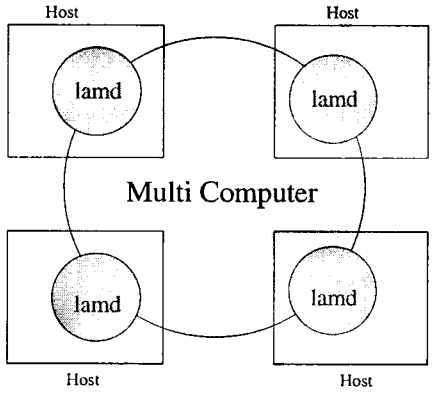

Figure 2.1: Local Area Multicomputer

One of the pseudo servers sends the link information (port of the local server socket etc.) to *lamboot* which collects this information from all the *lamd's*. *lamboot* distributes the complete link information to all the nodes. Now *lamd's* can connect to each other and create a fully interconnected topology where local *lamd* acts as the local interface to the multicomputer (Figure 2.1). Any process can attach to its local *lamd* through the known named socket created by the *lamd* and request services. Processes can request to launch a new process on the multicomputer, kill a running process attached to the multicomputer or to communicate with other processes.

2.3.2 Starting an MPI application on LAM

The *mpirun* command is responsible for launching the application in the Local Area Multicomputer. The process of launching an application starts by *mpirun* attaching itself to the kernel (local *lamd*) to initialize its kernel interface. Attaching to the kernel also creates a UNIX socket to the kernel, which allows *mpirun* to communicate with the kernel to launch the application. In the next step, *mpirun* builds the application by parsing the application schema file or the command line arguments. *mpirun* does not start the application program itself but instead sends a request to the kernel to launch the application. The local *lamd* passes this request to the remote daemons and they launch the processes as requested. Starting the application also involves closing the standard input and output file descriptors of the processes started on the remote nodes and redirecting them to the local node through *lamd*. Finally *mpirun* sends the process table to all the processes in the application.

On startup each MPI program should call MPI_Init() before proceeding to its message passing code. In LAM/MPI, MPI_Init() calls the LAM layer initialization routine. This routine must attach the process to the LAM kernel, as done by *mpirun*. Then it must receive the process table sent by *mpirun* through the kernel interface. After this step, each process of the application knows about the other processes belonging to the application (*world*). This information includes ranks of the processes, the nodes they are running on and the link information (Figure 2.2).

After exchanging process information, MPI processes can continue to use the *lamd* for communication or they can establish direct links for communication.

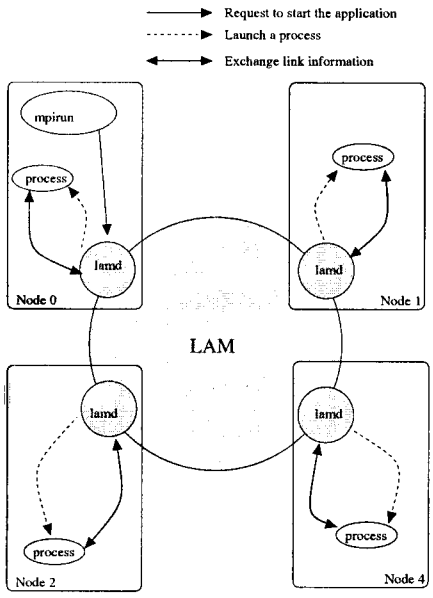

Figure 2.2: Launching an MPI application on LAM

2.3.3 Request Progression Interface

LAM/MPI implements the top layer of the MPI functions independent of the details of the communication system. The interface of this upper layer to the communication system is called the Request Progression Interface (RPI). The low level details of the communication is encapsulated in the RPI layer. LAM has several RPI's. These can be broadly divided into two.

- lamd RPI,
- c2c RPI.

Depending on the runtime flags, the system decides on which RPI to use. All LAM/MPI implementations have the *lamd* RPI. This RPI uses *lamd* to send and receive messages. Although it is slow, because the messages have to go through

the lamd, it provides facilities to monitor the progress of the messages. The c2c RPI bypasses *lamd* and processes communicate directly. Currently there are three different c2c modes implemented in LAM/MPI. They are TCP/IP which uses sockets for client to client messages, UNIX sysv and usysv shared memory modes. Only one of these three can be used and it must be selected at configuration time. The RPI layer is the place to implement a new communication system for the LAM/MPI. Details of the RPI layer is published in [19]. Personnel communication with the authors revealed that this document is continuously updated to reflect the current RPI layer. An MPI communication call on LAM/MPI goes though four main stages.

- Build a request: For all communication calls, a *request* is built by the library. The *request* keeps the current state of the communication. The function `_mpi_req_build()` allocates space for a new request and saves the rank, tag, context id, data type and the user buffer of this request. If needed, it allocates a separate buffer for packed data. Then `_mpi_req_build()` calls the relevant RPI layer function for the RPI layer specific initializations on the requests. The correct RPI function is selected through a user supplied flag to *mpirun*. If *lamd* RPI is selected then the `_mpi_req_build()` calls `_rpi_lamd_build()`. For the c2c RPI the matching function is `_rpi_c2c_build()`.

- Start the request: The function `_mpi_req_start()` starts the request. However at this stage the RPI layer is not required to actually initiate the data transfer [19]. The function `_mpi_req_start()` sets the request state and packs the data buffer if necessary. Then it calls its RPI specific counterparts, `_rpi_c2c_start()` for c2c mode and `_rpi_lamd_start()` for the *lamd* mode.

- Advance the request: Actual data transfer takes place in this state. The

function _mpi_req_advance() simply calls the RPI advancing functions, _rpi_c2c_advance() or _rpi_lamd_advance(). If the upper layer communication routine that called the _mpi_req_advance() is a blocking routine, then the RPI advancing function is called repeatedly until the blocking requests comes to the LAM_RQSDONE state. If it is not a blocking call, then the RPI advancing function is called repeatedly until it cannot advance any request further. Note that the advancing functions try to progress all the requests in the request list, not just one request.

- Complete the request: For all the requests that have reached the LAM_RQSDONE state, _mpi_req_advance() calls the _mpi_req_rem() to remove the request from the request list and then calls _mpi_req_end(). Data unpacking, if necessary, is done by _mpi_req_end(). If the request is a persistent request, then it is initialized for further communication, otherwise it is deallocated. _mpi_req_destroy() decrements the reference counts to the communicator and the data type, frees the pack buffer and calls the RPI specific destroy function. Both RPI specific destroy functions, _rpi_lamd_destroy() and _rpi_c2c_destroy(), return without doing any work.

In addition to the above, RPI layer specific routines are provided to initialize the application at the start (when MPI_Init() is called) and to cleanup the RPI state (when MPI_Finalize()) is called. _rpi_lamd_init() and _rpi_lamd_destroy() take care of the *lamd* specific initialization and finalization. The c2c specific initialization is done by _rpi_c2c_init() and cleanup is done by _rpi_c2c_finalize().

2.4 MPICH

MPICH [40] is a freely available MPI implementation[2]. Some of the developers of MPICH were also active participants of the MPI forum and they undertook the task of implementing MPI. In a manner similar to how LAM evolved from Trollius, MPICH has its roots in three earlier systems; P4, Chameleon and Zipcode. Several concepts of Zipcode (contexts, groups and *mailers* or communicators) were adopted by the MPI standard.

MPICH isolates the communication device dependent parts of the MPI in an abstraction called the *Abstract Device Interface (ADI)* [39]. The device independent code, which represents a relatively large portion of the code, is portable. The ADI must be implemented for each new environment.

ADI handles the packetizing, buffering and envelope matching and managing pending message requests, among other functions. Implementation of the ADI can take advantage of the capabilities of the hardware to provide these functionalities, however, that is transparent to the upper layers. Upper layers interact with the ADI through a well defined interface.

The ADI contains another interface called the *channel interface*. This layer provides the basic functionalities of moving the data and control messages from one process to another.

The ADI layer is analogous to the RPI layer of LAM/MPI and as in LAM/MPI, upper layer creates a *request* in response to a user call and calls the relevant ADI functions.

Portability of MPICH can be attributed to the ADI. MPICH has been ported to several environments by developing new ADI's. One such port particularly inter-

[2]http://www-unix.mcs.anl.gov/mpi/mpich/

esting to us, MPICH-BIP, is discussed later.

2.5 MPI Implementations for Myrinet

2.5.1 MPICH-BIP

MPICH-BIP [21] is an MPICH port that implements the ADI for communication over Myrinet. The ADI is developed on BIP (Basic Interface for Parallelism) [18], which is a very efficient interface for Myrinet. BIP provides efficient access to the Myrinet interface, bypassing the operating system and system calls, allowing for zero copy message transfer. BIP provides two messaging protocols.

- Long Messages: This is a rendezvous protocol. Receive has to be posted before the send.
- Short Messages: Small messages can be sent right away. If the message is not expected at the receiver, it goes to a circular queue. The size of this queue is finite and the send call blocks if the destination buffers are full. A credit based flow control is used for small messages.

The receiver does not specify the sender to receive a message, however, sends and receives are matched according to a *tag* in the message header. Only one receive can be active for one particular tag.

BIP is a low level protocol and it is not intended for use by itself to develop parallel programs. MPICH provides the upper layers to BIP to provide an MPI programming environment. Most MPICH ports use the channel interface to implement the new ADI. However, according to the developers of MPI-BIP [21], the new ADI was implemented by plugging BIP into a non-documented interface layer.

The BIP small message protocol is used for MPICH control messages and small application messages are encapsulated in the control messages. To avoid the overflow of fixed size queues on the receiver, a credit based flow control is used. These control messages include requests to send large messages, acknowledgements and credits for flow control.

BIP does not know about the MPI semantics and MPI message matching is done on the upper layers (inside the new ADI). As shown by Chaussumier et al. [7], the host must intervene by sending control messages to handle nonblocking communication. They point out that due to this limitation, it is not possible to implement nonblocking MPI communication routines as asynchronous routines in BIP.

2.5.2 MPI-FM

MPI-FM [20, 22] is another port of MPICH based on Illinois Fast Messages (FM) [32, 33]. The FM interface is similar to Active Messages [35], where each message carries an address of user level instruction sequence called a handler. This handler is invoked upon receipt of the message which in turn can act on the data. This model of message passing is closer to the hardware model and has shown improved performance over the conventional send/receive model.

FM 1.x provides three communication primitives; `FM_Send()` to send a long messages, `FM_Send_4()` to send a 4 byte message, `FM_Extract()` to process the received messages and execute their handlers. When sending a packet, data is copied into the NIC memory and the NIC is informed about the send. On receipt of the packet, it is DMAed to a queue on the host memory and `FM_Extract()` retrieves and processes the packet from this queue.

The MPI implementation on FM 1.x showed only 20% of the FM 1.x bandwidth [23]. This poor performance can be attributed to extra memory copying due to the mismatch of the MPI and FM protocols. In response to this, FM 2.x was developed with additional interfaces.

As with BIP, FM is also a general purpose interface. Therefore MPI functionality is completely handled in the upper layers on the host and host intervention is necessary to progress the message transfer.

2.5.3 PM

PM (1.2) [12] is a Myrinet messaging layer that provides two message transfer primitives; asynchronous and zero copy. Asynchronous primitives do not transfer the message directly from a user buffer; instead the user copies the data to a memory area accessible by the NIC and the NIC transfers the message. The same type of intermediate copy operations occur at the receiving end. The zero copy primitives transfer the message from the user buffer to a remote user buffer using *remote writes.* The sender must get the address of the remote buffer before transferring the message. Both the sender and the receiver must make sure that the memory pages of the user buffers are locked in the physical memory of the host. Assuming that the application reuses the buffers for another send or receive operation, PM keeps the virtual to physical address translation in a *cache* and unlocks these pages only when this cache is full.

MPICH provides the upper layers for the MPI implementation on PM. Tezuka et al. [12] discuss two implementations of MPI on PM. One implementation uses the asynchronous primitives and the other uses the zero copy primitives. The performance of the implementation that uses asynchronous primitives suffers due to

intermediate memory copies. The zero copy implementation sacrifices a considerable portion of the raw PM bandwidth because of the sender/receiver negotiations. These sender/receiver negotiations are necessary because the message matching and finally deciding the receive buffer is done by the upper layers and not by the PM.

2.5.4 MPI_NP

MPI_NP [41] is an MPI messaging layer designed for network processors and implemented for Myrinet. The goal of MPI_NP was to offload the computation related to communication to the embedded network processor. Unlike the other implementations of MPI for Myrinet, MPI_NP is not implemented on a general purpose network interface protocol. The custom Myrinet control program developed for MPI_NP is an MPI specific control program. MPI_NP uses LAM/MPI as the MPI library and implements a Myrinet RPI.

MPI_NP provides a *channel* abstraction for host processes to communicate. An MPI_NP channel is a bidirectional communication path between every two MPI processes and created at application initialization time. A channel also represents the basic scheduling unit on the NIC. The Myrinet control program schedules the channels and manages the message queues (envelope queues) and data buffers on the NIC. Channels are flow controlled through a credit based mechanism. If a channel does not have sufficient credits to send a message, it goes to the blocked state and the MCP does not schedule the channel until it gets sufficient credit.

MPI_NP provides three messaging protocols; full credit, rendezvous and eager sending. If the channel has enough credit to send a message, the local NIC sends it immediately and the message is buffered on the receiving NIC. When the receiving host extracts the message, credits are send back to the sending NIC. The

rendezvous messages have two cases due to the expected and unexpected messages. An expected message is a message for which corresponding receive is already posted and unexpected messages have to be kept in a queue waiting for receives. Small messages are sent even without the credit with the assumption that there is enough space on the receiver and receiver is only slow in replenishing the credits. If this assumption is wrong, the receiver will drop the message. However, the receiver notifies the sender whether it has accepted or dropped the message. In the case it has been dropped, the sender will not send the message again until it has enough credits. This is known as the eager protocol. Note that message matching is completely done on the NIC.

The Myrinet control program also manages buffers on the NIC. To support zero copy (i.e., to avoid memory to memory copy), MPI_NP uses the memory on the NIC as a system buffer. It is not always possible to allocate a contiguous message buffer on the NIC, nor it is possible to accommodate any size message on the NIC. When the messages are claimed by the host or sent to the receiving NIC, message buffers are reclaimed by the NIC. However, over time message buffers get fragmented and the MCP is further burdened by the buffer management.

Myrinet DMA engines need the physical address of the host data buffer in order to transfer data from the host buffer to the NIC buffer. The host must also ensure that these pages are locked in the physical memory (i.e., the operating system does not swap out the pages). MPI_NP uses a loadable kernel module to provide the page pinning functionality to the application running on the host. MPI_NP locks pages on each send (or receive) call and unlocks them before returning from the call.

The current implementation of MPI_NP only provides blocking communications. The performance figures reported in [41] show that MPI_NP has one way

minimum message latency of 68 microseconds and maximum bandwidth of 72 MB/s.

Chapter 3

Design

3.1 Evolutionary Notes

MPI_NP II builds on our previous experience with MPI_NP. My involvement with MPI_NP started with experimenting with the MPI_NP code at the LAM RPI layer to add nonblocking communications. Although MPI_NP included nonblocking communication in its design, the Myrinet RPI layer for LAM, implemented by A. Wijeyeratnam, had only a blocking interface. This work involved a study of the functionality and the implementation of the LAM RPI layer. To implement nonblocking communications, the blocking Myrinet RPI layer had to be broken into several parts to map with the standard RPI layer interfaces. It was a valuable experience in understanding the problems encountered by implementors of MPI libraries. This study has shown how a single threaded MPI library progresses messages, specially nonblocking messages. In LAM, a communication call translates into a progression of all the messages not related to the current call, but left behind by the previous nonblocking calls. A nonblocking send/receive call will start the request and progress it as far as possible. However it does not block during this process. When a request cannot

be progressed further, the current state of the request must be saved and control returned to the upper layer. This request will be progressed further when another communication request is issued by the application (the two requests need not to be related). We had to break down the blocking MPI_NP send/receive routines to several functions to match this mechanism. We also had to modify the request data structure to save the progress state, so that the request can be progressed later.

To reduce the overhead of sending a message, we have also experimented by removing the upper layers and interfacing the send routine directly with the low level Myrinet send routines. We could only save 3 microseconds in this experiment (reduced the overhead from 7 microseconds to 4 microseconds in the best case). However the resulting code broke the architecture of LAM/MPI.

We had to study the LAM/MPI in general and the RPI layer in specific, to make the changes mentioned above. However none of the above work involved changes to the MPI_NP Myrinet control program. The changes were done by modifying the code running on the host, that interacts through a well defined interface with the Myrinet Control Program (MCP). The next evolutionary step involved adding features to the MCP.

Our original intent was to start from the MPI_NP and to add new functionality to it. These included broadcasting, dynamic process creation, PUT/GET operations etc. In summary, we wanted to use our Myrinet control program with newer versions of LAM/MPI that supports MPI-2 specification (partially). We also wanted to explore the possibility of reducing the latency and increasing the bandwidth. This work required major changes to the MCP, and we encountered several problems in attempting to change the MCP.

- Optimized, Non Modular code: The MCP was written to get maximum perfor-

mance out of the slow processor on the network card. Code level optimizations were used to save space on the limited memory and for fast execution. The result was non-modular code.

- Steep learning curve of Myrinet: Developing correct code to run on the Myrinet interface was a difficult task. We came across the following difficulties.
  - We could not find any proper debugging tools for the Myrinet programs.
  - Myrinet programs have to deal with the asynchronous operations of the on-board processor, host program and the DMA engines. Some registers need a certain number of instruction cycles to change from one state to another. This further complicates the programming on the NIC.

Soon we found that changing non-modular code in an asynchronous environment, without proper debugging tools, was a steep task. Since major changes to the original design were intended in our new design, we decided to start writing the program from scratch. However, we were able to use some code segments and structures from MPI_NP. On the LAM side, the old Myrinet RPI was no longer usable with the new design and a new RPI layer was needed. The most important MPI_NP code preserved on the LAM side, in the new design, is the code that exchanges the link information. MPI_NP code was tightly integrated into the LAM code which made it difficult to test MPI_NP without LAM. We decided to develop MPI_NP II as a separate library that can be linked with the LAM code. This enabled us to test the library separately and to find out whether LAM introduced any overhead.

3.2 Reflection on MPI_NP

MPI_NP reported relatively high latency for small messages. The minimum latency was 68 microseconds [41], where 75% of that time was spent on the sending and receiving NIC's. It is interesting to note that 30 microseconds was spent on the receiving NIC and 22 microseconds was spent on the sending NIC. Experiments on the same environment with BIP is also reported in [41]. Minimum message latency for BIP was approximately 12 microseconds in the same environment. MPI_NP provides two major functionalities not found in BIP. They are MPI message matching on the NIC and support for multiple outstanding messages. However these additional functionalities cannot explain the extra time spent on the NIC. Message matching is trivial in the common case (i.e., only one outstanding message, with the matching request immediately posted). Since the common case was used for the micro bench marks, additional MPI functionality itself cannot explain the additional cost. MPI_NP also uses a pessimistic page pinning scheme. It pins pages at the start of each send/receive call and unpins them on the completion of the call. BIP caches pages after the first call, so messages from the same buffer do not incur page pinning costs on subsequent calls [18] (BIP does not use page pining on small messages). However the page pinning cost is a cost incurred on the host and it cannot explain the 75% of the time spent on the NIC. Furthermore small messages do not incur page pinning costs on the send side and only one system call will be on the critical path at the receiving end. To investigate the reason for the overhead on the NIC, we needed to break down the cost even further. However the complex code made the further instrumentation difficult. We decided to analyze the system at a higher level.

MPI_NP had a complex protocol implemented on the NIC. This protocol

handled several cases including expected messages, unexpected messages, eager send, large messages and small messages. Complete details of this protocol can be found in [41]. All the communication between two processes were through a *channel.* A channel represented a bi-directional communication path between two processes. The unit of scheduling on the NIC is the channel, which is comparable to a process in the operating system and like a process each channel goes through different states. Communication is flow controlled by a credit mechanism and the channel is put into a blocked state if it uses up all its credits. This is an interesting point in this discussion. Consider the following scenario. A message consumes all the credit, but there is no matching receive posted at the receiving end. In this case the channel is blocked and the NIC will not schedule it until it has the credit. This blocked message can stay unmatched even after the receiving end posted receive requests to the capacity of the the receive request queue. If there is no matching send, the receive side will send the request to the send side of the channel to be matched. However, this request is matched only with the head of the blocked channel to avoid complete implementation of message matching on the sender. A consequence of this protocol is that the other messages posted after the blocking message cannot proceed even if they have matching receives posted at the other end. If the channel remains blocked even after the receive end used up all the space for the receive requests, then the message progress never occurs, resulting in deadlock. This scenario is shown in Figure 3.1.

This protocol imposes a strict limit on the *look ahead.* The receiver must post a matching receive request within a limited window. Until a matching receive is posted for the send that consumed all the credit, other messages have to be held on the channel. This can happen even if the other messages have matching receives

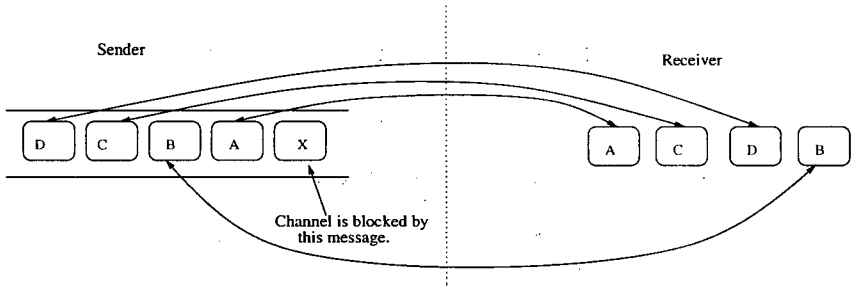

Figure 3.1: Messages on a blocked channel

posted.

On top of the complex protocol, MPI_NP also did message buffering on the NIC. On the receiver's end messages can be uploaded to the host in a different order than they arrived. This leaves holes in the buffer space. MPI_NP maintains a linked list to keep track of the non contoguous free areas. On receipt of a message, NIC must find a free space to keep the message and on transferring the message to the host, the NIC must reclaim this area. This buffer fragmentation is shown in [41] and it can impose a heavy overhead. Pakin et al. [33] noted that on Myrinet even mundane pointer and looping overheads can degrade the performance significantly.

3.3 Design Overview

One of the design goals of MPI_NP II was to improve the performance of MPI_NP. We decided to simplify the protocol to improve the performance. However we wanted to keep the MPI message matching on the NIC. We have identified that an MPI message consists of two components, the envelope and the payload. Only the envelopes are needed for the message matching and message matching is the central functionality of an MPI system. Note that the message ordering rule of MPI is es-

sentially the ordering of the envelope. The ordering rule is to match the sender and the receiver in the correct order. The envelopes can make sure that the sends are delivered to the correct receives. The payload does not (and should not) have any impact on this ordering. After packing the non contiguous data, only the envelope is interpreted by the MPI system. Furthermore message buffering on the system is optional on MPI calls. Gropp et al. [40] note that if the system is responsible for message buffering, user programs can fail in mysterious ways. If the user wants message buffering on the system, MPI provides separate buffered calls (`MPI_Bsend()`). For a buffered send the user must first register a buffer with the system. We decided to remove the data buffering from the NIC and to retain only the envelope handling system. The new design is based on the idea that the NIC must act as an envelope manager and not as a buffer manager. The payload is transfered directly from the sender buffer to the receiver buffer with the NIC's acting as a staging area. This is essentially a rendezvous protocol. To reduce the impact of a rendezvous protocol on small payloads, such payloads are considered as part of the envelope.

Since the NIC has only a limited memory it can handle only a limited number of envelopes. Burns et al. [11] have shown that envelope space is a limited resource that must be managed. They also point out the importance of the MPI system to advertise its guaranteed number of envelope resources. LAM/MPI advertises its Guaranteed Envelope Resources (GER). Bruck et al. [14] also define *k-buffer correctness* for MPI programs running on their system depending on the number of buffers available on the receiver. With the limited space on the NIC, this concept is ever so important in our design. This upper bound on the number of envelope spaces per channel imposes an upper bound on the number of unacknowledged messages on a channel. However, our design, as discussed later, does not impose a rigid look

ahead as in MPI_NP. The new design allows messages to be remain unmatched for an indefinite time without blocking the other messages. For example, the first send posted at the start of a process can be collected by the receive posted at the end of the receiving process, provided at least one envelope resource remained matched or free all the time. We consider the page locking/pinning as orthogonal to this research and do not discuss the issues related to page pinning. We assume that pages are on the physical memory when the send/receive pair is matched. On our implementation we pre-pin pages and cache them on the NIC and messages are send from this pinned area. Page pinning is an important issue and a technique is discussed in [12].

Our design of an MPI communication system on Myrinet consists of three components (Figure 3.2). The design is not specific to Myrinet. Our only assumption is a network interface with memory and a processor. However, the design can best described with respect to our implementation, which is specific to Myrinet.

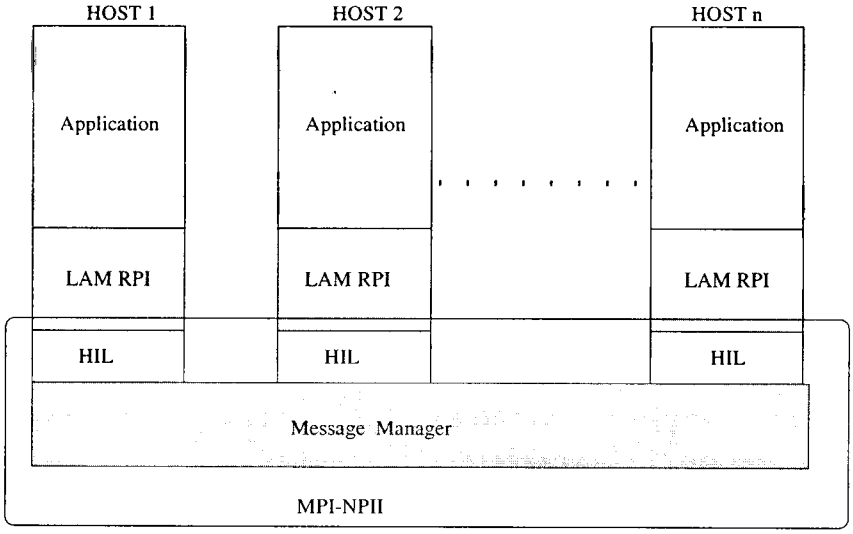

Figure 3.2: MPI_NP II architecture

The three components of MPI_NP II are:

- Myrinet Message Manager for MPI (M4).
- Host Interface Layer (HIL).
- A mechanism to plug in the above two components to the LAM-RPI.

3.4 Myrinet Message Manager for MPI (M4)

The major component of our design is M4 and the bulk of the implementation time was spent on implementing this. The task of M4 is to handle the MPI envelopes and to transfer data from the send user buffer to the receive user buffer. Functionality and the design of M4 can best be described by comparing and contrasting it with the design of LAM itself. Our analysis of LAM comes in useful here.

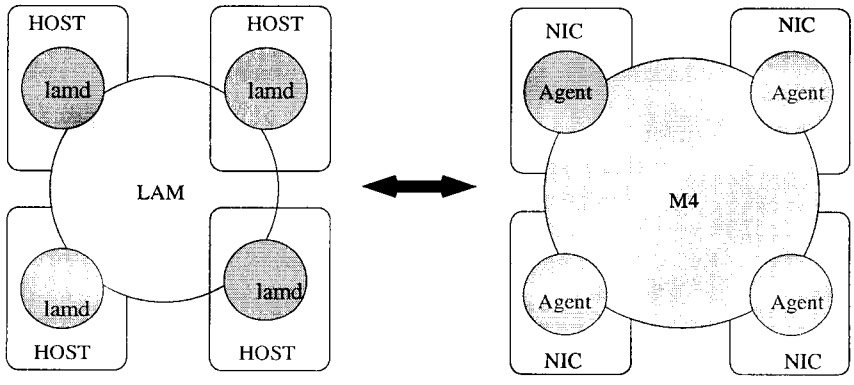

Figure 3.3: Comparing M4 and LAM

- *Local Area Multi-Computer* is formed by local daemons (*lamd*) running on host processors. Local processes interface with the multi-computer through the local *lamd*. M4 is formed by a set of Myrinet control programs running

on NIC's. We call these control programs **Local Agents**. Local processes interface with M4 through the local agent (Figure 3.3).

- *lamd's* form a fully connected topology using sockets. Local agents also from a fully connected topology through the Myrinet switch. The only software component needed on each local agent to form this topology is a simple routing table.

- Local process, on startup, must initialize its *lamd* interface. This involves creating a UNIX socket to the local *lamd*. The MPI_NP II process must initialize its interface to M4. This is done by mapping a shared memory area on the NIC.

- The dominant functionality of LAM is to provide a mechanism for communication between processes attached to it. The *only* functionality of M4 is to provide a communication mechanism to MPI programs interfaced with it. LAM daemons do not know about MPI (LAM was designed as a general purpose multicomputer and MPI was implemented on top of LAM). However, M4 can interpret MPI envelopes.

- LAM can launch processes, send signals to them, and schedule them. M4 does not have any of these capabilities. It simply acts as a message manager. In our implementation the top layers use LAM for launching and managing processes.

- Users can query the *lamd* to get the status of processes and messages. In the same way users can query the local agent to get the state of the messages in our system.

In summary, this design is analogous to implementing the communication functionality of LAM as a separate asynchronous process running on the NIC's. Other than that, some parts of the upper layer responsibilities (envelope matching) were also delegated to M4. A separate process to handle communication is not peculiar to LAM. PVM [38] also has daemons to handle various tasks, including communication. Our design utilizes the NIC processor and memory to implement this *daemon* on the NIC, which is a logical place for the communication *daemon* to reside.

The design of M4 is centered around two abstractions, namely the *Channel* and the *microchannels*. Channels represent the bulk and static communication state between two processes within a single MPI communicator. Microchannels represent the transient communication state for a message within a channel. The notion of microchannels is provided to capture the idea of *look ahead* in nonblocking communication and message scheduling. We will discuss these ideas in the following sections.

3.4.1 Channels and Microchannels

The common notion of a channel is a communication path between two processes. Messages sent from the sending end are received in the same order at the receiving end. However this does not cleanly capture the MPI communication semantics. MPI messages can be received in an order other than the order they were sent. Strangely enough this does not violate the message ordering rule (non overtaking messages) in the MPI standard. The key to this disorder is the message matching on the *tag*. The receiving process can selectively receive messages by the message tag. If the channel can provide this semantics transparent to the communicating process, then

the interface of the process to the channel will be extremely simple. We can call such a channel an *MPI channel*, since it provides the MPI semantics transparently to the user. One implication of this out of order message receiving is that messages must be preserved until the receiver asks for them. Both ends of the channel must keep the state of the message alive until the payload is delivered. As discussed later, the sender reserves a send slot on the channel and the receiver reserves a request slot on the channel for a single message. Then they both use these slots to send and receive that particular message (or payload) as a *stream*. This represents a channel, in the common notion, within this MPI channel. We call this a *microchannel* and it represents the transient state of the communication of one message. This concept has an analogy in the domain of operating systems.

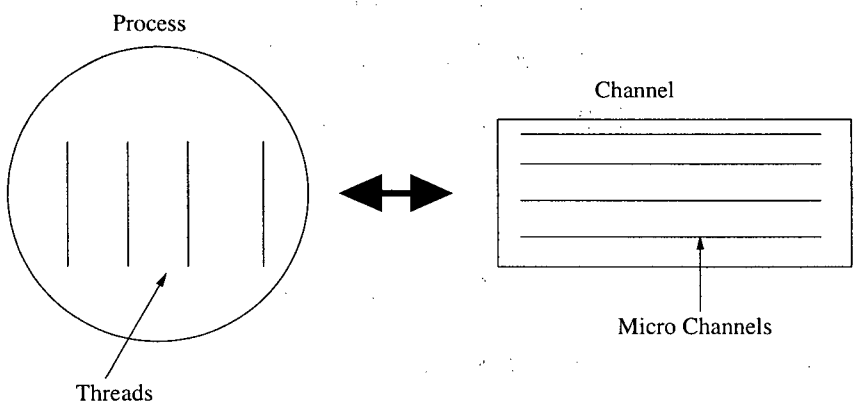

Figure 3.4: Comparison between channels and processes

- Like a process in an operating system, the MPI Channel represents the bulk state of the communication between two processes. In MPI_NP, the unit of scheduling inside the NIC was the channel. The channels in MPI_NP go through several states analogous to a process (ready, blocked etc).

- A microchannel is like a thread. It resides within a channel. The microchannel is also light weight. The setting up cost is minimal. The unit of scheduling on the local agent of MPI_NP II is the microchannel. It is analogous to an operating systems scheduling a thread instead of a process.

- In MPI_NP, a message can block the progress of another message within the same channel. This is comparable to a thread, within a process, that can block another thread, if the operating system schedules only the processes. Since MPI_NP II schedules microchannels (or messages), rather than a channel, one message cannot block the progress of another message. This is comparable to an operating system using a thread as a scheduling unit.

Setting up a channel

When the M4 system is started only the local agents can communicate with each other. Host processes cannot communicate over the Myrinet at this stage. The first step for a host process to establish communication with another process is to create a channel. The host process must ask the local agent to set up the channel. This is done by informing the local agent about the *context id*, *local rank*, *remote rank* and the *node* of the remote process. The local agent reserves a channel data structure for this channel and informs the agent on the remote NIC about this request. The channel data structure has a set of *send slots* and a set of *receive slots*. The number of send (or receive) slots is termed as the *channel width k*. This parameter represents the number of messages the channel can handle (in one direction). There is a one to one relationship between local send slots and the remote receive slots. Each send slot has space for the message envelope (note that small payloads are part of the envelope) and a precompiled header to point to its counterpart on the remote node.

This precompiled header includes a reference to its remote receive slot which allows for *remote writing* of envelopes to the remote NIC. When the local agent sends an envelope to the remote agent, the remote agent does not have to decide on where to put the envelope. The local agent specifies the remote location (i.e., location on the destination NIC). Routing information from the local node to the remote node is also precompiled and replicated over all the slots. However header information is incomplete until the remote rank also requests to create a channel. On request from the local agent, the remote agent simply checks whether it has already set up a state for a channel matching this request. If it can find one, then it will complete the header information on each slot and set the channel state to READY and inform the agent that sent the request, otherwise it drops the message. However, the channel will finally set up when the remote process too comes up with a request to create a channel with the local process. Once this three way handshake has occured the two processes are ready to communicate.

Our protocol is similar to MPI_NP, with one major difference, that the channel structure of MPI_NP did not have direct correspondence between send slots and receive slots. Precompiled header and routing information are also unique to this new design. MPI_NP channels were between two processes and the communication of multiple communicators between two processes takes place over the same channel. However in our design, even between the same two processes, different communicators must create different channels. This decision was taken in order to separate the communication between two communicators. This decision was crucial to our design, which gives a certain upper bound to the users about the look ahead (discussed later). Users know only about the messages generated by their applications and are generally unaware of the messages generated by the different libraries they

may use. An application library that a user may use must use a different communicator to separate its communication from the user communications. Otherwise non-deterministic behavior may happen [6]. Therefore, in our design, different channels are created by the MPI system for different communicators between the same two processes.

M4 is aware of the MPI communicators. When a channel is created it is added to the correct communicator on each local agent. This linked list of channels on a communicator is used to implement MPI broadcasting on the M4.

Communication on channels

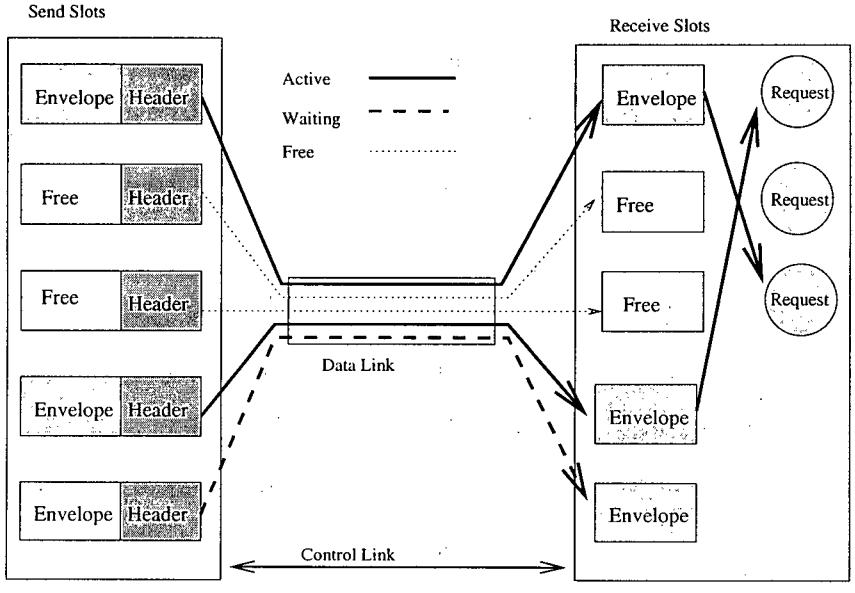

Figure 3.5: Message progress in a channel

Once the channels are set up, two ranked processes can communicate. The complete state of the channel is kept on the M4. Once the three way hand shake is completed, the M4 keeps the send slots synchronized with the corresponding receive

slots on the remote end. A send slot on the local send of the channels is a guarantee of a receive slot on the remote end of the channel. This ensures that once a send slot is allocated on the local agent, the communication never fails due to lack of envelope space on the remote agent.

The interface of the channel to the host application is extremely simple. The host process can send a message simply by acquiring a send slot on the channel (Figure 3.5). Acquiring a free slot is equivalent to setting up a microchannel. The host deposits the envelope and the address of the data buffer on the host into the free slot. The local agent *remote writes* this envelope to the receive slot on the remote NIC.

Communication over this microchannel is independent of communication over other microchannels in the same channel and the completion depends only on the posting of the receive request. The receiving process deposits the matching envelope and the address of the user buffer onto a free receive request. Note that receive requests are different from the receive slots. Receive slots are to receive the envelopes from the remote agent. Receive requests are to receive matching requests from the local process. These two rendezvous on the local agent and set up the path for the payload. Until this receive slot is matched with a receive request the microchannel is in the *waiting* state. Once the message is matched with a receive request the microchannel goes into the *active* state, if it is a large message. This involves informing the sending agent to send the payload. If it is a small message, the remote agent can simply upload the message with the envelope and free the receive slot and inform the local agent to free its send slot.

The local agents transfer the payload of matched large messages from the send user buffer to the receive user buffer asynchronously. The agents have the flex-

ibility in this scheme to schedule these active microchannels. This can be done in any order now, since the matched envelopes have already found the correct send/receive pair according to the ordering rule. Agents transfer the message using *cut-through delivery* [17] as a single worm from sender buffer to the receive buffer. The NIC's memory is used only as a staging area for the transfer.

Broadcasting

LAM/MPI implements broadcasting (`MPI_Bcast()`) using point to point routines. It uses two algorithms depending on the number of nodes participating for the broadcast. If the number of nodes (n) is less than or equal to 4, then a linear broadcasting algorithm is used; the root node sends n-1 point to point messages. If the number of nodes is larger than 4, then an $O(\log n)$ broadcasting algorithm is used where a broadcasting tree is constructed and the root node sends the message, over point to point routines, to $\lceil \log_2(n) \rceil$ nodes. In both cases the host must copy the **same data** multiple times to the NIC memory to send them to several nodes over the point to point routines. An obvious improvement is to download the message once and to have the NIC send the message to multiple nodes.

M4 implements broadcasting on the Myrinet. The current implementation only supports small message broadcasting on the Myrinet. However any size message can be broadcasted without using this special scheme. The broadcasting scheme implemented on Myrinet is a simple linear broadcasting (without the use of a broadcasting/multicasting tree). We decided to implement the linear broadcasting scheme because it can be implemented without changing the send/receive protocol.

On the root node specified on the `MPI_Bcast()` call, the host acquires free send slots on all the channels to the processes on the given communicator. Then it

copies the envelope and the payload only to **one slot** (slot on the *primary channel* of the communicator) and posts a *broadcast request* to the local agent. Note that the envelope has a special broadcast tag. The `MPI_Bcast()` call can return immediately after this. On the other nodes, `MPI_Bcast()` translates to a `MPI_Recv()` for that special broadcasting tag. Local agent of the root node traverse through the channel list on the communicator and sends the message on all these channels. For non-root nodes these messages are normal messages and they will be delivered to the matching receive request (the request with the special broadcast tag).

We did not change the simple point to point communication on the NIC to implement the broadcasting. Only the local agent of the root node knows that a message is a broadcasting message. This method allows M4 to broadcast the message asynchronously without any heavy overhead on the NIC protocol. We achieved this by having the knowledge about the *communicator* on the M4. Implementing broadcasting on M4 for large messages or creating a broadcasting tree requires changing the simple protocol. Because of this we decided not to implement M4 broadcasting for large messages and a tree based broadcasting algorithms in the current version. Large messages uses the normal LAM method of broadcasting using point to point communication routines (over Myrinet).

3.4.2 k_safe programs

The channel width k, represents the number of microchannels on a channel. A channel cannot hold more than k waiting microchannels. If a channel has at least one active or free slot, then the process can communicate over this channel further. If it has a free slot, it can communicate immediately. If it only has active slots, then the process can communicate in the near future, since the matched messages are

guaranteed to complete within a finite amount of time. Once an active microchannel is completed, it releases the resources (slots), allowing the process to use it for sending messages.

A complicated situation arises if all the send slots are in the waiting state. Two situations can arise here:

- On the receive side, the request slots also are full. This represents a case where no further progress can be made. Since the receive host cannot post more requests, the channel will be deadlocked.
- Receiver has more request slots. This represents only the timing difference between the two processes (i.e., the receiver is slow in posting the receive requests). Matching receive requests *may* post in future. The channel temporarily cannot accommodate more send envelopes.

We define a *k_safe* program to be a program that avoids the former situation on all the channels. This is defined as follows.

Let **S** be the ordered sequence of all sends on a channel and let **R** be the ordered sequence of all receives posted at the other end of the channel.

$$S = \{s_1, s_2, \ldots s_n\}$$

$$R = \{r_1, r_2, \ldots r_n\}$$

Let $S_j \subseteq S$ and $R_j \subseteq R$ and $j \leq n$ such that

$$S_j = \{s_1, s_2, \ldots s_j\}$$

$$R_j = \{r_1, r_2, \ldots r_j\}$$

Then if S_j and R_j have at most $k-1$ unmatched sends or receives for all $1 \leq j \leq n$ then the communication over that channel is *k_safe* for that direction. Note that if there is a blocking call (send or receive) in the sequence, then none of the calls after that in the sequence can participate in the matching process until the blocking call is matched. If this is true for all the channels (both directions) then the program is *k_safe*.

This definition has a practical implication. A sequence of k sends posted at the sending end may have a matching sequence of k receive requests at the receiving end, but there can be a *time delay* in the posting of those receive requests. In this case the implementation can either report an error to the next nonblocking send call or save the request on the host (to be posted later) and return without any error. The Guaranteed Envelope Resource scheme (discussed later) [11], takes the first option. However this can lead to a situation where an application completes without an error in one instance and the same application returning an error on another instance. The second option is more flexible and consistent. In this scheme a *k_safe* program always completes and a non *k_safe* program does not complete irrespective of the timing differences between the sending and receiving nodes.

Only a *k_safe* program can complete on our system. These are the programs that can complete with k send slots. Note that even $k = 2$ can give an infinite *look ahead*, i.e., a send posted at the beginning of the program can stay unmatched on the channel till the end of the process without blocking the channel, provided the other sends match immediately. Under this scheme, a few send slots allow programs with largely disordered matching send/receive pairs to complete.

We can compare this with two similar approaches. Burns et al. [11] define *Guaranteed Envelope Resources* (GER) as the number of guaranteed envelope spaces

on the remote process. In that approach a nonblocking standard send that would consume the last available remote envelope would be an error. The definition of *k_buffer correct* programs by Bruck et al [14], also deals with the correctness of a program in the face of limited remote resources. A *k_buffer correct* program is a one that completes without deadlocking in an environment with k system buffers per processor. *k_ buffer correctness* was defined in terms of a *global program* introduced in [15].

3.5 Host Interface Layer

The host interface layer provides the interface to M4. It interacts only with the local agent. The agent interface is initialized by mapping the shared structures on the agent to the host address space. Before any communication begins, the application must set up channels to the processes that it wants to communicate with. HIL has several communication functions that have a direct mapping to the blocking and nonblocking send/receives of MPI. However these communication routines expect a contiguous data buffer unlike the rich collection of data types in MPI. HIL communication routines also expect a channel number instead of ranks and context id's. The upper layer has to map the ranks to the channel number and to pack the data if necessary. Unlike M4 and the RPI layers, HIL does not interpret the envelopes. Its sole purpose is to provide the thin interface needed by the above RPI layers to M4. However HIL provides both blocking and non blocking send/receive routines and two routines to check the state of a non blocking communication. Upper layers can use either the blocking or nonblocking routines to map to the MPI routines. However blocking can be done on the RPI layer for blocking calls. Therefore we can use only the nonblocking HIL calls on the RPI layer.

HIL also provides the interface to the special broadcasting scheme implemented on M4.

3.6 Myrinet RPI layer for LAM/MPI

The MPI communication routines interface with the Myrinet specific communication routines through the RPI layer. The RPI layer uses the HIL functions to communicate over the Myrinet. As discussed earlier, the HIL is a thin layer. Since M4 itself takes care of the message matching, the RPI layer only has to inject the envelope into the channel.

On start up, the application calls MPI_Init(). As discussed earlier, the LAM specific part of `MPI_Init()` (`laminit()`) gathers link information sent by *mpirun* and creates the process table, then `laminit()` calls the RPI layer to perform RPI specific initialization. The correct RPI layer is selected by a flag given as an argument to `mpirun`. If the `myrinet` RPI was selected by the user, then the myrinet specific RPI functions are called. The function `_rpi_myr_init()` creates channels to all the processes of the application. Note that we inherited this function from MPI_NP, however we changed the functionality to adapt it to the new M4 and HIL. Unlike MPI_NP, which directly interacts with the Myrinet interface, we only have to call the HIL layer `channel()` routine for each process on the application to create the channels. The channel number for each process is recorded in the process table.

For all the communication routines of the application a *request* is created. As discussed earlier, these requests go through the stages of *building, starting, advancing and completion.* For each of these stages, an RPI layer specific routine is called to perform RPI specific functions for that stage. For the *Myrinet* RPI these functions are `_rpi_myr_build()`, `_rpi_myr_start()`, `_rpi_myr_advance()`

and _rpi_myr_destroy(). All these functions were inherited from MPI_NP, but they were changed to call the HIL functions. Only _rpi_myr_start() and _rpi_myr_advance() do useful work. The other two functions are there for compatibility and future expansions.

_rpi_myr_start() does the *myrinet* RPI specific initializations. We have created a function pointer on the C2C part of the request structure to store a pointer to the advancing function. _rpi_myr_start() sets this function pointer to the appropriate advancing functions depending on the type of the request.

_rpi_myr_advance() calls the advancing function. When the advancing function returns, it sets the function pointer to a function that can handle the next stage of the request progress. For a send/receive call Myrinet RPI has two advancing functions, one to post the request to the M4 and another functions to check for the completion of the request. Note that none of these functions are blocking. Upper layers decide the blocking/nonblocking nature of the call and call the RPI layer functions repeatedly until no more progress can be made, if it is a nonblocking call, or to the completion of a blocking call.

We bypass this mechanism and call the HIL functions directly for the MPI_Bcast() for small messages. LAM implements broadcast over point to point routines, but since we have a special broadcasting scheme for small messages, we bypass this mechanism and call the HIL routine to broadcast small messages.

Chapter 4

Evaluation

We tested the system on a cluster of 266MHz Pentium II PC's with 128MB of RAM, connected by a Myrinet network. The Myrinet NIC on our testbed had a 33MHz LANai 4.1 processor and 1MB of SRAM. The host operating system was Linux (kernel version 2.0.35).

We measured the overhead, latency and bandwidth for the point to point communication and the overhead at the root node for the `MPI_Bcast()`. Experiments were carried out for both MPI_NP II and MPICH-BIP for comparison. The version of the MPICH-BIP used was mpich-bip-0.99b (BIP version 0.99e). Latency and bandwidth were measured using the *echo test* [16] (same as the ping-pong test for end to end delay described in [28]). One node starts a timer and sends data to the receiving node. The receiving node simply echos back the data. Once the first node gets the reply, it stops the timer and records the *ping-pong* time. For each data size this test was repeated multiple times (256). Echo tests described in [16, 28] measure time outside the loop, however, we timed each send-receive pair and reported the median of the 256 measurements. This eliminates the extreme timing values caused by process switching and other traffic on the network. The timer used was the CPU

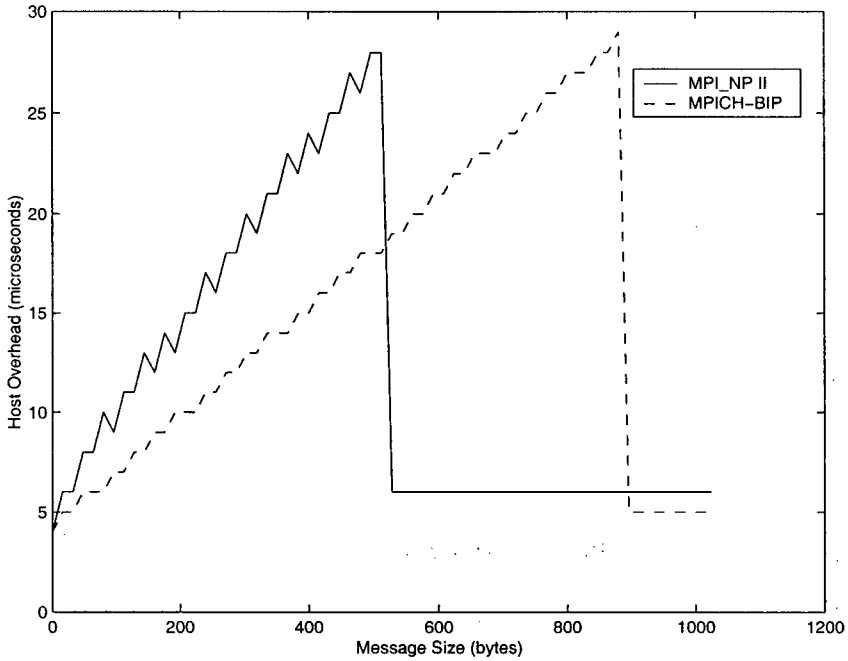

Figure 4.1: Host overhead for sending a message

clock cycles elapsed from the last reboot. This is same as the instruction used to get the CPU clock cycles in /usr/src/linux-2.0.35/arch/i386/kernel/time.c.

4.1 Host Overhead

Figure 4.1 shows the overhead incurred by the host in sending a message. This is the time for the nonblocking send (`MPI_Isend()`) to return after posting the send request. The minimum host overhead, for both MPI_NP II and MPICH-BIP, is about 4 microseconds to post a send request with a zero byte payload. For MPI_NP II this is the cost to acquire a free send slot on the channel and to copy the message envelope into the free slot. The MPI_NP II shows a steady increase in the host overhead with the message size, up to the message size of 512 bytes. MPICH-BIP

shows the same type of increase in host overhead up to the message size of about 900 bytes. This corresponds to the cost incurred by the host in copying (PIO) the message payload into the NIC. Both MPICH-BIP and MPI_NP II use PIO to copy small messages into the NIC. The relatively small rate of increase of host overhead for MPICH-BIP is due to the data copy mechanism used by MPICH-BIP. BIP uses the *write-combining* [27, 18] feature of Pentium II processor for efficient data copying to the NIC. The jagged line for the MPI_NP II shows the effect of the cache line. Data sizes of multiples of 32 bytes are written to the NIC more efficiently since the size of the cache line is 32 bytes. This effect of the cache line is not evident in the curve for the MPICH-BIP; again due to the *write-combining* technique of BIP.

The transition point from small message protocol to the large message protocol occurs at a message size of 512 bytes in MPI_NP II (900 bytes for MPICH-BIP). When sending a large message the sending host (MPI_NP II) acquires a send slot and copies the payload and pointer to the host data buffer into the free slot. This cost is constant irrespective of the message size and this is the reason for the constant host overhead for messages larger than 512 bytes for MPI_NP II. The results for the MPICH-BIP suggests that a similar scheme is used in MPICH-BIP. However as shown in [7], BIP cannot asynchronously transfer a nonblocking MPI message. The return time for `MPI_Isend()` for BIP, only indicates the time to initiate the message transfer. In contrast MPI_NP II transfers the message without any further intervention from the host. The only function of the subsequent `MPI_Wait()` call is to confirm whether the message is actually transfered. This is possible because the message matching and the deciding on the receive buffer are done by M4. Although the large message protocol reduces the host overhead, it increases the latency. Therefore it is necessary to balance the host overhead and the latency to

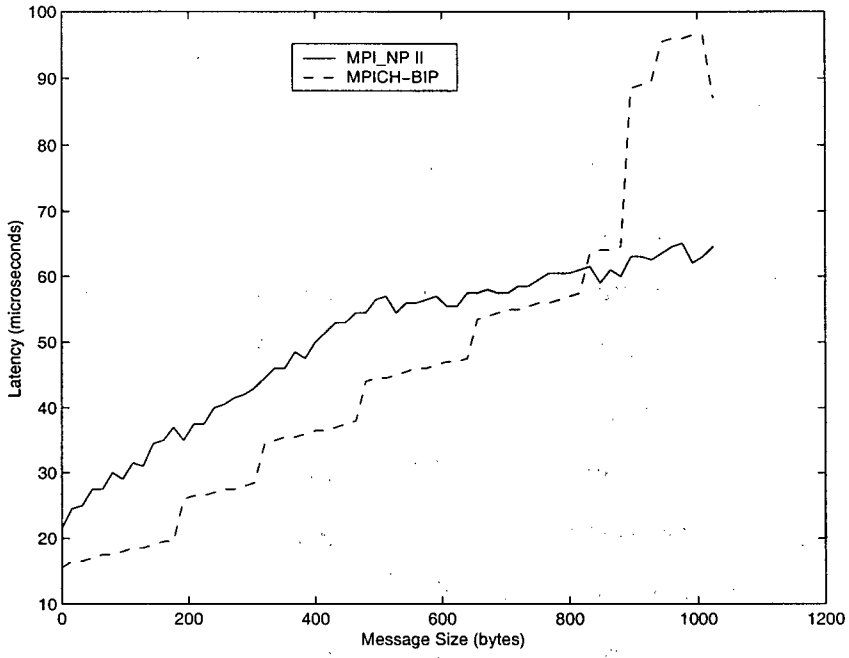

Figure 4.2: One way latency for small messages

provide better overall performance.

4.2 Latency (End to End Delay)

Figure 4.2 shows the one way message latency (end to end delay) for both MPI_NP II and MPICH-BIP. The minimum latency (for zero byte payload) for MPI_NP II is 22 microseconds and the same measurement for MPICH-BIP is about 15 microseconds. Figure 4.2 also shows that there is a smooth transition from small message protocol to the large message protocol in MPI_NP II indicating that our choice of 512 bytes as a transition point from small to large message protocol is a reasonable one. MPICH-BIP does not show such a smooth transition. There is a sudden jump in latency around 900 bytes and sudden drop around 1000 bytes. On browsing the BIP

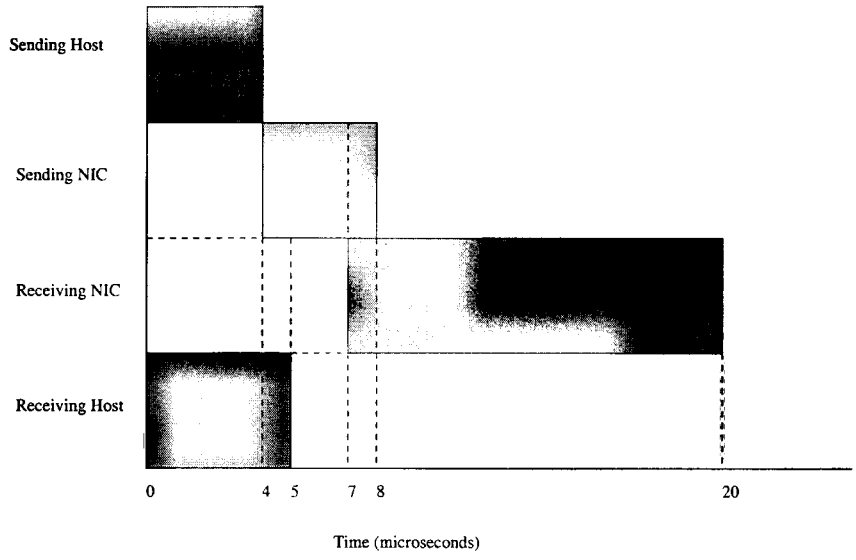

Figure 4.3: Breakdown of one way message latency

source code we found that the small message size is defined as 1016 bytes. However MPICH source code shows that maximum size for a data packet is 888 bytes. This may be the reason for the above behavior in MPICH-BIP latency. The steps in the MPICH-BIP latency is similar to the latency of MPICH on IBM SP2 presented in [13]. Authors attribute this to the fixed packetization cost of MPICH. However this does not seem to be the case since the jumps occur in the intervals of approximately 200 bytes whereas the packet size is 888 bytes.

A breakdown of one way message latency for a zero byte payload is shown in Figure 4.3. This breakdown is given for an expected message. For simplicity, we assumed the receive request was posted at the same time as the send request, however, the only requirement for the minimum latency is that the receive request is posted before the receiving NIC gets the message. One way message latency also includes a negligible cost for the receiving host to copy the envelope from the copy

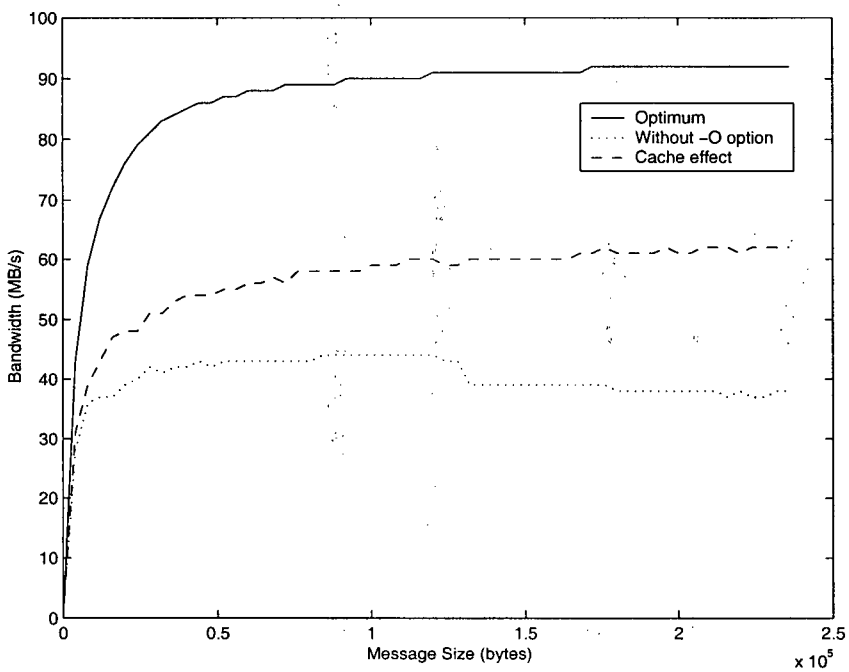

Figure 4.4: Bandwidth

block to the request structure. This cost is not shown in Figure 4.2.

The cost on the receiving NIC includes the cost of message matching which we found to be 4 - 5 microseconds. This is the best time since the message matches with the head of the expected message queue. Matching a message further down in the queue increases the matching cost. Note that matching occurs on envelopes and the cost is independent of the message size.

4.3 Bandwidth

Figure 4.4 shows the bandwidth of MPI_NP II under three conditions.

- Optimum: This is the maximum bandwidth achievable on our system. MPI_NP II reported a maximum bandwidth of 92MB/s. The sender sends data without modifying the buffer.

- Cache effect: The effect of modifying the data buffer before sending is shown in this curve. Since the data on the cache is changed, they must be written back to the main memory before the DMA can transfer the data.

- Without -O option: The -O option to the *mpirun* command specifies that all nodes are homogeneous. On homogeneous systems LAM/MPI does not pack simple data types and therefore avoids the extra copying necessary for packing the message. Figure 4.4 shows the effect of memory to memory data copy on the host needed for the message packing.

The maximum bandwidth of 92MB/s is an improvement on the maximum bandwidth of MPI_NP, which was 70MB/s. However, as shown in Figure 4.5, it is below the maximum bandwidth of MPICH-BIP. The MPICH-BIP recorded a maximum bandwidth of 105 MB/s.

4.4 Host Overhead for Broadcasting

We measured the time for `MPI_Bcast()` call to return on the root node for MPI_NP II with M4 broadcasting, MPI_NP II with broadcasting implemented over send/receive and MPICH-BIP. This time is the host overhead on the root node. Figure 4.6 shows the time for `MPI_Bcast()` call to return on the root node, when the number of nodes is 4.

The MPI_NP II broadcasting on M4 shows the best performance of the three curves. Broadcasting using this special scheme only involves copying the data once

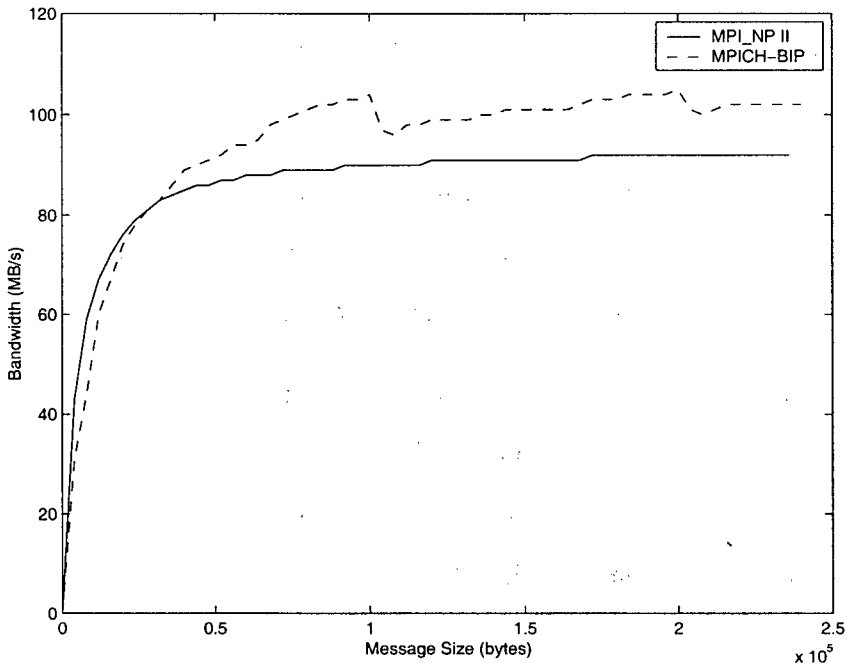

Figure 4.5: Comparison of MPICH-BIP and MPI_NP II bandwidth

to the NIC, where as the other two curves show the effect of copying the same data multiple times. Figures 4.2 and 4.1 showed better performance of MPICH-BIP over MPI_NP II. Therefore we expected MPICH-BIP to show lower host overhead than the MPI_NP II broadcast implemented on send/recv. However the Figure 4.6 shows that MPICH-BIP has higher overhead in broadcasting. We cannot explain this.

`MPI_Bcast()` returns immediately for a zero byte payload. This is the reason for zero overhead for zero length messages in all the three cases. Note that MPI standard (MPI 1.1) does not require MPI_Bcast to be a synchronizing call. Therefore a `MPI_Bcast()` call with a zero length buffer can return immediately on all the nodes.

LAM/MPI uses a linear broadcasting scheme (on top of the point to point routines), if the number of nodes is less than or equal to 4. Figure 4.7 shows

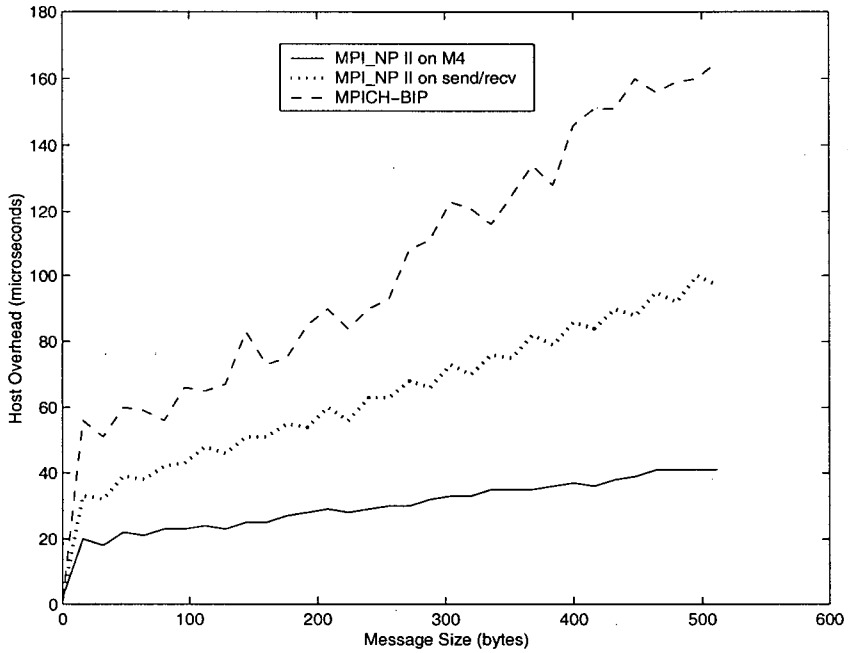

Figure 4.6: Host overhead for MPI_Bcast()

broadcast overhead on the root node for both the standard and M4 broadcast. The M4 broadcasting increases only by about 2 - 3 microseconds when the number of nodes increased from 3 to 4 and this gap is constant over the different data sizes. However the standard broadcasting shows a significant increase in host overhead from 3 node broadcasting to 4 node broadcasting. Furthermore this increase is proportional to the message size. The M4 broadcasting only has to acquire one more free slot when the number of nodes increased from 3 to 4. The standard broadcasting scheme has to acquire a new free slot and copy the data again to the new slot. This clearly shows the advantage of the M4 broadcasting system.

The Figure 4.8 shows that the host overhead on the root node for normal broadcast remains the same when the number of nodes increased from 4 to 8. This is

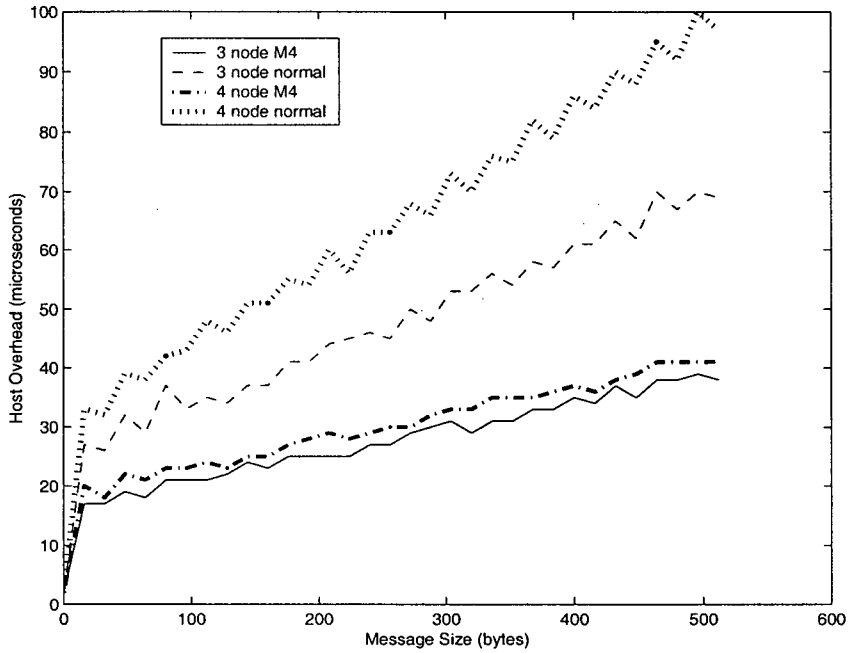

Figure 4.7: Increase of host overhead in MPI_Bcast with the number of nodes

because now the LAM/MPI broadcasting uses a broadcasting tree and the root node only sends data to three of the child nodes. The M4 broadcasting only implements a linear broadcasting and the host overhead increases with number of nodes. However this increase from 4 nodes to 8 nodes is only about 10 microseconds. The standard broadcasting scheme again shows another jump in host overhead when the number of nodes increased from 8 to 9. This indicates another level of the broadcasting tree. However the rate of increase of the host overhead with the number nodes is constant for the M4 broadcasting.

These results show the advantage of M4 broadcasting over the broadcasting implemented over point to point routines. The M4 scheme was possible because of the MPI awareness of the Myrinet control program (i.e., M4 knows about the

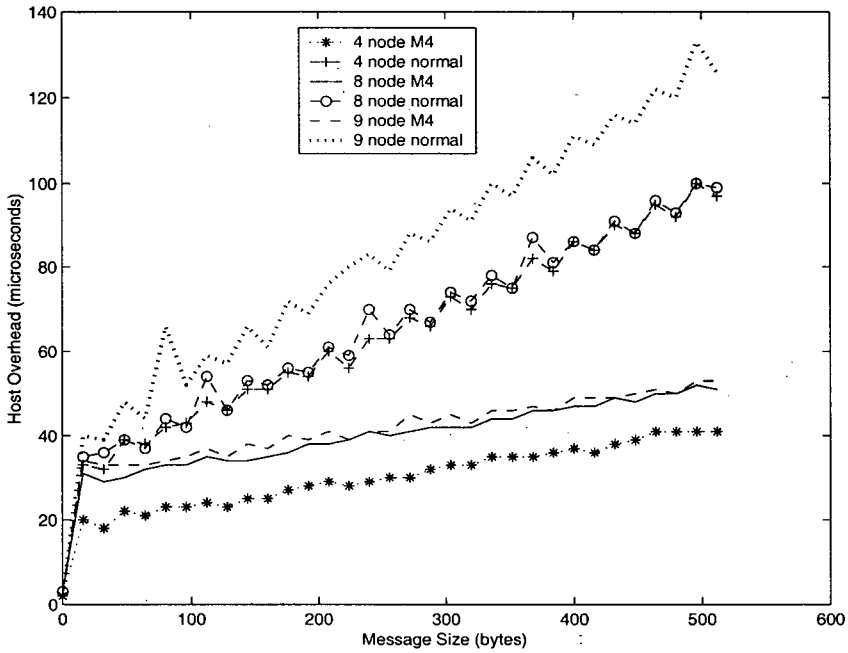

Figure 4.8: Increase of host overhead in MPI_Bcast with the number of nodes

communicator and the members of the group). This supports our claim that an MPI-aware program on the NIC can improve the performance, especially the host overhead.

4.5 Overlapping Computation and Communication

Two test programs were constructed to measure the extent to which it is possible to overlap computation and communication. One program does not take advantage of the overlapping (i.e., it does not do computation while the communication is in progress), while the other program does attempt to overlap the computation with communication. Algorithms for these two programs are shown in Figures 4.9 and 4.11.

```
main()
{

  /* MPI Initialization code. */
  if(rank == 0)
    {
      /* Start the timer here. */
      MPI_Isend(buf, msize, MPI_CHAR, 1, MY_TAG, MPI_COMM_WORLD,&req);
      compute();
      MPI_Wait(&req, &st);                                                    10
      MPI_Irecv(buf, msize, MPI_CHAR, 1, MY_TAG, MPI_COMM_WORLD, &req);
      compute();
      MPI_Wait(&req, &st);
      /* Stop the timer here. */
    }
  else
    {
      MPI_Irecv(buf, msize, MPI_CHAR, 0, MY_TAG, MPI_COMM_WORLD, &req);
      MPI_Wait(&req, &st);
      MPI_Isend(buf, msize, MPI_CHAR, 0, MY_TAG, MPI_COMM_WORLD,&req);        20
      MPI_Wait(&req, &st);
    }

  /* MPI Finalization code. */
}
```

Figure 4.9: Computation overlapped with communication (Case A)

For the program in Figure 4.9 (case A), the computation (`compute()`) is started after a nonblocking call and before calling `MPI_Wait()`. The `compute()` routine could be an arbitrary computation routine, which in our case it increments an integer while checking whether it is larger than a given number. In a system that progresses the nonblocking communication asynchronously this allows for overlapping of the computation with the communication. Timing diagrams for case A are shown in Figure 4.10 (diagrams are not drawn to a scale). It shows that in a system that allows asynchronous progress of messages, the `compute()` can overlap with the

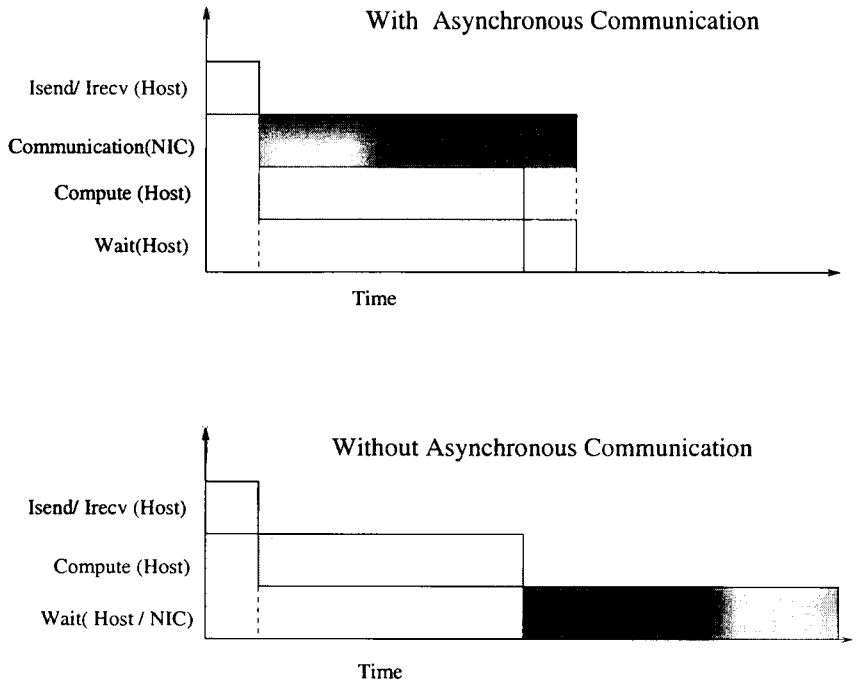

Figure 4.10: Timing diagram for Case A

progress of the message. The `MPI_Wait()` blocks until message transfer is completed. If the computation time is greater than the communication time, the `MPI_Wait()` returns immediately, since by the time it is called the communication has already completed. Figure 4.10 also shows the timing diagram for Case A on a system that does not progress messages asynchronously. In this case, `MPI_Isend()/MPI_Irecv()` initiate the communication, but the progress is made by the `MPI_Wait()` call. Therefore actual message transfer starts only after the completion of the the computation (when `MPI_Wait()` is called). `MPI_Wait()` does not return until the completion of the message transfer.

In Case B (Figure 4.11), the computation is started after the `MPI_Wait()` call. Since `MPI_Wait()` is a blocking call that returns only after the completion of

```
main()
{

  /* MPI Initialization code. */
  if(rank == 0)
    {
      /* Start the timer here. */
      MPI_Isend(buf, msize, MPI_CHAR, 1, MY_TAG, MPI_COMM_WORLD,&req);
      MPI_Wait(&req, &st);
      compute();                                                          10
      MPI_Irecv(buf, msize, MPI_CHAR, 1, MY_TAG, MPI_COMM_WORLD, &req);
      MPI_Wait(&req, &st);
      compute();
      /* Stop the timer here. */
    }
  else
    {
      MPI_Irecv(buf, msize, MPI_CHAR, 0, MY_TAG, MPI_COMM_WORLD, &req);
      MPI_Wait(&req, &st);
      MPI_Isend(buf, msize, MPI_CHAR, 0, MY_TAG, MPI_COMM_WORLD,&req);    20
      MPI_Wait(&req, &st);
    }

  /* MPI Finalization code. */
}
```

Figure 4.11: Computation is not overlapped with communication (Case B)

the communication, it does not overlap the computation with the communication. The timing diagram for Case B is given in Figure 4.12. Note that the total time in this case is identical to the time for Case A without asynchronous communication.

We tested these two programs on MPICH-BIP and MPI_NP II for different size data communications and with a fixed amount of computation. Results are shown in Figures 4.13 and 4.14. The communication time is proportional to the size of the data. The Time axes on these two figures give the time for the given code segment to finish on node 0. The code in Figures 4.9 and 4.11 shows where the

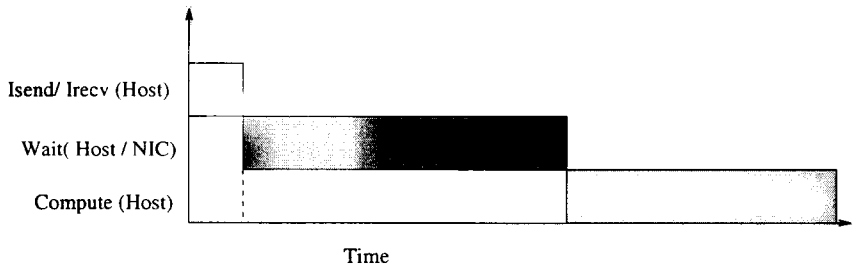

Figure 4.12: Timing diagram for Case B

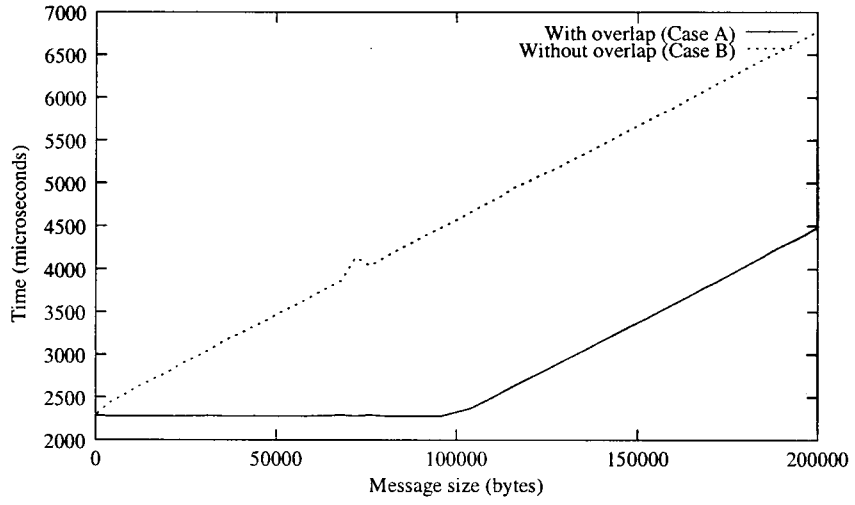

Figure 4.13: Effect of overlapping on execution time (MPI_NP II)

timer is started and stopped.

Figure 4.13 shows that the overlapped program (Case A) performs significantly better on MPI_NP II than the non overlapped program. The curve for Case A shows a constant time up to 100000 bytes of data. The reason is that the computation time is greater than the communication time, which increases with the data size. (Note that the computation time is constant irrespective of the size of data communicated). Therefore up to 100000 bytes the communication time is completely

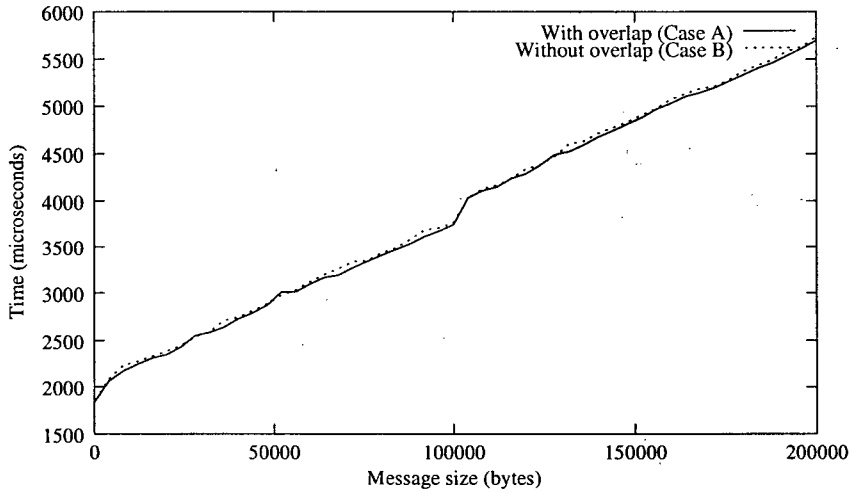

Figure 4.14: Effect of overlapping on execution time (MPICH-BIP)

masked by the computation time. (i.e., by the time `MPI_Wait()` is called, which is after the computation, the communication has finished and the `MPI_Wait()` can return immediately.) After 100000 bytes the dominant factor is the communication and we can see the total time increases. However it is far less than the time for Case B because part of that communication time is overlapped with the computation time.

Let T be the time for the program to finish and let $t_{compute}$ be the time for the computation and $t_{communicate}$ be the time for the communication. Note that the host overhead for `MPI_Isend/Irecv` is about 5 microseconds (Figure 4.1) and is negligible compared to $t_{compute}$ and $t_{communicate}$. The execution time for Case A is given as follows.

If $t_{communicate} \leq t_{compute}$ then $T = t_{compute}$, otherwise $T = t_{communicate}$.

For case B, $T = t_{communicate} + t_{compute}$, since we wait until the return of the `MPI_Wait()` to call the compute routine.

Figure 4.14 shows the results for the same two programs on MPICH-BIP. The two cases, Case A and B, do not show any significant difference in reported execution time. As discussed before Case A on a system without asynchronous communication is equivalent to Case B. This clearly indicates that in MPICH-BIP the communication is not progressed while the computation is taking place. For both Case A and Case B the total execution time (T) is the sum of $t_{communicate}$ and $t_{compute}$.

These results clearly show the advantage of using the NIC processor to progress communication asynchronously and strongly support our thesis. Although MPICH-BIP performed better in some micro benchmarks like latency and bandwidth, MPI_NP II has a clear advantage in applications that make use of nonblocking communications.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

We showed in this thesis that by using an MPI specific control program on the network processor, the performance of MPI applications can be improved. We designed and implemented MPI_NP II to support our thesis. The MPI_NP II was compared and contrasted with MPICH-BIP, which does not have an MPI specific messaging layer on the Myrinet.

The results from some of the micro bench marks showed that the performance of MPI_NP II is slightly less than the performance of MPICH-BIP. For example, the minimum one way message latency for MPICH-BIP is about 8 microseconds less than MPI_NP II and the bandwidth for MPICH-BIP is about 15 MB/s higher. However, the performance of `MPI_Bcast()` showed the advantage of having MPI specific control program on the NIC. MPI_NP II showed significantly low host overhead in broadcasting small messages. When broadcasting a 512 byte message on 4 nodes, the host overhead for MPI_NP II is only 40 microseconds while for MPICH-BIP it is 160 microseconds. This advantage far outweighs the disadvantages in latency and

bandwidth. This was possible because the control program on the NIC took over the responsibility of sending the message to multiple nodes.

The most important advantage of having an MPI specific control program on the NIC is that it allows for overlapping of computation and communication. We have shown that MPI_NP II can significantly improve the performance of MPI applications that make use of nonblocking communication. The advantage of performing computation while the communication is progressed asynchronously outweighs the slightly larger message latency and bandwidth. We have shown a program that can complete in 4500 microseconds on our system while taking 5700 microseconds on MPICH-BIP. In doing so we have shown that MPI_NP II can progress messages asynchronously while MPICH-BIP needs host intervention. This is possible because the control program on the NIC is aware of the MPI.

One of the reasons cited as a stumbling block to migrate MPI functionality onto the network processor is the relatively slow processor. However we identified that the key MPI task to be implemented on the NIC, to allow for asynchronous progress of messages, is message matching. Note that message matching is not a computationally intensive task. We managed to implement some of the MPI functionality on the NIC, to allow asynchronous progress of messages, without unduly burdening the slow processor.

We implemented the *MPI Channels* on the NIC and the MPI message matching semantics are handled inside the *channels*. By migrating message matching onto the NIC, MPI_NP II provided a simpler interface to the host.

In conclusion, we have shown that an MPI-aware program on the NIC can reduce the host overhead, simplify the interface of the low level communication system to the MPI library, allow for overlapping of computation and communication

and can provide an efficient group communication mechanism.

5.2 Future Work

The current implementation of MPI_NP II is by no means a complete implementation. One of the major limitations in this implementation is that messages can be sent only from a pre-pinned memory area. A side effect of this limitation is that MPI_NP II currently cannot send or receive complex data types. Complex data types have to be packed into a separate buffer before sending. Since this buffer must be pinned and registered with the NIC before sending, the current implementation cannot handle it. The NIC itself can be used to solve the problem of non contiguous data types. Rather than packing data into a system buffer, there is a possibility to use the NIC to DMA the data from different parts of the host memory. This solution also allows for zero copy message transfer for non-contiguous data types, which is important considering that the network bandwidth is comparable to the memory bandwidth.

The current implementation cannot handle ANY_SOURCE message matching (MPI_NP II can handle ANY_TAG messages). An obvious solution is to add the receive request to all the channels in the communicator. However this solution does not scale well and a more elegant solution is needed. A solution we suggest is to use a separate ANY_SOURCE request queue for each communicator on the NIC. When a message is received it must be compared with the request list on the channel and with the ANY_SOURCE request list. If matches are found on both lists, the conflict must be solved without violating the MPI ordering rule. This can be done by time stamping the receive requests.

Only the communication on MPI_COMM_WORLD is possible in the current

implementation. We did not provide for creation of new communicators after the initialization time. A complete implementation must provide for creation of new communicators and communication over those communicators.

Currently MPI_NP II can handle only 32 channels, each with 8 microchannels. Static allocation of memory for channels and the limited NIC memory led to this limitation. We took this design decision to relieve the NIC from memory management tasks and to keep the MCP code simple. Chun et al. [2] discuss the idea of using the NIC memory as a cache for active communication *end points* (comparable to channels). Use of the NIC memory as a cache for the active channels provides for relatively large number of channels. However performance can degrade due to channel swapping.

The broadcasting scheme implemented on the NIC can handle only small messages (less than 512 bytes). Furthermore, messages are broadcasted without constructing any broadcasting tree (linear broadcast). We restricted the broadcasting scheme in order to simplify the protocol between the NIC's. However, this solution may not scale well for a large number of nodes. The protocol between the NIC's has to be changed to allow for large message broadcasting and to make use of an efficient broadcast tree.

By implementing this simple broadcasting scheme on the NIC we have set the stage for implementing other collective communication routines. The possibility of implementing *gather* and *scatter* operations is promising since, as discussed before, the NIC can gather data from different parts of the host memory and can send them to other nodes. On receiving the data, the NIC can also scatter them to different host memory locations by means of the DMA engines. However, it is doubtful whether it is efficient to implement the *reduce* operation on the NIC, since *reduce*

may involve heavy computation. The *reduce* function takes an arbitrary function as input to reduce the data from each process. Performing reduce on the NIC would require support for executing an arbitrary function which is not advisable given the memory and the speed of the NIC processor.

Bibliography

[1] A. Wijeyeratnam and A. Wagner. MPI-NP: A Myrinet communication layer for LAM. In *Proc. of the 11th Parallel and Distributed Computing and Systems*, Cambridge, Massachusetts, November 1999.

[2] B. Chun, A. Mainwaring and D.E. Culler. Virtual Transport Protocols for Myrinet. In *Hot Interconnects V*, Stanford, CA, August 1997.

[3] C.A.R Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666 – 677, August 1978.

[4] Thomas Stricker Christian Kurmann. A Comparison of Three Gigabit Technologies: SCI, Myrinet and SGI/Cray T3D. In Alexander Reinefeld Herman Hellwagner, editor, *SCI: Scalable Coherent Interface*, volume 1734 of *Lecture Notes in Computer Science*, pages 39 – 68, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999. Springer-Verlag.

[5] Cornell University and The Ohio State University. Trollius Reference Manual for C Programmers. http://sunsite.org.uk/computing/operating-systems/trollius/, March 1990.

[6] D. W. Walker. An Introduction to Message Passing Paradigms. In C. E. Vandoni, editor, *Proceedings of the 1995 CERN School of Computing, CERN 95-05*, pages 165–184, Arles, France, August 1995.

[7] Frederique Chaussumier, Frederic Desprez, Loic Prylli. Asynchronous Communication in MPI - The BIP/Myrinet Approach. In Jack Dongarra and Emilio Luque and Tomas Margalef, editor, *Recent advances in parallel virtual machine and message passing interface: 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26–29, 1999: proceedings*, volume 1697 of *Lecture Notes in Computer Science*, pages 485–492, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999. Springer-Verlag.

[8] G. Burns. A Local Area Multicomputer. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[9] G. Burns, R. Daoud and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94*, Toronto, Canada, June 1994.

[10] G. Burns, V. Dixit, R. Daoud and R. Machiraju. All About Trollius. Occam Users Group Newsletter, August 1990.

[11] Greg Burns, Raja Daoud . Robust MPI Message Delivery with Guaranteed Resources. MPI Developers Conference at the University of Notre Dame, June 1995.

[12] H. Tezuka, F. O'Carroll and A. Hori. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, Orlando, USA, March 1998.

[13] Hubertus Franke, C. Eric Wu, Michel Riviere, Pratap Pattanik, Marc Snir. MPI Programming Environment for IBM SP1/SP2. In *Proc. of the 15th International Conference on Distributed Computing Systems*, pages 127 – 135, Vancouver, Canada, May 1995.

[14] J Bruck, D. Dolev, C. Ho, M. Rosu and R. Strong. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 64 – 73, Santa Barbara, California, July 1995.

[15] Jehoshua Bruck , Danny Dolev, Ching-Tien Ho, Rimon Orni, Ray Strong . PCODE: An Efficient and Reliable Collective Communication Protocol for Unreliable Broadcasting Domains. In *IEEE International Parallel Processing Symposium*, Santa Barbara, California, April 1995.

[16] J.J Dongarra, T. Dunigan. Message-Passing Performance of Various Computers. Technical Report CS-95-299, University of Tennessee, May 1996.

[17] K.G. Yocum, J.S. Chase, A.J. Gallatin and A.R. Lebeck. Cut-Through Delivery in Trapeze: An Exercise in Low-Latency Messaging. In *IEEE Symposium on High-Performance Distributed Computing (HPDC)*, Portland, OR, August 1997.

[18] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *Workshop, PC-NOW, 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, Orlando, USA, March 1998.

[19] LAM Team. Porting the LAM 6.3 communication layer. Technical Report TR 00-01, University of Notre Dame, August 1999.

[20] M. Lauria. High Performance MPI Implementation on a Network of Workstations. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1996.

[21] Loic Prylli, Bernard Tourancheau, and Roland Westrelin. The design for a high performance MPI implementation on the Myrinet network. In *Proc. of EuroPVM/MPI'99*, Barcelone, Spain, September 1999.

[22] M. Lauria and A.A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4 – 18, January 1997.

[23] M. Lauria, S. Pakin and A.A. Chien. Efficient Layering for High Speed Communication: Fast Messages 2.x. In *Proc. of the 7th High Performance Distributed Computing Conference*, Chicago, Illinois, July 1998.

[24] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html, June 1995.

[25] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, July 1997.

[26] Michael S. Warren, Donald J. Becker, M. Patrick Goda, John K. Salmon, Thomas Sterling. Parallel Supercomputing with Commodity Components. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1997.

[27] MindShare Inc. Tom Shanley. *Pentium Pro and Pentium II System Architecture.* Addison-Wesley, Reading, Massachusetts, 1998.

[28] N. Nupairoj and L. Ni. Performance Evaluation of some MPI Implementations on Workstation Clusters. Technical report, Department of Computer Science, Michigan State University, October 1994.

[29] Margaret A.S Petrus. Service Migration in a Gigabit Network. Master's thesis, University of British Columbia, Vancouver, Canada, 1998.

[30] R. Bhoedjang, T. Ruhl and H.E. Bal. Design Issues for User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53 – 60, November 1998.

[31] R. Hempel and D. W. Walker. The Emergence of the MPI Message Passing Standard for Parallel Computing. *Computer Standards and Interfaces*, 21(1):51–62, May 1999.

[32] S. Pakin, V. Karamcheti and A.A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2):60 – 73, June 1997.

[33] Scott Pakin, Mario Lauria, Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. of the Supercomputing'95*, New Orleans, December 1995.

[34] Jeffrey M. Squyres, Andrew Lumsdaine, William L. George, John G. Hagedorn, and Judith E. Devaney. The interoperable message passing interface (IMPI) extensions to LAM/MPI. In *Proceedings, MPIDC'2000*, March 2000.

[35] T. Eicken, D. Culler, S. Goldstein and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Annual Symposium on Computer Architecture*, pages 256 – 266, Gold Coast, Australia, May 1992.

[36] T. Sterling and D. Savarese and D. J. Becker and J. E. Dorband and U. A. Ranawake and C. V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, August 1995.

[37] T.E. Anderson, D.E. Culler, D.A. Patterson and the NOW Team. A Case for Networks of Workstations: NOW. *IEEE Micro*, February 1995.

[38] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency Practice and Experience*, 2(4):315 – 339, December 1990.

[39] W. Gropp and E. Lusk. An abstract device definition to support the Implementation of a High-Level Point-to-Point Message-Passing Interface. Technical Report Preprint MCS-P392-1193, Argonne National Laboratory, March 1995.

[40] W. Gropp, E. Lusk, N. Doss and A. Skjellum. High-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[41] A. Wijeyeratnam. An MPI Messaging Layer for Network Processors. Master's thesis, University of British Columbia, Vancouver, Canada, 1999.