An MPI Messaging Layer for Network Processors

by

Ashley Wijeyeratnam

B.Sc., The University of Colombo, Sri Lanka, 1996

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

October

1999

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 OCTOBER 15, 1999

DE-6 (2/88)

Abstract

We describe the design and implementation of MPI-NP, a Myrinet communication system tailored to support LAM, a public domain version of MPI. The goals of MPI-NP are to reduce the time spent on the host for message processing, and to promote the overlap of computation and communication. MPI-NP achieves these goals by off-loading as much of the computation as possible to the network processor. MPI-NP relieves the host of several tasks, making more CPU cycles available to the application, but pays a price for heavy utilization of the slower network processor, by adding a significant overhead to message latency. Although part of the overhead can be attributed to the hardware of our testbed, the philosophy of MPI-NP characteristically does not provide the best latency possible because of performance disparities between host and network processors. Users are left with the choice of deciding on the trade-off of latency versus host overhead. Applications which are not latency bound can expect to perform well on MPI-NP.

Contents

Abstract ii

Contents iii

Acknowledgements v

1 Introduction 1

1.1 Overview 1

1.2 Motivation 2

1.3 Thesis Statement 4

1.4 Methodology 4

1.5 Synopsis 5

2 Background 6

2.1 Parallel Computing and Metacomputing 6

2.2 The Message Passing Interface 7

2.2.1 LAM 9

2.2.2 MPICH 10

2.3 Gigabit Networking 11

2.3.1 Existing Technologies 11

2.3.2 System Architecture 13

2.3.3 Performance Issues 14
2.4 Related Work 16
2.4.1 Selected Implementations of MPI 16
2.4.2 Messaging Systems on the Myrinet 18

3 The Design of MPI-NP **23**
3.1 Channels 24
3.2 Flow control 27
3.3 Message protocols 28
3.3.1 Full Credit 28
3.3.2 Message Rendezvous 29
3.3.3 Eager send of small messages 30
3.4 Message matching 31
3.5 Supporting Zero Copy 34
3.6 Hardware issues 35
3.7 Architectural Overview 37

4 Evaluation **41**
4.1 Bandwidth 42
4.2 Host Overhead 43
4.3 Latency for Small Messages 44
4.4 Discussion 47

5 Conclusions **48**
5.1 MPI at the Network Layer 48
5.2 Future Work 50

Bibliography **53**

Acknowledgements

I am greatly indebted to my supervisor, Professor Alan Wagner, for inspiring and guiding me through the two years I spent at UBC. I am grateful to Professor Mike Feeley for the many general and project specific discussions we had, that influenced my way of thinking, and to Professor Norm Hutchinson, who was always ready to push his sleeves up at the hint of a problem. Thanks also to the folk in the Distributed Systems lab for the wonderful times we had.

I am especially grateful to the occupants of the Motha residence who put an extra effort to make Vancouver feel like home to me. Chamath Keppitiyagama was a tower of strength, challenging every point, supporting every move I made. Finally, I wish to thank my family, who's continued support is immeasurable.

ASHLEY WIJEYERATNAM

The University of British Columbia

October

1999

Chapter 1

Introduction

1.1 Overview

Clusters of commodity processors connected by high speed networks have become an attractive platform for high performance parallel computing. In the last few years there has been a dramatic increase in processor speed, but, more importantly for clusters, there has been an even more dramatic improvement in network performance. Networks with 50MHz workstations on 10 Megabit per second ethernet of a few years ago are now being replaced with 650MHz PCs on Gigabit per second networks. The gap between processor, memory and network performance has narrowed to the extent that cluster computing is now viewed as a viable platform for high performance computing [Bake99].

Delivering the performance of high speed networks to the application remains a challenge. Software overhead for communication is a significant bottleneck in the performance[Kara94]. Much of the overhead comes from processing protocol layers, copying data between buffers and making transitions between user space and kernel space. Typically, the application makes the data location known to the kernel which copies the data into a system buffer and proceeds to write it into the network interface which sends it across the wire. One way of reducing software overhead is to make the network interface (NI) accessible at the user level so that the application could transfer data

directly to the NI, avoiding the overhead of going through the kernel.

A new generation of network interface cards (NICs) provide solutions to these issues by being highly reliable and accessible at the user level. These networks, called System Area Networks, span a small area of about 25 to 50 square feet. Their limited span allows data transmission to be almost error free eliminating the need for heavyweight reliability protocols.

The most interesting feature of newer NICs is an embedded programmable processor. The advantage of a programmable NIC is it makes it possible to tailor the communication interface to a particular application to improve performance by replacing all purpose protocol stacks with a specific, thin and simple user-level interface.

1.2 Motivation

Programmable interface cards, most notably Myrinet[Bode95], have been used in a variety of projects. The focus of these projects have ranged from simple communication interfaces [Pryl98, Chun97, Laur97] to issues of security and protection [Buzz96, Dubn97a] to the support of specific applications [Coad99, Bhoe98]. The common goal of these interfaces is to provide a lean, general purpose interface to give good bandwidth and latency over a range of message sizes. In some cases these goals are accomplished by restricting the flexibility of the interface and, in most cases, by making minimal use of the network processor (NP). The justification given[Iann98] for minimizing the amount of message processing in the NIC is that the NP runs too slowly to handle messages in a timely manner. It has been found that processing messages on the host gives the best performance in terms of message latency and bandwidth.

Almost all of these systems are designed specifically for the Myrinet which, introduces difficulties that are specific to the Myrinet hardware and need not be the common case as we show in chapter 4. Comparably cheap embedded processors at about 10% the cost of host processors with clock speeds of more than 38%[Micr99, Dief99] of the host are currently available and the trend signals a faster growth of embedded processor performance. With a better designed NIC and a

faster processor running at speeds closer to the host CPU, entrusting more tasks to the NP becomes feasible. The focus of this thesis is to investigate network support for MPI on the assumption that the network processor is capable of efficiently handling moderate workloads in comparison to the host processor.

There are several advantages to off-loading computation to the NP and tightly integrating the interface to a specialized message passing system like MPI.

- Message passing alone involves a non-trivial amount of computation. Since the target applications for a high speed message passing system are often computationally intensive parallel applications, it is desirable that the host CPU devote as little time for message passing as possible. Off-loading message processing to the NP makes additional CPU cycles available to the application.

- The host could add several messages into the communication pipeline without waiting for them to make progress because message progress would, to a large extent, be handled by the NP.

- There are performance advantages for protocols required by special MPI communication routines like `Synchronous Send` and `Ready Send` that can be handled more efficiently between two NICs rather than at the higher level between two hosts.

- Current communication systems implement collective communication like broadcast, gather and scatter on top of point-to-point routines. It is more efficient to implement collective routines on the NIC rather than on the host because doing so would require only a single interaction per routine between host and NIC.

Host overhead is a cost incurred on every communication instance and is a parameter that directly determines the granularity of parallelism that can be effectively exploited by a distributed application. It is an issue that, along with heavier NP utilization, has so far been unconsidered.

1.3 Thesis Statement

Utilizing embedded network processors to handle messages of a message passing application would reduce host overhead on a high speed communication system thereby making more host CPU cycles available to the application for computation while the network processor manages communication.

1.4 Methodology

We have taken this alternative approach in the development of MPI-NP (**M**essage **P**assing **I**nterface on the **N**etwork **P**rocessor) to test the feasibility of our objectives. In designing MPI-NP we analyzed the communication layer of LAM, a popular public domain implementation of MPI written to work with TCP/IP, and determined the division of work between the host and the NI based on our objective of minimum host overhead. Functionality such as routing, message matching, buffer management, flow control and protocol processing was migrated down to the NIC.

In implementing MPI-NP we have made use of the optimizations present in existing high speed interfaces with the aim of achieving low latency and high throughput. The principles of MPI-NP are not specific to any type of hardware. The only assumptions we make are about capabilities found in current network interfaces, which are that the NI is accessible at user-level and that it has an embedded programmable processor and some memory which can be used as system buffers. The implementation was done on Myrinet NICs and has optimizations pertaining to the Myrinet hardware in order to optimize performance.

Our design relieves the host of several tasks but, in comparison to other message passing systems on similar platforms, performs poorly on message latency due to the fact that our testbed was unsuitable for heavy utilization of the network processor. The current state of the hardware indicates that NICs have made improvements in design and processor speeds compared to our test bed. MPI-NP would make bigger percentage improvements than existing systems if the performance of network processors improves with respect to the host CPU. Still, we expect the host processor to be the more powerful of the two and therefore it would be faster to perform certain tasks on the

host. Applications tightly bound by latency would perform well if the work on a message's critical path were executed in the fastest manner possible, which would be on the host. Other applications which are not latency critical would benefit greatly by the philosophy of MPI-NP.

1.5 Synopsis

In the following chapters we present the issues of designing a message passing system and explain the design decisions we made in implementing MPI-NP. Chapter 2 highlights the need for message passing systems and briefly describes the MPI standard and some implementations of it. It also includes an overview of networking technology and brief reviews of related communication systems for the Myrinet. Chapter 3 contains a discussion and justification of the overall design of MPI-NP. We evaluate the performance of MPI-NP in chapter 4 and finally present our conclusions in chapter 5 along with a description of further work needed for MPI-NP to be complete.

Chapter 2

Background

In this section we discuss the issues surrounding distributed computing in general and the message passing paradigm in particular. We review two existing public domain implementations of the Message Passing Interface (MPI) standard. We then discuss high performance networking technology and the new issues introduced by them. Finally we review methods employed, by existing message passing systems implemented on such networks, to address these issues.

2.1 Parallel Computing and Metacomputing

Distributed applications fall broadly into two categories.

- Applications that combine the processing power of the networked machines to enhance their performance thus turning the network into a single parallel processing machine
- Applications that cooperatively use resources, like storage and computing power, made available on the network that are not available otherwise.

The scientific and engineering computing community have found in workstation clusters a cheap and convenient mode of parallel processing. Massively parallel processors are fast losing their appeal. Contributing to this phenomenon are their prohibitive costs compared to declining

workstation costs, and the increase in performance of workstations and networking technologies. The price/performance ratio of workstations improve at 80% a year whereas that of supercomputers improve at about 20% to 30% a year[Ande95]. Networking technology has advanced over the last few years to provide bandwidths and point-to-point latencies such that the network is no longer the bottleneck in the performance of a distributed application. These combined advances have turned supercomputing into an affordable commodity[Bake99].

The widespread availability of workstations, supercomputers and storage devices on networks has prompted the development of metacomputing systems[Grim98] that bring together these diverse resources and present it to applications as one unified system. Such a system will consist of millions of hosts and other computing resources connected by high speed links and provide desktop users with supercomputing power. The vision of a nationwide metacomputer was presented several years ago[Smar92] and now projects like Globus[Fost97a], by providing tools to integrate applications, middleware and the network, and Legion[Grim97], by supporting distributed object oriented abstractions, work towards bringing that vision closer to reality.

2.2 The Message Passing Interface

A common characteristic of distributed and networked parallel applications is their method of communication. Data is transferred by passing messages between host processes. This holds true for scientific applications as well as for metacomputing experiments. Of the two parallel computing paradigms, namely shared memory and message passing, networked parallel computing, dictated by the distributed nature of the hardware, follows the latter. Message passing is the most common and well understood model of parallel computation. At the time the MPI standard[Mess95] was proposed, there existed many implementations of message passing libraries each with its own syntax, semantics, strengths and limitations. The standardization effort by the MPI Forum was an attempt at enabling applications written using these libraries to be portable. The standard itself only specifies the application programmer interface and the behavior of its features in an

implementation.

An MPI program is made up of several autonomous processes executing their own code in their own address spaces in an MIMD style. Processes communicate with each other by invoking MPI communication primitives. The number of processes are decided at startup time in MPI-1. Primitives for dynamic process creation and deletion are described in the MPI-2[Mess97] specification. Processes belong to a group that initially includes all processes of the application and are identified by their rank in the group. While they can form themselves into sub groups, a process is allowed to be a member of many groups simultaneously. Processes can also arrange themselves into graph or Cartesian topologies. Processes within a given group can communicate among themselves, or a process of one group can communicate with any process of another group. The former is known as intra group communication while the latter is called inter group communication. Objects called communicators[Fost96] define communication spaces.

MPI has a rich collection of point-to-point communication functions: buffered, synchronous, ready and standard. Each of these modes can be blocking or non-blocking. Each MPI message carries, in addition to its data, an *envelope* containing the source, destination, tag, and context ID of the message. Messages are received (matched) based on the contents of the message envelope. Applications can use wildcards, `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, to receive a message from any source and/or with any tag. MPI also provides for one-sided communication routines such as put and get, and collective communication routines such as broadcast, scatter and gather operations which serve the typical communication needs of most parallel computing applications.

A key property of MPI messages is that they are non-overtaking. The message ordering rule states that messages sent from one process to another are made available to the receiver in the order in which they are sent. The receiver may choose to pick the second message over the first, if they can be distinguished. If two receive operations match a message and they are both still pending, the message is delivered to the first.

The notion of a message taken by most message passing libraries is a contiguous data buffer. Whenever non contiguous data of possibly different types are to be sent, they are packed into a

contiguous buffer by the sender and unpacked by the receiver. This is an expensive operation requiring too many copy operations by both processes. MPI provides a facility to define data types that are non contiguous and where the data elements differ in their fundamental types.

Several implementations of MPI customized for various environments are available in the public domain. Two of the more popular versions are LAM[Burn94, Burn89] developed at the Ohio Supercomputer Center and MPICH[Grop96] developed by Argonne National Laboratory and Mississippi State University. Both of these implementations are based on precursor systems. The following sections give brief overviews of these systems.

2.2.1 LAM

The Local Area Multicomputer (LAM) is a subset of an operating environment called Trollius[Burn90], originally developed for message passing on transputer nodes.

The core of LAM is a multi tasking micro kernel that runs as one UNIX daemon per host. MPI messages can be sent either via the LAM daemon or directly to the receiving process in client-to-client mode. The daemon is a legacy of Trollius and is useful during the application development stage, providing facilities for debugging and monitoring messages. A LAM based MPI application, even if started up in client-to-client mode, still needs the daemon to help setup the initial connections. The processes once connected form a fully interconnected topology. Application specific virtual topologies are built on top of this.

The upper layers of LAM use a Request Progression Interface (RPI) to monitor and manage messages. When a process needs to send or receive a message, it does so as a request for service. Requests are maintained in a queue and can be in one of four states, INIT, START, ACTIVE and DONE. The RPI layer reads and writes data into TCP sockets in non blocking mode, trying to transfer as much data as possible from one request before servicing another.

If the message content is fragmented in the application, the data is gathered and packed into a single buffer. Certain data types require byte order conversions to be performed on them before transmission. The message is wrapped in an envelope containing addressing and identification

attributes and placed in a request. The request is then added onto the request progression queue to be serviced.

2.2.2 MPICH

The design of MPICH is based on mainly three libraries, p4[Butl94], Chameleon and Zipcode[Skje94]. The p4 library supports multiple models of parallel computation. Programmers can use monitors to coordinate access to shared data in a shared memory model. It contains message passing functions and global operations that can be used in a distributed memory model. In addition p4 routines can be used to manage collections of processes. The library is portable to many different types of parallel machines, workstations and environments.

Zipcode is a message passing system designed to support parallel libraries and large scale multicomputer software. Many of its features were adapted into the MPI standard specification as well. The main contribution of Zipcode was the notion of a "context". It uses process groups to limit the scope of message passing activities, defines separate communication contexts to enable library development and allows different notations of process naming to support virtual topologies.

MPICH is a library whose routines form a layered hierarchy. The portable two upper layers interact with the device dependent lower layer through an Abstract Device Interface (ADI)[Grop95]. The ADI provides four kinds of services, namely specifying messages to be sent or received, moving data between the application and the hardware, managing pending messages and providing information about the environment. The lowest layer is an interface to the device which does all of the work. The interface has been written using the chameleon macros and, for its UNIX workstation implementation, the device it connects to is p4. Since p4 is a message passing library by itself, all layers on top are simply interfaces to it.

The major concern of MPICH is portability. This emphasis causes it to perform a number of operations that do not take place in LAM. Some of these additional operations are listed below.

- Checks arguments to function calls for validity. Datatypes have a special 'cookie' in them to

hold their integrity.

- Keeps pointers to datatypes and opaque MPI objects (e.g. communicator) in an array and performs lookup based on implementation platform.
- Keeps communication devices in a table and perform lookup every time a message is sent.
- Copies message from user buffer to a packet.
- Adds an eXternal Data Representation (XDR) wrapper
- Converts all data from host to network byte format

These additional operations take its toll on the performance of MPICH compared to LAM[Nupa94]. Yet because of the ease of replacing the ADI, MPICH is a popular choice among MPI implementors. Most new implementations of MPI are simply clones of MPICH with a custom built ADI.

2.3 Gigabit Networking

The advances in network performance is another key factor in the popularity of cluster supercomputing. The last few years have seen LANs supporting increasing bandwidths of up to a Gigabit per second and decreasing point to point latencies of less than 5 microseconds. In this section we review two of these technologies and discuss their implications on systems development.

2.3.1 Existing Technologies

Gigabit Ethernet

Ethernet is the standard and most widely used communication medium in LANs. High speed networks were built on Fast Ethernet or 100BASE-T until the recent emergence of Gigabit Ethernet[Giga99]. Gigabit Ethernet uses the CSMA/CD protocol, has the same frame format and size as its predecessors and therefore makes the upgrade easy. Unlike 10BASE-T and Fast Ethernet, it supports full-duplex operating mode for point to point connections. A point can be a switch or

an end station. The CSMA/CD protocol is used only when there are shared connections and in such a case it operates in half-duplex mode.

Operating over optical fibre channels, it has been experimentally shown to achieve a throughput of over 720Mbps with a 100 percent offered load and collisions in half-duplex mode[Giga99]. Theoretically it can support a full-duplex throughput of 2 Gbps.

Although relatively new to the market, it's low cost compared to Myrinet and the ease of application migration virtually ensures Gigabit Ethernet to be the dominant technology.

Myrinet

Myrinet [Bode95] is a networking technology that offered Gigabit-per-second throughput, before the commercial availability of Gigabit ethernet. Based on technology used for packet switching in massively parallel processors, Myrinet has several interesting features.

- Each channel, a pair of which makes up a full duplex Myrinet link, has a data rate of upto 1.28 Gbps.
- An almost negligible bit error rate makes the network very reliable and eliminates the need of higher level protocols that traditionally assumed an error prone physical layer.
- Myrinet being a point-to-point network, the physical layer is not shared. Therefore the capacity of the entire network increases with the number of nodes. The network can be scaled by chaining switches.
- Packets are routed using a wormhole routing mechanism. A packet consists of a header, body and a tail. The header contains routing information that is stripped at each routing point. Once a packet enters a channel, it occupies the channel until its tail passes through. Other packets are prevented from using that channel in the meantime.
- Taking advantage of its reliable communication medium, the Myrinet uses cut-through routing, where data packets are forwarded as soon as they are received, as opposed to store-and-

forward routing, where the entire packet is buffered and verified before being forwarded.

- By using cut-through routing when a channel becomes blocked, the packet need not be queued on the routing circuit or node. The packet is blocked with flow control provided by the link.

Applications using a lightweight communication interface can obtain latencies of 5 microseconds and bandwidths of upto a Gigabit per second[Pryl98]. The only drawback of Myrinet is its price, which currently is about the same as that of a high end personal computer.

2.3.2 System Architecture

A NIC on our Myrinet testbed has a custom designed 33MHz processor, 1MB SRAM and 3 DMA engines, one each for transferring data from NIC to the wire, from the wire to the NIC and for transferring data between the host and the NIC. Data transfer is initiated by a Myrinet Control Program (MCP) running on the NIC.

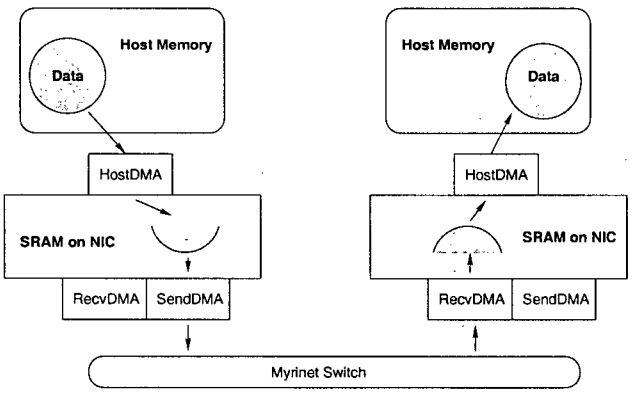

Figure 2.1: Dataflow between hosts

Figure 2.1 shows the basic data flow between two hosts on the network. The network interface resides on the hosts IO bus and therefore in addition to the DMA channel the host can access memory on the NIC via programmed IO (PIO).

The three DMA channels and the CPU all reside on the same memory bus called the LBUS, as shown in Figure 2.2. The bus arbitration protocol gives the CPU the lowest priority, with the

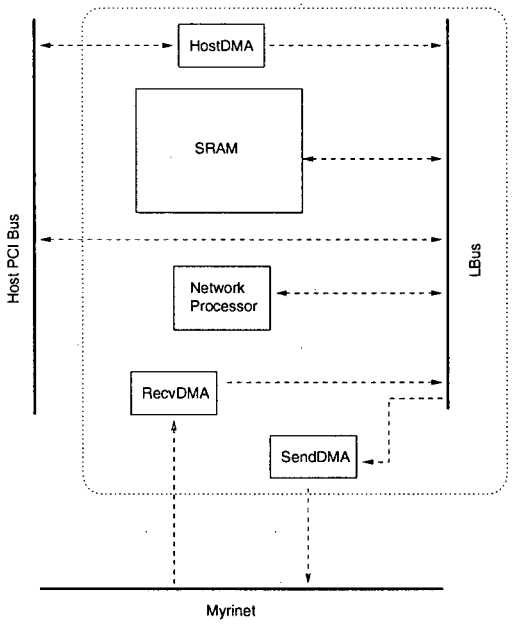

Figure 2.2: Myrinet NIC Architecture

RecvDMA getting the highest priority.

2.3.3 Performance Issues

The challenge facing network interface designers is to deliver hardware performance to the application with minimal degradation. Traditionally, it was the operating system that handled interaction with the NIC, multiplexing it among user processes. While doing so meant that users need not be concerned about security and sharing, it also meant expensive transitions into kernel space and copying data into kernel buffers prior to its transmission. Unix sockets have such properties, but were suited for Ethernet mainly because the communication medium was unreliable and slow. Efforts to impose the socket model on Myrinet have yielded poor performance rates[Rodr97] compared to the hardware capacity. Drastic differences in hardware characteristics require reinventing the communication interface model.

Following the path opened by microkernel researchers, network interface designers have stripped network management from the operating system and have introduced the concept of

User Space Communication where a user library takes advantage of hardware properties like low bit-error rates and sequenced delivery to implement a thin communication layer, eliminating OS overheads to directly interact with the network device in order to bring hardware performance closer to the application. Circumventing the OS creates a security problem. If more than one application makes use of the network interface simultaneously, measures must be taken to protect their data and address space boundaries from each other. The issue of security is addressed by Hamlyn[Buzz96] but systems like BIP[Pryl98] and PM[Tezu98] assume that the system has only a single user. BIP even goes as far as to restrict the number of communicating processes to one so that the communication layer can be kept simple and fast.

Protocols like TCP required user data to be copied onto kernel buffers before being transmitted, and vice versa. Data movement between buffers has been shown to be a primary cause for high latency in communication systems[Drus96]. A system transferring data to and from the user's data structures directly into the NIC is said to employ a **zero copy protocol**. Since NIC memory is mapped onto the user's virtual address space, messages can be moved between host and NIC memory by the host, same as performing a copy operation in host memory. Data transfer by DMA involves the additional step of determining the corresponding physical address of the host data area because the DMA is initiated by the NIC which is a device on the host I/O bus. Virtual to physical address translation involves looking up kernel data structures, so OS assistance is an absolute necessity in this case. Consequently, the page in host memory that holds the data area must be prevented from being swapped out by the VM system before DMA is complete. This is achieved by marking the page as "pinned". Some operating systems impose a limit, in addition to the obvious physical memory constraints, on the number of pages that can be pinned at any given time. Then pinned pages turn into a scarce resource that needs to be managed well.

When data transfers that are managed by the NIC are completed, the host should be informed so that the relevant data areas can be made free. The host can learn of this event either by being interrupted by the NIC or by polling on a status variable. Both these methods have their drawbacks. If the host spin waits, it would spend valuable CPU cycles which could otherwise be

spent on useful work. Interrupts on the other hand take up a significant proportion of message latencies [Thek93] for small messages due to the cost of the host CPU vectoring the interrupt and the cost of servicing the interrupt. If interrupt servicing causes a page fault, the cost of the operation goes up by three orders of magnitude [1]. If a message is kept waiting on the wire until the interrupt service routine can determine its destination in host memory, the message can block other messages and cause congestion because of the wormhole routing property of the Myrinet. The default behaviour of the NIC if a message is left unserviced for longer than a specified time, is to reset itself, causing all state to be lost.

2.4 Related Work

A review of existing work on message passing systems and interfaces for high speed networks is given in this section.

2.4.1 Selected Implementations of MPI

Both MPICH and LAM use point-to-point communication primitives to implement collective communication routines. MPI-CCL [Bruc95] focuses on optimizing collective communication on networks built on an unreliable broadcast medium such as Ethernet. It is built on URTP, a User level Reliable Transport Protocol that has both point-to-point and multicast capabilities. URTP is partially an extension to the OS kernel built on top of the networks Data-Link layer, and partially a user level library. Buffer management is unique in that send buffers are managed by the MPI-CCL layer and receiver buffers are managed by URTP. An incoming message goes through two layers of buffers, kernel and URTP, before being deposited into the user provided buffer. Moving data between buffers is in keeping with its objective of freeing kernel buffers as soon as possible so that packets are not dropped and retransmitted for lack of buffers.

MPI-FM [Laur97] was one of the first implementations of MPI on the Myrinet hardware. It

[1] disk I/O takes tens of milliseconds while message latencies are measured in tens of microseconds

replaced the P4 device of MPICH with an FM (Fast Messages) device. It addressed the multiple data copy problem by adding an upcall to the FM layer so that a received message can be reconstructed directly in its destination buffer, and by gathering non-contiguous data on the host into the network interface instead of into a temporary buffer as done in TCP/IP implementations.

The Globus metacomputing toolkit supports an implementation of MPI called MPICH-G [Fost98a, Fost98b] that insulates users from the details of underlying diverse computer architectures and networks. A metacomputing system can include environments that have diverse resource management, process management and security services, in addition to heterogeneous hardware with varying communication infrastructure. MPICH-G attempts to provide solutions for problems such as host access control, authentication, process scheduling process monitoring and customized communication hitherto left unaddressed by message passing libraries. Providing support for multimethod communication is the Nexus [Fost97b] secure channel device which uses TCP/IP when in the wide area, vendor-specific protocols within a host and shared memory within an SMP cluster. Because it uses Globus services MPICH-G can implement (but has not yet done) MPI-2 features like dynamic process creation, which the original design of MPICH did not encourage.

The Virtual Interface (VI) Architecture [Comp97] is a standard that specifies an interface between a SAN and the host with an intent of replacing heavy weight protocols such as TCP/IP. The VI Architecture consists of a VI kernel agent - a device driver that performs OS related operations, a VI user agent - a user library that implements the VI API giving users direct access to the NIC bypassing the OS, and a VI NIC - which has local memory and DMA engines which can access host memory without host CPU interaction. Its most interesting feature is the specification of an RDMA[2] operation where a local user process can initiate DMA on a remote host in order to access data of a remote application in a cache coherent manner. The VI standard has heavy industrial backing and so the introduction of a (commercial) implementation of MPI for VI called MPI/Pro [Dimi99] was not surprising. MPI/Pro uses threads for notification and to ensure message

[2]RDMA in the context of VIA stands for Remote DMA, and not Receive DMA as referred to in later chapters in the context of Myrinet

progress without user interaction. Long messages are handled by first using a rendezvous protocol and then by initiating an RDMA. By using multiple queues to hold receive requests, it reduces to O(1) the search complexity for a matching request on message arrival.

2.4.2 Messaging Systems on the Myrinet

A whole new avenue of research was opened with the introduction of user-level programmable NICs. Several research projects have proposed interesting methods for getting the most out of a high performance NIC. Some of those projects are reviewed in this section.

BIP[3]

BIP [Pryl98] is low-level message interface for Myrinet which thinly veils the hardware with a user library that delivers close-to-raw performance to the application and guarantees reliable in-order delivery of messages. An implementation of MPI called MPI-BIP is built using this library. It avoids packet dropping and retransmission by ensuring that the receiving node has buffer space, before transmitting a message. A credit based flow control scheme is used to this effect where the sending node has prior knowledge of the receiver's buffer availability. The communication protocols are message size dependent. Short messages are sent immediately to the destination node, irrespective of whether a receive has been posted, unless the sender runs out of credit in which case the sending routine blocks. Both sender and receiver rendezvous before the transmission of a long message, ensuring that the message can be delivered to the application immediately.

BIP has a simple channel interface to MPI where the number of channels can be set at initialization time by the user (the number of channels depends of the amount of memory available in the NIC). Messages are injected and removed from these channels. There is no look-ahead capability to a BIP channel and message matching is done by the host. BIP headers consist of the message route and tag. The MPI envelope is part of BIP message which has to be stripped by the host As a consequence of the simple channel implementation, BIP has the restriction that at

[3] **Version 0.95a**

most one send and receive can be active at a given time on a channel. The one message restriction imposes a limit to the degree to which communication and computation can be overlapped. Large messages are broken into fixed size packets (the optimal size of which is a function of the message length) and are pipelined along the critical path of host-to-NIC, NIC-to-wire, wire-to-NIC and NIC-to-host so as to get the best bandwidth. Messages are transferred directly between user space and the NIC. OS intervention is sought only to pin pages during DMA and this is done with the aid of a kernel module.

The BIP interface is limited to a single process. It provides notification of network errors to the upper layers and expects a higher level protocol to recover from them.

Hamlyn

Initially designed for large scale MIMD multicomputers, the Hamlyn network interface [Buzz96] was extended to the Myrinet to provide applications with a simple but efficient interface to the underlying hardware circumventing the OS. It introduced the concept of sender-based memory management where the sender specifies the message destination in the remote hosts memory so that messages can directly be deposited on receipt instead of being buffered. A 64 bit protection key in the message header ensures that applications are protected from unwanted messages being deposited in their address space. This process does require additional synchronization between sender and receiver. A unique packet numbering scheme is used so that out-of-order packets on an adaptive routing network can be reassembled sequentially. A receiving process can decide on the notification mechanism by indicate whether it needs to be interrupted or whether it polls a notification queue. On receiving a complete message, Hamlyn indicates its arrival by updating a notification queue in the host and interrupting the receiving process if needed. Message headers (metadata) are maintained in the applications memory in the host.

PM

PM [Tezu98] is another low-level interface which is also derived from MPICH and has explored issues related to zero-copy. Distinguishing it from other Myrinet interfaces is its cache of pinned

pages. It makes the assumption that most instances of data transfer occur from a local region in memory and therefore when a page is pinned for DMA, it is kept pinned even after the DMA completes so that the next data transfer would not have to incur the page pinning overhead. Pinned pages are maintained until system resource limitations require some of them to be freed in order to pin more pages. They are then freed in LRU fashion.

PM encourages remote memory writes where the sender specifies the destination address of a message. This is so that after an initial synchronization phase, the relevant data areas can be pinned and kept pinned until the application ends. Such a scheme, though cost efficient, is incapable of supporting a general purpose specification like MPI.

Active Messages

An Active Message [Eick92] is one that includes in its header an address of a user level process that, upon message arrival, is woken up to handle the message. This process is a privileged interrupt handler that is expected to quickly pull out the message from the network. AM is based on the programming model that arriving messages have preallocated buffers or that the message contains a simple request to which the handler can immediately reply. The Myrinet implementation of Active Messages, AM-II [Chun97], supports multiple host applications and has three different size based protocols. Applications have a staging area in their virtual address space to which medium and large messages are copied before sending. On the receive side, large messages are again sent through the staging area whereas medium messages are deposited in their final destination. Small messages transfers are zero-copy operations. Messages follow a request-reply scheme where all messages are acknowledged.

The AM API implements an abstraction called an endpoint with several configurable properties. Two endpoints form a virtual interconnect which we could call a channel. They protect applications from each other and have bindings to physical communication resources. Applications are free to create as many endpoints as they see fit. Since NIC memory is limited, host memory is used as a cache to hold inactive endpoints. Endpoint faults are serviced similar to page faults in

the VM system. One of the methods AM-II uses to maintain flow control is to assign to endpoints, credits based on queue sizes. In this scheme, individual senders can regulate themselves but multiple simultaneous messages to the same receiver from several senders can overrun receive buffers. Anything exceeding the endpoint's outstanding message limit is dropped and NACKed.

Illinois Fast Messages

Fast Messages [Paki97] from the University of Illinois is a low-level Myrinet message passing interface with a very small API. Similar to Active Messages, FM includes in its header, the ID of a message handler which would extract the message from the network. Unlike Active messages, FM does not follow a request-reply scheme and it also guarantees that messages are delivered in order.

Drawbacks of FM-1, chiefly intermediate buffering due to message header processing and unexpected receives, were fixed in FM-2 [Laur98] to offer better system level performance. In doing so, it increases the size of its API from three functions to five while supporting zero-copy gather and scatter operations. It also imposes a stream abstraction where messages can be sent and received in pieces.

The speed difference between the network processor and host processor was a key factor [Iann98] in dictating the work division in FM. The firmware running on the NIC was kept as simple as possible, thereby assigning all the work to the host processor. Message matching, fragmentation and reassembly are examples of tasks handled by the host.

Virtual Memory-Mapped Communication

VMMC [Dubn97b] is a user level communication interface supporting multiprogramming, buffer protection and zero-copy. Data transfer occurs after the sender and receiver make a rendezvous. Data areas exported by the receiver are imported and mapped onto the sender's virtual address space. User programs interact with a VMMC daemon to submit import export requests which are passed on to corresponding daemons on other hosts over ethernet. The daemon locks exported pages and passes their physical addresses to the importing daemon which writes them into a page table maintained in the NIC. Protection is ensured by having a local page table for each sending

process. Arriving messages are placed in user memory without requiring an explicit receive request.

VMMC-2 [Dubn97a] improves on the design by removing daemon mediation and having all communication over the Myrinet. It also relaxes the import/export requirements by accepting unexpected messages for a receiver, which are transferred to a default buffer and kept there until a receive is posted. Sends and receives are based on virtual memory while a user managed TLB contains virtual to physical translations. The UTLB is maintained in the host in order to accommodate a large number of mappings. Additionally, a UTLB cache is also maintained in the NIC to reduce lookup times from the host. VMMC-2 has a retransmission protocol to recover from network errors.

Trapeze

The Trapeze messaging system [Ande98] is integrated to the VM system of the OS and designed to support the Global Memory Service [Feel95]. GMS uses the network as a backing store for virtual memory pages swapped out of host memory and the main objective of trapeze is to support low latency transfer of pages across the network. The focused nature of its design allows trapeze to support fixed message sizes and offer no protection because communication is only between kernels.

Its most interesting feature is the way it handles messages. The Myrinet firmware uses cut-through delivery during data transfer, thereby fully utilizing the data pipeline from sending host memory to NIC to wire to receiving NIC to receiving host memory (see figure 2.1). It initiates DMA as soon as some data becomes available, rather than waiting for a complete packet. This operation is performed on a single packet, instead of fragmenting the data into smaller packets and incurring packet processing overhead. By getting all DMA engines in the data path to work simultaneously trapeze achieves bandwidth almost equal to that offered by the hardware.

Chapter 3

The Design of MPI-NP

The primary objective in designing MPI-NP was to reduce communication overhead on the host and offload it to the Network Processor in order to allow for as much overlap of computation and communication as possible. In order to achieve this objective the tasks related to communication should be identified. Candidate tasks/responsibilities for migration could be listed as

- routing of messages to the proper destination
- complete and reliable delivery of messages
- maintenance of protection boundaries between processes so that multiple processes can be supported
- management of message buffers
- implementation of specialized protocols for non-standard routines such as `Synchronous Send` and `Ready Send`
- matching of `Receive` requests with messages that have arrived
- performing collective communication with a group of peer processes

Traditionally some of these tasks were the responsibility of the host operating system. However since the NI is at user level, there is now the potential to move some or all of them to the NIC. In the case of MPI-NP, since we were interested in off-loading computation, we perform these tasks on the NIC. In the following sections we describe the abstractions and methodologies employed by MPI-NP to handle these tasks.

Another goal of message passing systems is to deliver the performance of the underlying hardware to the application. This is true of parallel applications which, by their very nature, are time critical. These characteristics include low latency, high bandwidth as well as the fair and efficient management of the memory and hardware resources in the network. Even though our objectives make few assumptions of the underlying hardware, an implementation has to ultimately be on top of a certain type of architecture. In this chapter, we also describe issues arising from the hardware we use, namely Myrinet, in trying to get good performance.

3.1 Channels

It is desirable that the communication pipeline support multiple active messages simultaneously so that the sending host could initiate transfer of multiple messages by simply depositing them at the top of the pipeline and then return to computation or continue sending messages to more destinations while the the messages themselves make progress, as opposed to blocking on the second message to process `A` because the first message is still in the pipeline or blocking on a **Send** to process `B` because a message for process `A` has blocked the network. Although it is not practical to have a pipeline of unlimited capacity, it is desirable that it support at least a few outstanding messages.

With multiple messages awaiting service, the NP must ensure that messages adhere to the MPI ordering rule while preventing messages which are waiting for rendezvous from blocking the progress of other messages. Since MPI allows an application to have multiple processes per host and MPI-2 allows for dynamic process creation, there is the added concern that when the NIC is shared by all communicating processes on the host, the messaging system should ensure that

messages do not unfairly use system resources and that they are delivered only to the intended destination process.

If the application topology were made known to the NIC, that information could be used to ensure a fair distribution of resources and also to route messages to their destinations.

In an effort to address the above issues, MPI-NP implements the abstraction of a virtual channel; a bidirectional communication path between every two MPI processes. Channels are implemented on the NIC where a channel consists of buffer space to store the body of messages and rings to hold MPI envelopes. They have complete knowledge of resources on either end. Resources used by channels are independent from one another which ensures protection and makes the behavior of the channel deterministic. The lack of buffer space in one channel does not block the progress of other channels as depicted in Figure 3.1.

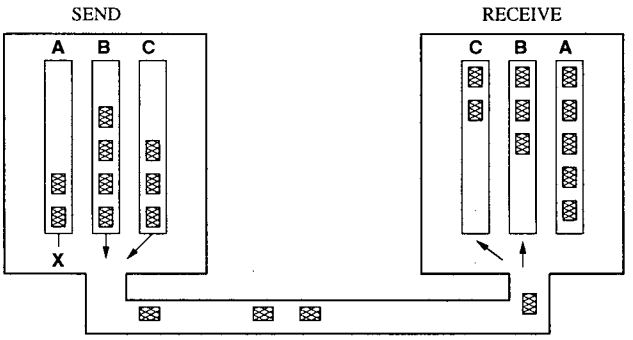

Figure 3.1: Message Progress in Virtual Channels

MPI-NP channels are defined between two processes and therefore can carry multiple message tags and MPI contexts. The only constraint of a channel is that it adheres to the MPI message ordering rule and does not allow message overtaking. This constraint does not apply to messages across channels. We define the *message visibility* inside a channel to be the look-ahead capabilities of receivers trying to extract messages from the channel. Because messages are buffered inside the channel structure several messages may be active at the same time inside the same channel but their 'tag' may not necessarily be identical. When matching messages, it is possible that, based

on tag values, a later message be matched over the oldest message. Any of the messages whose envelopes have arrived at the destination node are visible and available to be matched.

MPI-NP maintains a global channel queue on the NIC to schedule message transfer on the wire. MPI-NP schedules channels similar to how an operating system does processes. Figure 3.2 shows the state transitions of a channel.

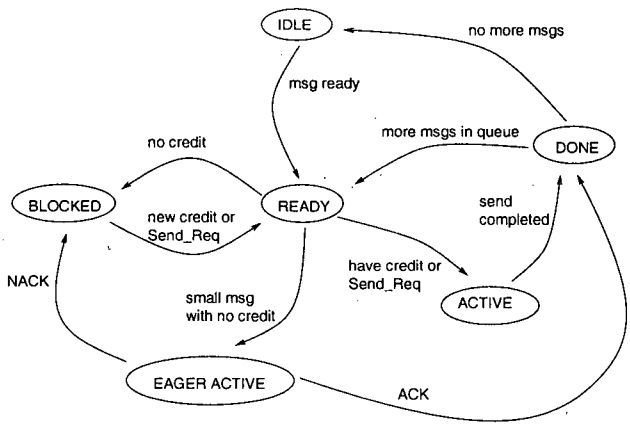

Figure 3.2: State machine for channels

Channels with messages ready for sending are placed on a ready queue. Each channel has its own ring of messages, allowing the application to continue injecting messages into the channel, even though the channel itself could be blocked. If a channel does not have sufficient credit to send the message at the head of its ring, it is deemed to be blocked and is rotated to the end of the scheduling queue. When the message is sent, the channel is removed from the queue or added back to end if it has more messages. Channels are serviced in a round robin fashion.

Channels are created at application `INIT` time. The current implementation has no support for dynamic process creation primitives described by MPI-2. As a result channel resources are statically allocated at `INIT` time. These resources include slots in a message table, send and receive buffer space. A possible solution for dynamic channel allocation is discussed in section 5.2. MPI-NP currently supports up to 64 channels (in 1 Mbyte of SRAM) where, in the case where the system's channel capacity is fully utilized, each channel is guaranteed at least 4.5KBytes each of send and

receive buffer space (9KB in total) and 32 slots in the message table. The number of messages that can be buffered in a channel is a function of the message slots owned by a channel and the message size. On the sender's side, the number of messages that can be injected into the NIC is limited by the size of the channel's send ring, which is currently at set 16, and the rate at which messages can be sent across the wire.

In summary

- Channels are created at application `INIT` time.
- A channel is made up of send and receive buffers, a table of received but unmatched messages and a table of messages expected by the application
- Channels facilitate fair use of resources by all communicating processes, adhere to the message ordering rule, simplify message matching and provide protection to processes at the NIC level.

The concept of a channel exists in BIP, AM and FM as well. BIP channels, implemented on the NIC, support only a single message in its pipeline. AM also implements its channels on the NIC and has no limit to its pipeline since its protocol allows for messages to be dropped. Channels in FM are completely implemented in the host and are distinguished by the message handler that services messages. FM uses flow control to limit the capacity of its pipeline to the number of receives posted.

3.2 Flow control

Reliable delivery of messages and network congestion avoidance are important issues in low level message passing systems. The wormhole routing characteristic of Myrinet can have messages block the path of other messages while they are in the network. A well designed MCP should retrieve all messages from the wire as quickly as possible in order to reduce congestion in the network. But messages that are DMAed in from the wire need to be buffered until they can be transferred to the host. Since the application matches messages based on a tag value rather than in order of

arrival, messages could occupy the receive buffer for an indefinite period of time. If the rate at which messages arrive surpasses the rate at which the host accepts them, buffer overflow occurs. Two possible ways of handling buffer overflow is to either use flow control to stall the sender and prevent new messages from entering the pipeline or to drop messages as TCP does and have it retransmitted.

We chose the former method because flow control helps ensure that messages do not cause network congestion and bandwidth is not wasted by dropped messages. MPI-NP uses a credit-based flow control mechanism similar to that used by BIP. The credit value is a combination of the size of the receive buffer and the number of free message slots on the receiver side of the channel.

Unlike other low-level interfaces, flow control is entirely handled within the NIC. The `MPI_Send` primitive hands the message to the NP which than handles its delivery to the destination. If all of the resources inside a channel are exhausted then flow control ensures that the next `SEND` will block until space becomes available, essentially reverting to sender-buffered communication [Buzz96], a strategy successfully used in other systems such as Hamlyn, FM and BIP.

3.3 Message protocols

MPI-NP handles the rich variety of message primitives defined in MPI such as blocking/non-blocking, synchronous, standard and ready-receive. It uses three protocols for transferring messages between a sender and receiver; full credit, rendezvous and eager sending of small messages. The first protocol is used when the sender has credit where as the other two are followed only when the sender does not have sufficient credit.

3.3.1 Full Credit

This is the simplest case, when the sender knows that there is enough buffer space on the receiver to accommodate the message, it is sent immediately. The sender's credit depreciates as messages are sent, and it is replenished when the receiving process removes the message from the channel's

receive buffer. Credit is transferred along with ACKs that are generated for every message. The sender doesn't necessarily block for an ACK if it has full credit, unless the message was a Synchronous Send, in which case it blocks until it gets an ACK informing it that the message was collected by the receiving process.

3.3.2 Message Rendezvous

There are two cases to consider depending on whether the send or the receive occurs first. Expected messages are messages for which the receive has already occurred, otherwise they are called unexpected messages. The rendezvous mechanism used by MPI-NP is essentially receiver driven.

Unexpected messages

In the case that the Send routine occurs much earlier than a matching Receive routine, the Send routine causes the message to be added to the appropriate channel structure on the NIC and the channel to be added to the scheduling queue. If there is sufficient credit the message is forwarded to the destination and its envelope is added to the message table. Once a Receive that matches an envelope is posted, the message is directly DMA'ed into the receiver's buffer.

If there isn't sufficient credit then a RECV_REQ control message is sent to the destination and the channel becomes blocked. In this case, the protocol becomes receiver-initiated and waits for the receiver to arrive and the message turns into an expected message.

Expected messages

In the case that the Receive routine occurs much earlier than a matching Send routine, the Receive executes and adds it's envelope to the expected message queue on the NIC. The NP sends out a SEND_REQ control message to the source specified in the message envelope (ANY flags are discussed in Section 3.4). If the send request matches the head of the channel send ring then the source responds with the message itself otherwise the source (i.e. sender) responds with a NACK. When the message finally arrives the source NIC sends out a RECV_REQ where now the

destination responds with a SEND_REQ causing the message to be sent out by the NIC on the sender side. Note that SEND_REQs always get a response in the form of a message or a NACK, whereas RECV_REQs do not require a response. This is to avoid misinterpreting control messages of the same MPI message as those of two different messages in the case where both the SEND_REQ and RECV_REQ occur simultaneously. Since the RECV_REQ matches an unacknowledged SEND_REQ it can be discarded with the knowledge that the sender will reply to the SEND_REQ with the message. If the protocol were changed where in such a case the receiver responded with another SEND_REQ, the sender would misinterpret this for a new SEND_REQ and respond with a second message which would have no corresponding receiver nor buffer space and would have to be dropped. The rendezvous protocol we follow avoids this situation.

Messages that are transferred in response to a SEND_REQ are expected and can be sent without credit. Since a matching receive exists, they are guaranteed to be delivered to the application and cannot block the progress of other messages.

MPI-NP's rendezvous protocol is receiver-initiated for all messages which do not have sufficient credit. In this case, the protocol takes advantage of this fact by having a special common receive area and directly transferring it to the host. For large messages, where sufficient credit is either difficult or impossible to acquire, the protocol degrades to ready-receive or, in the case of very large messages, to synchronous communication. Unlike the rendezvous mechanism in PM, we do the message matching on the NIC which can operate asynchronously to the host.

3.3.3 Eager send of small messages

Credit for a channel could be delayed by the receiver being busy sending out a large data message. The protocols described so far require the sender to block until it gets new credit or a SEND_REQ. A different protocol called eager send is used for very small messages, which could be important, characteristically small, control messages of the application, in an attempt to deliver them a little sooner. The current implementation uses an arbitrarily set size as the threshold. The optimal message size could be determined by the intersections of the plot of the time taken to send messages

of varying length with the plot of time taken to generate a control message.

In eager send, the sender sends the message immediately without waiting for credit. If the message is accepted by the receiver, it responds with an `ACK`. The sender would block until it gets the `ACK`, which is not too much of a delay, since it would have blocked anyway had it waited for credit.

If, on the other hand, the channel is full and the message cannot be accommodated, the receiver drops the message and returns a `NACK` to the sender. The sender does not re-send the message until it has sufficient credit to ensure the message will not be dropped again. Dropped messages waste the hardware resources of the network, but because messages are small and are dropped only once, its effect is expected to be minimal.

3.4 Message matching

Messages are matched based on three parameters; rank of process, message context and tag. Since messages are sent through channels, the rank is automatically matched, leaving only context and tag to be explicitly matched. These parameters, along with other meta information are contained in an envelope which is sent as the message header. Message headers of all incoming messages are stored in a message table which is shared by all active channels.

We initially tried an implementation that had the message table residing on the host because of space constraints on the NIC. This introduced additional overhead to the host because the host was required to search the table every time a `Receive` occurred. It also complicated the host interface as well as the work done by the MCP because in addition to buffering messages, the MCP was required to keep track of the message table using only costly DMA operations while the host had to synchronize with the MCP before searching, to avoid a possible race condition if the MCP were to upload new envelopes at the same time the host was searching the table. Although it could be safely said that host processors would always be faster than embedded network processors, the synchronization overhead added to heavy maintenance overheads on the NIC makes having the

message table on the host infeasible.

The current implementation locates the message table in the NIC in order to remove the mentioned overheads. The NP now bears a reduced maintenance cost while the host interface is considerably simpler. The only drawback of this scheme is that the size of the message table per channel is decreased due to the limited memory on the NIC, thus reducing the number of messages that could be buffered on the NIC.

Message matching takes place in three different ways.

- The message is sent first. It is buffered on the receiver and its envelope is placed in the message table. When the application posts a receive, the host places a request containing the parameters. The MCP matches this with the envelope in the table and goes on to deliver the body.

- The receive is posted first. The receiver maintains a list of expected messages. When the message eventually arrives, the NP matches it with an entry in this list and immediately transfers the message to the application's address space.

- The message is larger than the credit available. In this case, the sender transmits a `RECV_REQ` which is matched with with a pre-posted receive. If there is no match, the channel stores the envelope and compares it with subsequently posted receives until matched.

Wildcards

One of the advantages of the MPI-NP rendezvous protocol and the ability to match messages on the NIC is that it simplifies the handling of MPI wildcards. In simple low-level interfaces MPI wildcards must be handled by the host. The MPI standard describes two wildcards, `MPI_ANY_TAG` and `MPI_ANY_SOURCE`.

`MPI_ANY_TAG` wildcards are handled on a per channel basis, where we only search messages within a channel. The number of messages is limited to the visibility within a channel, which is not

expected to be large and therefore the search is sequential. One would expect that for most part, requested messages are near the top of the channel. We could implement a more sophisticated channel searching mechanism, however it may incur additional cost on every search and would require the message table to be well utilized in order to pay off. The current implementation keeps the simple and most common case fast.

Handling the MPI_ANY_SOURCE wildcard is a little more complicated. We are again constrained by the message ordering rule which requires MPI_ANY_SOURCE requests to be treated the same as other Receive requests. MPI-NP orders messages by channel since the common case is for messages to be matched by rank. When the host makes an MPI_ANY_SOURCE request messages in all channels become candidates for matching. We could have had an additional structure that held information about messages in all channels so that an MPI_ANY_SOURCE could be matched by simply querying this structure, but maintaining it would mean adding and removing information for every message received or delivered to the host. We have followed a simpler approach that makes the common case fast but is not as efficient for wildcards.

When the MCP gets an MPI_ANY_SOURCE request from the host, it searches sequentially all channels associated with the process making the request, until a match is made. For this reason channels are grouped by the process they belong to. For every MPI_ANY_SOURCE request, channels are searched in a rotating manner in order to prevent starvation by some channels. If there is no match, a request is added to all related channels. When a message arrives only requests in its channel are selected for matching. If a match occurs on an MPI_ANY_SOURCE request, the duplicated requests are removed by following a horizontal chain.

Our scheme allows requests to be met in the order that they were posted within a particular channel except for MPI_ANY_SOURCE requests, which are matched not necessarily in the order of message arrival across all channels because we do not compare message arrival times across channels. Since this is not a requirement of the MPI standard, we did not feel it necessary to add to the complexity by supporting this feature.

3.5 Supporting Zero Copy

Data movement between buffers has been shown to be a cause for high latency in communication systems [Drus96]. Most implementors aim to integrate a zero copy feature into their systems, where data is copied directly between application buffers and NIC buffers, eliminating traditional system buffers which served as intermediate staging areas. This has been made easier by the network being brought up to user level. MPI-NP uses memory on the NIC as its system buffer. NIC buffers on both sender and receiver ends make up one big system buffer. Data is transferred from user space to this system buffer and back into user space. we incorporate several strategies to achieve zero copy when messages do not fit our system buffer.

Sender buffering

Small messages are buffered on the receiver's NIC until their final destination address is known and zero-copy can be used. This allows messages to be sent before the receive is posted. Messages larger than the space available on the receiver's NIC are held on the sender until space becomes available. This is similar to what is done by BIP, FM and Hamlyn. In addition to enabling zero copy, a sender buffered scheme also contributes to maintaining flow control.

Address translation and page pinning

In order to DMA messages directly to and from the application's address space the NP needs to know its physical address. It also needs to be sure that the page in memory is not swapped out by the host operating system before the transfer is complete. We use a loadable kernel module in Linux to perform virtual to physical address translation and also to pin the page to prevent it from being swapped out. A designated area in host memory is used to hold translated addresses. It is not possible to hold all translations on the NIC since messages can span thousands of physical pages [Kim,97]. We could use the NIC to cache page entries but for very large messages the host would have to frequently synchronize with the NP and update the entries in order to adhere to the MPI message progress rule. A non-blocking call would not be able to perform updates and

message progress would stall until the user makes an `MPI_TEST` or `MPI_WAIT` call. By having a large page table in host memory, the host could, at most times, lock all the required pages and return to computation while the NP follows links in the table to send or receive the complete message. If the message is large enough that it runs out of page table entries, MPI-NP falls into user-driven mode where the message cannot make progress until another MPI communication routine is called by the application.

The current implementation supports more than 15,400 page entries in the page table, allowing a maximum message size of more than 60 MegaBytes to make progress without user intervention.

Holding page tables on the host for the sake of flexibility costs us communication bandwidth because the NP has to refer to it frequently in order to follow links and it can only do this by DMA. Our current implementation batches several pages into one entry in order to reduce the frequency of page table accesses. We have also optimized for the case where data resides on only one page by including that address along with the request so that the NP has one less reference to make.

3.6 Hardware issues

There are several optimizations that pertain directly to the Myrinet cards. One of these corresponds to size at which it becomes faster to use PIO rather than DMA to transfer the message from the NIC to the host. The cost associated with a DMA transfer is high and consists of overheads in address translation, page pinning, DMA setup and page unpinning. The technique used by MPI-NP is the same as that described by BIP. The threshold is experimentally determined and is dependent on hardware characteristics of the host machine. For small messages this decreases the message latency as seen in Figure 3.3, a plot of latency variations in MPI-NP.

The jagged PI/O line indicates a quirk in the hardware. We get the best latency when the message size is in multiples of 32 bytes which in our testbed also happens to be the width of the cache line[Shan97].

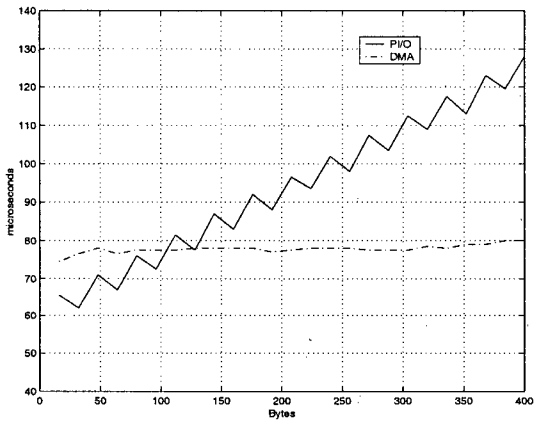

Figure 3.3: Variation of Latency with Method of Transfer

Getting back to our objective of reducing host overhead, we see in Figure 3.4 that the best `Send` overhead can be obtained if the message size threshold is in the region of 1 KByte, bigger than that shown in Figure 3.3. There is no overlap in the `Receive` overhead, indicating that all messages should be transferred through DMA. A possible reason for this is that the PCI bus has a write back cache which makes reads return slower than writes as the cache is not immediately flushed when data is written.

A second hardware specific optimization is message pipelining. There is a three stage pipeline from Host to NIC, NIC to NIC, and NIC to Host that when optimized can reduce the latency to send a single message. Both BIP and Trapeze describe techniques for pipelining of a single message. We use the Trapeze technique of cut-through delivery [Yocu97] where if the user's `Send` and `Receive` calls are properly timed, all DMA engines in the message's path are able work on the same message simultaneously in order to improve performance. The advantage of using cut-through delivery is that it is adaptive, doesn't require a fixed packet size and allows us to packetize the messages according to page boundaries.

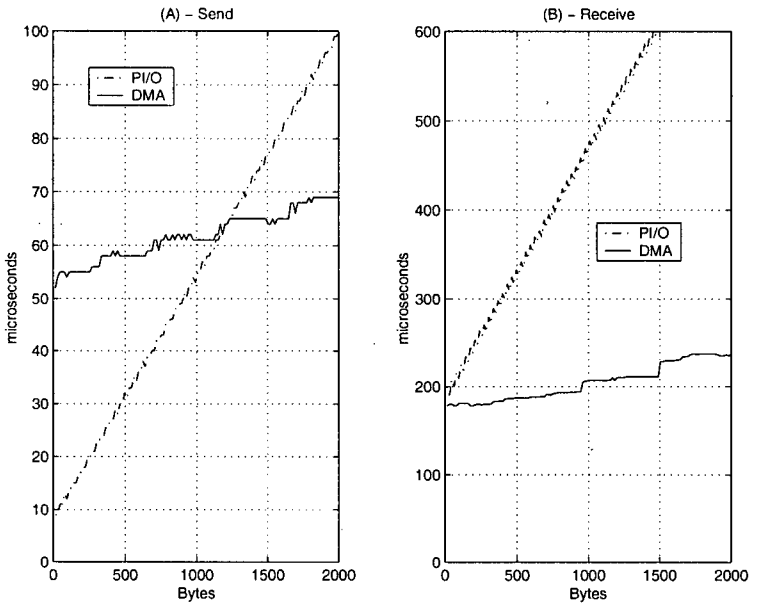

Figure 3.4: Variation of Overhead with Method of Transfer

3.7 Architectural Overview

MPI-NP consists of a user library, an MCP and a loadable kernel module. The library interacts with the MCP to exchange MPI messages and invokes the kernel module to pin pages to memory and translate virtual memory addresses into physical addresses. A fixed region in host memory called the copyblock holds a page table containing addresses of pinned pages.

The memory on the NIC is divided into two areas; the lower address region which holds the text, data and stack of the MCP and the upper region on which reside data structures and buffers shown in Figure 3.5.

Of the several lists maintained by the NIC, the most important is the list of channels. Each channel, whose use is described in section 3.1, has a list of expected messages (those for which the application has posted an `MPI_Recv` operation but has no matching message), a list of messages already received but yet unclaimed by the host, a private send ring, receive ring, and associated

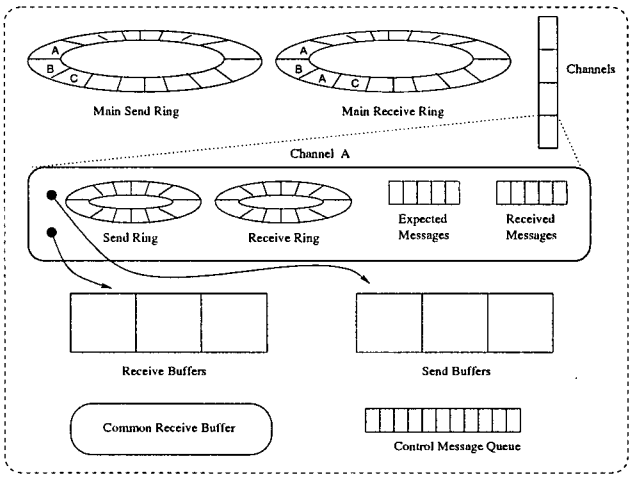

Figure 3.5: The Layout on the NIC

send and receive buffers. The rings hold the meta data while the buffers hold the message body. The MCP sends out messages in the order that it receives them from the host. Therefore the send buffer acts as a message queue as shown in Figure 3.6. When a message is too large to be added into the end of the buffer it is wrapped around to the beginning. In this way, the buffer can be used to accommodate messages of unlimited size as long as it makes progress on the network. If the buffer is filled before the entire message is downloaded onto the NIC, the remaining part of the message is left on the host until more buffer space becomes available.

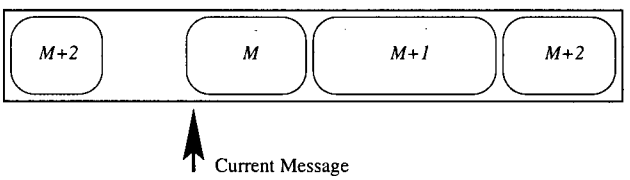

Figure 3.6: Messages in a Channel's Send Buffer

The receive buffer is organized as a heap because the application may select messages based on a tag value rather than in order of arrival. Messages are placed in the first available free space and are fragmented according to space distribution, as shown in Figure 3.7.

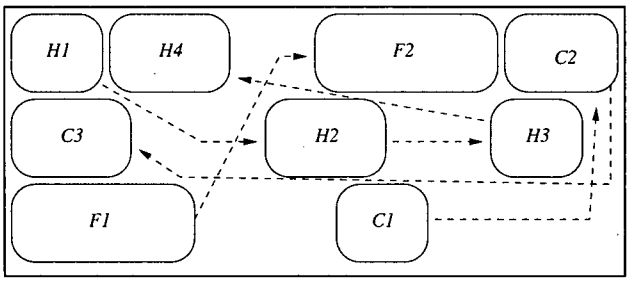

Figure 3.7: Messages in a Channel's Receive Buffer

The main send ring is in essence a ready queue for channels with outgoing messages waiting to be serviced. A channel can have only one entry in this ring in order to preserve the message sequence because at the time an entry is posted, it is not known whether the message can make progress. Since blocked channels are rotated in the ring to make room for active channels, multiple entries of a single channel in this ring can cause messages to be sent out of sequence unless an elaborate scheduling scheme is used. Maintaining only a single entry keeps ring management simple while preserving message sequences. The receive ring is a list of channels with messages which have been claimed by the application and are waiting to be uploaded. A channel can have a number of entries in this ring because message destinations are known and they are guaranteed to make progress.

When an expected message eventually arrives it is sent through a common receive buffer instead of going through the channel's receive buffer, since its destination is known, Messages longer than expected and non-MPI data packets are dropped.

In addition to data messages, MPI-NP uses control messages to enable MCPs to communicate with each other. Control messages are generated by events triggered either by the host or by incoming messages. Since there is only a single outgoing hardware link which may be utilized by both data and control messages, the latter is placed in a queue until this link becomes free and then transferred onto the wire.

Host Interface

The host has a narrow interface to the network and is based again on message size. When sending a small message, if the channel send ring is empty, the message is copied into the send buffer and a send request is placed on the NIC. If the channel is occupied by messages, the large message protocol is followed because the host overhead of accurately determining the location to copy the message to, taking the asynchronous operation of DMA engines into account, is too high.

When sending large messages, the host locks the relevant pages in memory and places a send request indicating the page table entry. It is then free to continue with application processing unless it is a blocking call, in which case it polls on the page table entries until they are freed by the MCP and then unlocks them.

When receiving messages, the host follows the same procedure as sending large messages, except that it places a receive request instead of a send request. No size distinction is made when receiving messages since the hardware dictates (Figure 3.4) that it is cheaper to DMA messages of all sizes.

Special send operations like `Synchronous Send` and `Ready Send` are handled in the NIC. Again the host only places the `Send` request which has one or more associated page table entries. The success of the operation is indicated by a status flag in the page table entry which is updated by the MCP in the usual manner.

Host requests are placed in a designated area in NIC memory which the MCP polls. If more than one process wishes to interact with the MCP, a system semaphore ensures that interaction happens only one process at a time to prevent a race condition from occuring otherwise. An alternative to obeying a semaphore is for the processes to place their requests in their respective channels and have the MCP poll each channel. This would be effective for the host processes, but makes the MCP's task tedious on any architecture.

Chapter 4

Evaluation

MPI-NP was developed on the Linux operating system (RedHat 5.1 kernel version 2.0.35). The experiments were conducted on a cluster of 266MHz Pentium PII PCs with 128MB of RAM connected by a Myrinet network with a LANai 4.1, 33MHz processor and 1 MB SRAM on the NIC.

In obtaining measurements, we tried to simulate a typical application's environment where the data has only just been generated and therefore total time includes time to flush the cache into memory before data transfer. We adopted a pessimistic page pinning policy (refer to sections 3.5 and 5.2) by assuming that every message was from a new data area and therefore for large messages we pinned the relevant pages every time data transfer was about to take place and unpinned them when transfer completed.

Timings were obtained in the following manner. The sending application generates new data, starts the timer, sends the data and receives the same amount of data as a reply before it stops the timer and determines Ts as shown in Figure 4.1. The receiver, waits for data to arrive, starts its timer, writes new data, stops the timer, sends the data back to the sender and determines Tr. One way time is given by $(Ts - Tr)/2$. This procedure was repeated 256 times for each message size and the median value was used in plotting curves. Time was measured by counting machines cycles and dividing it by the clock speed of the CPU to convert it into units of microseconds. We compare our results with BIP because its architecture is exactly the opposite of MPI-NP. MPI-NP

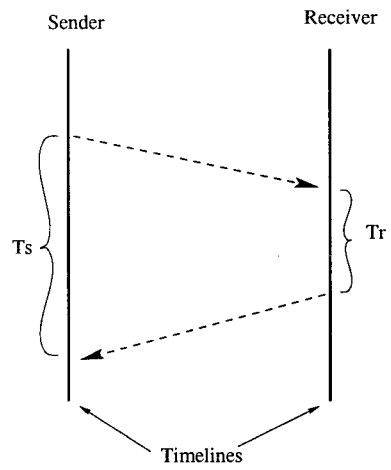

Figure 4.1: Time Measurements

tries to make maximum use of the NP whereas BIP has only a thin software layer on the NIC, barely enough to facilitate data transfer thereby getting the best performance from the network hardware.

4.1 Bandwidth

Figure 4.2 is a plot of the one-way bandwidth measurements as a function of message size. We reach a peak application level bandwidth of 70 MBytes per second.

In our first implementation, a page table entry held the translation of one physical page. This meant that data in a page in host memory required three DMA operations just to be downloaded to the NIC; one for fetching its address from the page table, one for downloading the data and one to mark that table entry as free so that the host can unpin the page. This is a significant overhead for 4KB, or less, of data per page. The method only yielded a peak bandwidth of 60 MBytes per second. The current implementation performs a page table lookup and update every four pages. Each page table entry now contains address translations of four pages to accommodate this protocol thereby reducing the frequency of page table look up. There is still a waste of bandwidth if the data does not span four pages. Communicating page information to and from host memory and

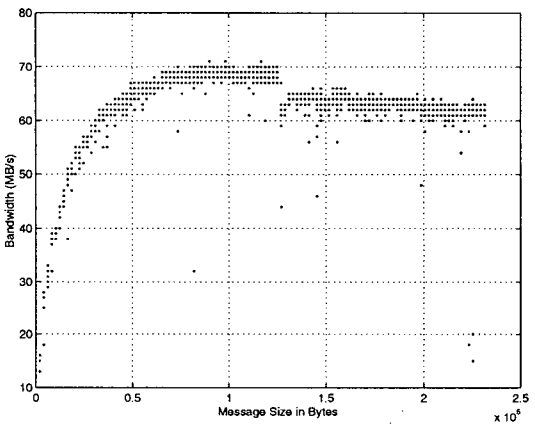

Figure 4.2: Bandwidth vs Message Size

the fact that MPI-NP pins and unpins the related pages every time a message is transferred, are the main reasons for the current implementation being unable to achieve a higher bandwidth. All these facts highlight the need for a better page management scheme to reduce the frequency of page table look-ups by the NIC.

Due to memory limitations on the NIC, for messages larger than available buffer space, the buffer is reused in a rotating manner, i.e. as soon as an area is known to be free it is filled with a block of data as explained in section 3.7. The drop in bandwidth as the message size increases beyond buffer size is caused by the overhead of reusing buffers.

4.2 Host Overhead

Since MPI-NP relieves the host of several communication related tasks the overhead incurred by the host is kept as low as possible. Figure 4.3 compares the host overhead, during a Send operation, of MPI-NP with that of BIP[1].

The overhead in MPI-NP gradually rises when the message size increases but when it passes

[1]This particular test was on the latest version BIP-0.98a which works on the newer Linux 2.2.x kernels. BIP-0.95a, the version used for other evaluations, had an extremely high and seemingly inaccurate overhead which, personal communication with the developers of BIP revealed, is not evident in the newer releases

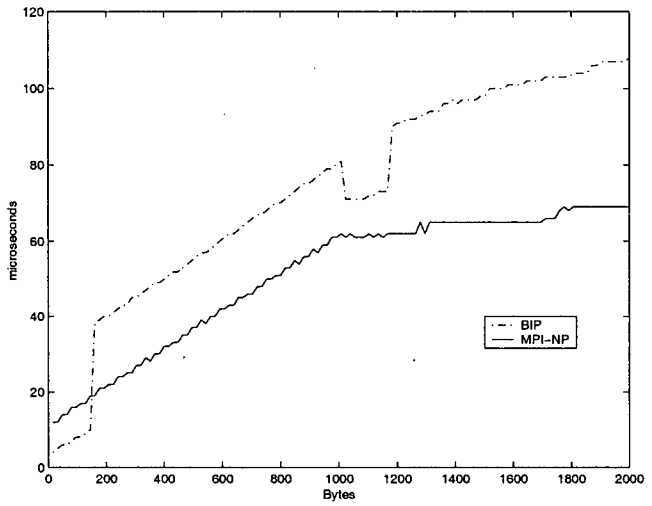

Figure 4.3: Comparison of Host Overhead Variation

the point where the protocol changes from small to large message size, 1024 Bytes, the rise depends on the number of pages pinned. The BIP curve makes a transition at 200 Bytes which we presume is the small message threshold. It's behaviour at 1000 Bytes is unexplained. BIP-0.97 has an identical behaviour at 1000 Bytes.

BIP performs better for very small messages but MPI-NP performs significantly better for larger messages. It must be noted here that BIP has a constraint of supporting only one message in the pipeline allowing it to make assumptions about the channel's occupancy whereas MPI-NP is more flexible and as a result has more host overhead in deciding which protocol to use depending on channel occupancy in the NIC, as described in section 3.7.

4.3 Latency for Small Messages

The lowest application level latency we obtained for a 4 Byte message was 68 microseconds. This value is high compared to the values reported by other messaging systems referred to in this paper. The median value of one-way latency as a function of message size is shown in Figure 4.4 and is

compared to the latency obtained by BIP. The sharp rise in the BIP curve at 256^2 Bytes is when BIP changes its protocol from small to large messages.

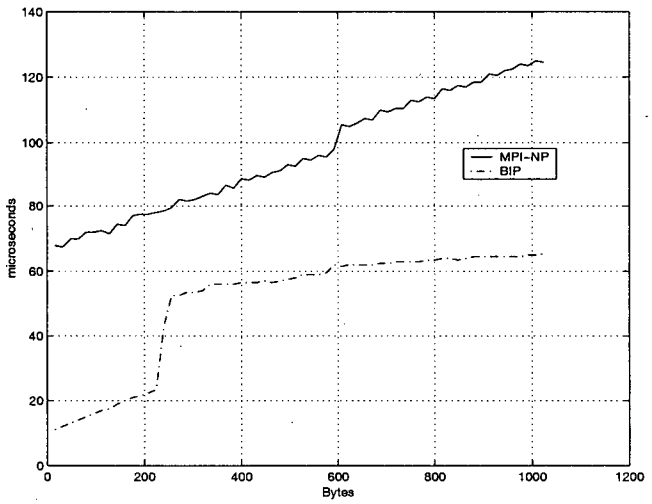

Figure 4.4: Comparison of Latency Variation

Figure 4.5 presents a breakdown of the time a small message spends in the communication pipeline when the message is expected, i.e. a matching receive is already posted. The overlaps of time segments on the NIC depict time required to clean up queues after a message has been sent/received.

We see that 75% of the time is spent on computation by the NIC. MPI-NP relies heavily on the NP, but the 33MHz processor on our testbed is unable to deliver good performance. In addition to being eight times slower than the host, it is further hampered by having to share the memory bus with the host and network DMA engines. This is a peculiarity of the Myrinet hardware used as our testbed.

Access to the local memory bus in LANai 4.1 is prioritized as follows. The host DMA gets priority over the NP on the rising edge of a clock cycle. On the falling edge, the Receive DMA

[2]BIP-0.95a was used for all other tests because it ran on the same Linux kernel MPI-NP was developed on. The threshold seems to differ between versions

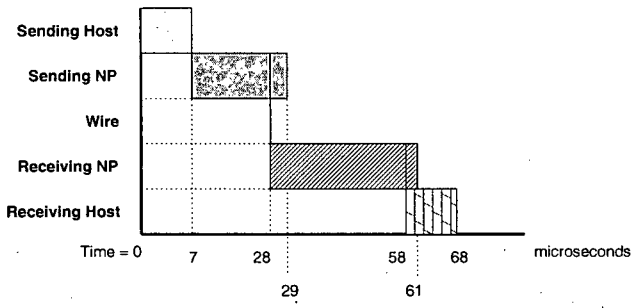

Figure 4.5: A Breakdown of Message Latency

gets highest priority, followed by the Send DMA with the NP again getting least priority. During program execution, the NP uses one edge of the clock cycle to fetch the instruction from memory and another edge to execute it. Contention for the memory bus occurs during fetching and, if the instruction required a memory reference, during execution. Figure 4.6A shows NP slowdown when one DMA engine is at work. The NP had access to memory on every other cycle edge. The computation had 50% memory reference instructions which is reflected by the fact that it slowed down by 50%.

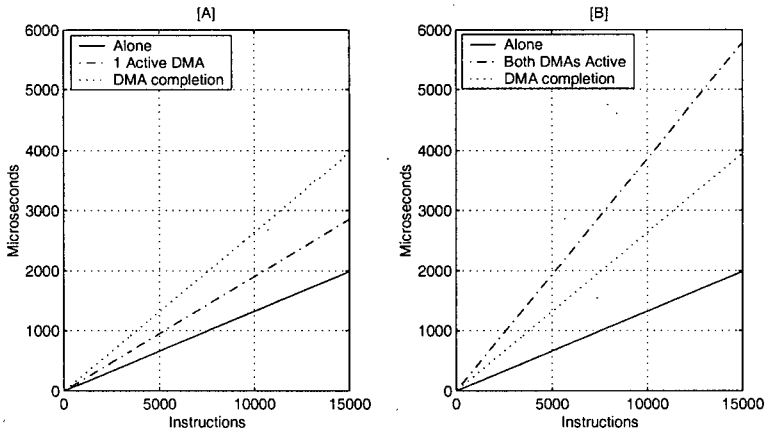

Figure 4.6: Impact of DMA Activity on the Network Processor

Figure 4.6B shows NP slowdown when both the host DMA and Send DMA engines are active. In this instance the NP loses access to both edges of a cycle. Since the LANai 4.1 chip does

not have an instruction cache, program execution stops until either DMA completes.

4.4 Discussion

The constraints mentioned above would no longer exist with the introduction of a faster network processor with an on-chip cache. Commonly available embedded processors have a cache, are several times faster than the one aboard the Myrinet NIC, are of extremely low cost and are improving fast in all above factors. The technology on our testbed is two years old. Yet the embedded processors available at that time were still more powerful than the LANai 4.1. An example of a two year old processor is the NEC R4300[Micr97] running at 133 MHz with a 16K cache and, most importantly, a price tag of $32 at the time. With a processor of similar power, the time a message spends in the NIC taken from Figure 4.5 would reduce less than 13 microseconds, not taking cache effects into account. With a cache, the NP need no longer block because of DMA activity and we would see a drastic decrease latency.

Current embedded processors are much more powerful[Micr99] as are host processors[Dief99]. Systems like MPI-NP that heavily utilize the NP have much to gain from hardware advancements. Since MPI-NP does more work on the NIC, its percentage of improvement would be greater than systems that do more processing on the host. Improvements in host processor speed will also be of higher benefit to applications using MPI-NP since they consume less CPU cycles for communication.

Chapter 5

Conclusions

Recently, computationally intensive applications with supercomputing needs have found PC clusters to be a cheap and effective alternative to massively parallel processors. Contributing to this change are rapidly improving price/performance ratios of personal computers and momentous advances in networking technology that allows data transmission throughput to be measured in Gigabits per second. A change in network interface design by including an embedded processor on the NIC and making it directly accessible at user level played a significant role in the development of several communication systems that come close to supplying the raw hardware performance of the network to the application layer. An issue largely overlooked in the development of these systems is the utilization of the NP to reduce communication overhead on the host, a parameter which directly impacts the granularity of communication in parallel applications.

5.1 MPI at the Network Layer

We have researched the feasibility of offloading communication related tasks from the host processor to the network processor in order to reduce host communication overhead. Our thesis was based on the assumption that the architecture of the NIC and especially the NP was capable of effectively handling more than average workloads. We have designed and implemented a prototype

message passing system called MPI-NP, to test our hypothesis. MPI-NP is designed to provide the communication layer for LAM, a public domain implementation of MPI.

We have made use of many of the optimization techniques described in the literature and added a few of our own, to support latency saving features like zero copy and eager sending, bandwidth enhancing features like cut-through delivery and page batching, flow control using credits and message rendezvous protocols. Based on hardware characteristics of our testbed, we decided to use PI/O for small messages and DMA for large messages when transferring data between host and NIC.

We were successful in relieving the host processor of tasks such as message matching, flow control, protocols for non-standard sends, and several other tasks which were traditionally responsibilities of the operating system such as routing, resource sharing, flow control and buffer management. MPI-NP implements a channel abstraction on the NIC and makes the NP aware of the process topology and details of message envelopes to facilitate many of these tasks. This information makes it possible to implement collective communication with minimum interaction from the host. The host now only initiates communication and, in most cases, is free to return to the application while the NP manages the messages. This work was focused more on designing the network layer. Related research at the UBC Distributed Systems Lab by Chamath Keppitiyagama has resulted in reducing the application interface of MPI-NP, in some cases, to just one function call, with a total host overhead of 4 microseconds in sending a message.

Migrating functionality to the network processor reduces host overhead. Unfortunately the configuration of NICs on our testbed impedes overall performance of the communication system as the NP is unable to effectively handle the workloads assigned to it. Existing technology and the microprocessor market is conducive to designing a better performing NIC. A change in hardware architectures where the NP performance is closer to the performance of the host would benefit applications and favour the design philosophy of MPI-NP.

The question to be asked here is how much improvement should the network processor gain in order for MPI-NP to be effective. The current trend suggests that host processors are faster than

NPs and therefore for certain types of benchmarks, like latency, it is always better to do more work on the host. BIP, the antithesis of MPI-NP, has reported a latency of about 4 microseconds by using only the BIP API. MPI support is added on top of the BIP library. We can surmise that a small BIP message spends about 2 microseconds in the network layer. In order to achieve a similar timing on similar hardware, MPI-NP requires a NP 24 times faster than the 33MHz LANai 4.1, which works out to be more powerful than the 266 MHz host processor. This leads to the question of which parameter, latency or overhead, is more important. The state-of-the-art indicates that a trade-off between the two parameters needs to be made depending on the requirements of the application. Applications with fine-grained communication needs are better served by the philosophy of BIP whereas compute intensive applications with more coarse-grained and collective communication needs would work well over MPI-NP.

5.2 Future Work

MPI-NP is not yet fully compliant to the MPI specification. In the current implementation channels are identified by the global rank of the process, which is how messages are matched with requests. The issue of handling inter-group and intra-group communication is still open. In group communication the process ranks of a group may not be the same as their global rank. We foresee additional parameters in the host interface and message protocols to support groups. This means that the NP would be required to do more work to determine group composition and absolute ranks of processes whereas the host complexity would remain unchanged.

The current implementation does not scale above 64 channels because of the static allocation of resources. There are several drawbacks to statically creating channels. First, the amount of buffer space available for each channel decreases as the number of MPI processes increase. Secondly, it is not possible for channels that are rarely used or not used at all to allow their resources to be used by other more active channels. A solution to this problem is to treat the NIC memory as a cache for channels. Channels would be dynamic and would not become active until they are first used and

when inactive, can be freed or swapped up to the host. The MPI-2.0 specification [Mess97] describes dynamic process creation and management which is already supported by LAM. In designing MPI-NP, we have foreseen the need to support dynamic channels, and have decoupled channels from network resources, such as buffers, message tables and request tables, as much as possible.

MPI-NP currently employs a pessimistic page management strategy. Other known strategies used get around the page management overhead need to be investigated in order to implement a better page management scheme. Possible alternatives would be to maintain a TLB on the NIC, as done by U-Net [Eick95] and VMMC-2, and to cache pinned pages, as done by PM, with the hope that the next message also comes from the same vicinity as the previous message, thus avoiding pinning and unpinning the same page twice in quick succession.

For very large messages, message progress is user driven in the current implementation. If the system runs out of page entries to accommodate the entire message, progress of non-blocking requests could stall until the user makes another communication request, at which time previous pending requests are attended to. Several implementations of MPI (LAM, MPICH, MPI-BIP) are user driven in this manner even though it means that they do not implement the MPI message progress rule. A possible approach would be to emulate MPI/Pro and FM by creating a new thread of control for every communication request, which would monitor message progress. The host overhead of running multiple threads, switching contexts between them and the cost of being interrupted by the NIC need to be quantified before such a method is selected.

Almost all MPI systems implement collective communication on top of point-to-point primitives which requires the host to interact with MCP several times incurring the associated software overhead every time. By making the NP aware of the application topology we have laid the foundation for migrating collective communication into the NIC. The implementation would require only one interaction with the host per call, further reducing host overhead. The performance of collective routines would also increase because of the reduced interaction.

One of the characteristics of the Myrinet is its negligible bit-error rate. Following the example of other systems[Arak98], MPI-NP takes advantage of this property to avoid error checking.

A future implementation would perform CRC checks on data and insulate the application from the rare network error.

Bibliography

[Ande95] Anderson, T.E., Culler, D.E., Patterson, D.A., and the NOW Team. "A Case for Networks of Workstations: NOW". *IEEE Micro*, February 1995.

[Ande98] Anderson, D.C., Chase, J.S., Gadde, S., Gallatin, A.J., Yocum, K.G. and Feeley, M.J. "Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet". *Proc. of the USENIX Technical Conference*, June 1998.

[Arak98] Araki, S., Bilas, A., Dubnicki, C., Edler, J., Konishi, K., and Philbin, J. "User-Space Communication: A Quantitive Study". *Proc. of Supercomputing 98*, November 1998.

[Bake99] Baker, M., and Buyya, R. "Cluster Computing: The Commodity Supercomputing". *Software - Practice and Experience*, Vol. 29, No. 6, 1999.

[Bhoe98] Bhoedjang, R., Romein, J., and Bal, H.E. "Optimizing Distributed Data Structures Using Application-Specfic Network Interface Software". *Intl. Conf. on Parallel Processing*, pp. 485–492, August 1998.

[Bode95] Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., and Su, W. "Myrinet - A Gigabit-per-Second Local-Area Network". *IEEE Micro*, Vol. 15, No. 1, pp. 29 – 36, February 1995.

[Bruc95] Bruck, J, Dolev, D., Ho, C., Rosu M., and Strong, R. "Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations". *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 64 – 73, July 1995.

[Burn89] Burns, G. "A Local Area Multicomputer". *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[Burn90] Burns, G., Dixit, V., Daoud, R., and Machiraju, R. "All About Trollius". Occam Users Group Newsletter, August 1990.

[Burn94] Burns, G., Daoud, R., and Vaigl, J. "LAM: An Open Cluster Environment for MPI". *Supercomputing Symposium '94*, June 1994.

[Butl94] Butler, R., and Lusk, E. "Monitors, Messages and Clusters: The P4 Parallel Programming System". *Parallel Computing*, Vol. 20, pp. 547 – 564, April 1994.

[Buzz96] Buzzard, G., Jacobson, D., Mackey, M., Marovich, S., and Wilkes, J. "An Implementation of the Hamlyn Sender-Managed Interface Architecture". *Proc. of the Second Symposium on Operating System Design and Implementation (OSDI '96)*, pp. 245 – 259, October 1996.

[Chun97] Chun, B., Mainwaring A., and Culler, D.E. "Virtual Transport Protocols for Myrinet". *Hot Interconnects V*, August 1997.

[Coad99] Coady, Y., Ong, J.S., and Feeley, M.J. "Using Embedded Network Processors to implement Global Memory Management in Workstation Cluster". *Proc. of the IEEE Symposium on High Performance Distributed Computing*, August 1999.

[Comp97] Compaq, Intel and Microsoft. "Virtual Interface Architecture Specification, Version 1.0". http://www.viarch.org, December 1997.

[Dief99] Diefendorff, K. "Athlon Outruns Pentium III". *Microprocessor Report*, Vol. 13, No. 11, August 23, 1999.

[Dimi99] Dimitrov, R., and Skjellum, A. "An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing". *Proc. of the Third MPI Developer's Conference*, March 1999.

[Drus96] Druschel, P. "Operating System Support for High-Speed Communication". *Communications of the ACM*, Vol. 39, No. 9, pp. 41 – 51, September 1996.

[Dubn97a] Dubnicki, C., Bilas, A., Chen, Y., Damianakis, S., and Li, K. "VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication". *Proc. of Hot Interconnects V*, August 1997.

[Dubn97b] Dubnicki, C., Bilas, A., Li, K., and Philbin, J. "Design and Implementation of Virtual Memory-Mapped Communication on Myrinet". *Proc. of the 1997 International Parallel Processing Symposium*, pp. 388 – 396, April 1997.

[Eick92] Eicken, T., Culler, D., Goldstein, S., and Schauser, K. "Active Messages: A Mechanism for Intergrated Communication and Computation". *Proc. of the 19th Annual Symposium on Computer Architecture*, pp. 256 – 266, May 1992.

[Eick95] Eicken, T., Basu, A., Buch, V., and Vogels, W. "U-Net: A User-Level Network Interface for Parallel and Distributed Computing". *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pp. 40 – 53, December 1995.

[Feel95] Feeley, M.J., Morgan, W.E., Pighin, F.H., Karlin, A.R., Levy, H.M. and Thekkath, C.A. "Implementing Global Memory Management in a Workstation Cluster". *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pp. 201 – 212, December 1995.

[Fost96] Foster, I., Kesselman, C., and Snir, M. "Generalized Communicators in the Message Passing Interface". *Proceedings of the MPI Developers Conference*, July 1996.

[Fost97a] Foster, I., and Kesselman, C. "Globus: A Metacomputing Infrastructure Toolkit". *Intl Journal of Supercomputer Applications*, Vol. 11, No. 2, pp. 115 – 128, 1997.

[Fost97b] Foster, I., Karonis, N.T., Kesselman, C., Koenig, G., and Tuecke, S. "A Secure Communications Infrastructure for High-Performance Distributed Computing". *Proc. 6th IEEE Int'l Symposium on High Performance Distributed Computing*, 1997.

[Fost98a] Foster, I., and Karonis, N.T. "A Grid-Enabled MPI: Message Passing in Heterogenous Distributed Computing Systems". *Proc. of SC98: High Performance Networking and Computing Conference*, November 1998.

[Fost98b] Foster, I., Geisler, J., Gropp, W., Karonis, N., Lusk, E., Thiruvathukal, G., and Tuecke, S. "Wide-Area Implementation of the Message Passing Interface". *Parallel Computing*, Vol. 24, No. 11, pp. 1735 – 1749, 1998.

[Giga99] Gigabit Ethernet Alliance. "Gigabit Ethernet: accelerating the standard for speed". *whitepaper*, May 1999.

[Grim97] Grimshaw, A., and Wulf, W. "The Legion Vision of a Worldwide Virtual Computer". *Comm. of the ACM*, Vol. 40, No. 1, pp. 39 – 45, January 1997.

[Grim98] Grimshaw, A., Ferrari, A., Lindahl, G., and Holcomb, K. "Metasystems". *Comm. of the ACM*, Vol. 41, No. 11, pp. 46 – 55, November 1998.

[Grop95] Gropp, W., and Lusk, E. "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.

[Grop96] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. "High-performance, Portable Implementation of the MPI Message Passing Interface Standard". *Parallel Computing*, Vol. 22, No. 6, pp. 789–828, September 1996.

[Iann98] Iannello, G., Lauria, M., and Mercolino, S. "Cross-platform Analysis of Fast Messages for Myrinet". *Workshop on Communication, Architecture and Applications for Network-based Parallel Computing (CANPC 98)*, February 1998.

[Kara94] Karamcheti, V., and Chien, A. "Software Overhead in Messaging Layers: Where does the time go?". *Proc. of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pp. 51 – 60, October 1994.

[Kim,97] Kim, J., and Lilja, D.J. "Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs". Technical Report HPPC-97-10, High-Performance Parallel Computing Research Group, University of Minnesota, 1997.

[Laur97] Lauria, M., and Chien, A.A. "MPI-FM: High Performance MPI on Workstation Clusters". *Journal of Parallel and Distributed Computing*, Vol. 40, No. 1, pp. 4 – 18, January 1997.

[Laur98] Lauria, M., Pakin, S., and Chien, A.A. "Efficient Layering for High Speed Communication: Fast Messages 2.x". *Proc. of the 7th High Performance Distributed Computing Conference*, July 1998.

[Mess95] Message Passing Interface Forum. "MPI: A Message-Passing Interface Standard". http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html, June 1995.

[Mess97] 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.

[Micr97] Microdesign Resources. "Chart Watch: Embedded Processors". *Microprocessor Report*, Vol. 11, No. 4, March 31, 1997.

[Micr99] Microdesign Resources. "Chart Watch: Embedded Processors". *Microprocessor Report*, Vol. 13, No. 3, March 8, 1999.

[Nupa94] Nupairoj, N., and Ni, L. "Performance Evaluation of Some MPI Implementations on Workstation Clusters". Technical report, Department of Computer Science, Michigan State University, October 1994.

[Paki97] Pakin, S., Karamcheti, V., and Chien, A.A. "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors". *IEEE Concurrency*, Vol. 5, No. 2, pp. 60 – 73, June 1997.

[Pryl98] Prylli, L., and Tourancheau, B. "BIP: A New Protocol Designed for High Performance Networking on Myrinet". *Workshop, PC-NOW, 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, March 1998.

[Rodr97] Rodrigues, S.H., Anderson, T.E, and Culler, D.E. "High Performance Local Area Communication With Fast Sockets". *Proc. of the USENIX 1997 Technical Conference*, January 1997.

[Shan97] Shanley, T., MindShare, Inc. *Pentium Pro and Pentium II System Architecture.* Addison Wesley, 1997.

[Skje94] Skjellum, A., Smith, S.G., Doss, N.E., Leung, A.P., and Morari, M. "The Design and Evolution of Zipcode". *Parallel Computing*, Vol. 20, No. 4, pp. 565 – 596, April 1994.

[Smar92] Smarr, L, and Catlett, C.E. "Metacomputing". *Comm. of the ACM*, Vol. 35, No. 6, pp. 44 – 52, June 1992.

[Tezu98] Tezuka, H., O'Carroll, F., and Hori, A. "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication". *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, March 1998.

[Thek93] Thekkath, C.A., and Levy, H.M. "Limits to Low-Latency Communication on High-Speed Networks". *ACM Transactions on Computer Systems*, Vol. 11, No. 2, pp. 179 – 203, May 1993.

[Yocu97] Yocum, K.G., Chase, J.S., Gallatin, A.J., and Lebeck, A.R. "Cut-Through Delivery in Trapeze: An Exercise in Low-Latency Messaging". *IEEE Symposium on High-Performance Distributed Computing (HPDC)*, August 1997.