The MPI Implementation on NetVM

by

Yanping Gu

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

December 2000

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Dec 15. 2000

Abstract

Advances in network technology now allow an application to reliably transfer data to a remote node without remote CPU intervention. It removes the communication overhead of traditional operating systems by allowing the sender to directly write to the memory of a remote node. However, translating *remote memory* semantics into standard stream-oriented interfaces like MPI is challenging. In this thesis we demonstrate the possibility and efficiency of using the *remote memory* technology as the underlying communication system in our implementation of MPI. Our performance results show significantly improvement on both microbenchmarks and real-world applications compared to LAM over Myrinet. The *remote memory* model is shown to be powerful in implementing MPI semantics.

Contents

List of Tables

List of Figures

To DSG Lab.

Chapter 1

Introduction

In this thesis we describe the issues in building MPIN (MPI on NetVM) and present the design and performance of the system. Implementing MPIN is motivated by the idea of providing a fast and efficient MPI (Message Passing Interface) system using *remote memory* technology, NetVM, as the underlying communication system. We explore the possibility and advantage/disadvantage of building MPI on top of NetVM. In this chapter we give a brief overview of the context in which we build MPIN, the issues raised, and the design approach we have taken.

1.1 Overview

User-mode networking and zero-copy protocols are widely accepted as being essential for reducing software overheads in high-performance communication. User-mode networking gives applications direct access to the network hardware, removing the operating system from the critical path. Zero-copy protocols transfer data directly between user space and the network hardware, eliminating host-CPU data copying for sending and receiving messages.

The *remote memory* model, which allows applications to directly access the memory of a remote node, is equally important for high-performance communication for two reasons. First, the separation of control and data can improve performance because many data transfer operations do not necessarily transfer control. As a result, an application can transfer data to a remote node without remote CPU intervention. Second, this approach provides reliable data delivery since the sender specifies the target address for the transfer. This sender-based memory management, together with a sufficiently reliable network hardware such as the Myrinet, eliminates receive buffer overruns and avoids the need for message buffering, acknowledgements and re-transmission protocols. Several other systems have also explored the advantages of using this approach [6, 8].

The semantics of MPI, however, specify a model of stream-oriented communication. There are two challenges in trying to translate *remote memory* semantics into the MPI stream-oriented model. First, we must separate the transfer of data from the transfer of control messages. Second, we must provide a form of stream-oriented communication.

This thesis demonstrates that the *remote memory* model is powerful in implementing MPI semantics. Since the *remote memory* model allows us to directly transfer data between application memories, there is no need for any intermediate buffering of the data either by the kernel communication system or an MPI daemon. It removes the overhead of traditional network interfaces thereby improving the performance of MPI communication operations.

Comparing to LAM, a widely used MPI implementation based on TCP, our results showed that MPIN greatly improves the performance for both microbenchmarks and real-world applications.

1.2 Synopsis

In the following chapters we present the design and implementation of MPIN and the performance for both the microbenchmarks and real-world applications. Chapter 2 provides background knowledge for the system. It includes an introduction to the MPI standard, NetVM, Myrinet and a review of selected existing MPI implementations. Chapter 3 describes the issues in building MPIN, the overall design approach, and the implementation. The performance of the system and the comparison with LAM over Myrinet are presented in Chapter 4. Chapter 5 is the conclusion of the thesis.

Chapter 2

Background

In this chapter we provide background to better understand the design of MPIN. We introduce the MPI standard, NetVM on which we built MPIN, and the Gigabit networking technology Myrinet. We then review some of the related existing MPI implementations built on various underlying communication systems.

2.1 Message Passing Interface

There are two parallel computing paradigms: shared memory and message passing. Message passing is generally used for parallel machines with distributed memory. Over the last decade many implementations of message passing systems have been developed by different vendors. However, several systems have demonstrated that a message passing system can be efficiently and portably implemented. With this goal, the MPI Forum started the standardization effort for the Message Passing Interface. The MPI standard provides vendors a base set of routines with which they can implement their systems. With this standard, applications built using different implementations of MPI are portable.

We introduce the basic features of MPI that have to be guaranteed in an MPI implementation. We describe the syntax of point-to-point communication operation interface, and the MPI semantics of message passing.

2.1.1 Point-to-point Communication Operation Interfaces

The MPI standard provides interfaces for a rich set of point-to-point and collective communication operations. In this section we describe the syntax of the interfaces for point-to-point communication operations, which form the basis of the MPI collective communication operations. The basic MPI communication mechanism is sending and receiving messages by processes. The interfaces of *Send* and *Recv* operations are defined as:

Send(buf, count, datatype, dest, tag, comm)

Recv(buf, count, datatype, source, tag, comm, status)

The *Send* operation specifies a send buffer in the sender's memory in which the message data is stored. The first three parameters specify the location, size and type of the message. In addition, an *envelope* of the message is also specified by the parameters. An MPI *envelope* consists of a fixed number of fields: *source*, *destination*, *tag* and *communicator*. Certain fields of the MPI *envelope* are included in the parameters of both *Send* and *Recv* operations and are used to distinguish messages. In a *Send* operation, three fields of the MPI *envelope*: *destination*, *tag* and *communicator*, are specified by the last three parameters in the interface.

MPI offers four communication modes for the *Send* operation: *standard*, *buffered*, *synchronous* and *ready*. Programmers decide which mode is suitable for their applications. For a *Send* operation in *standard* mode, it is up to the MPI implementation to decide whether the outgoing message is buffered. When the *standard* send returns, the sender buffer may not yet have been transferred to the receiver so the receiver must avoid reading or modifying the receiving buffer. *Buffer* mode buffers the outgoing message so that *Send* operation can return before the matching *Recv* is posted by buffering the messages at either the sender or, typically, the receiver. The safest mode among the four is *synchronous* mode. It completes only when the message is safely received by the receiver. In general it requires an acknowledgement message from the receiver after it gets the data. *Ready* mode requires that the *Send* operation may only be started after the matching *Recv* has started. Otherwise the operation is erroneous and the outcome is undefined.

The first three parameters of the *Recv* operations specify the receiving buffer, the size and type of the message. The receiving buffer has to be large enough to hold the incoming message data. The following three parameters specify the fields of MPI *envelope* for *Recv*, which are *source*, *tag* and *communicator*. They are used for selecting the matching incoming message. Information about a received message is stored in the *status* parameter.

Send and *Recv* operations can be both blocking and non-blocking. A blocking operation returns only when the operation completes. Users are allowed to reuse resources specified by the operation after that. A *non-blocking* operation, on the other hand, may return before the operation completes. It is completed by either MPI_Test() or MPI_Wait() operations. Any reuse of the resources specified by the uncompleted operation is prohibited.

2.1.2 MPI Semantics

MPI semantics guarantee certain general properties of point-to-point communications. In this section we describe two of them: the order and progress of message passing.

Order of messages is one of the key properties of MPI semantics. MPI messages are not allowed to overtake each other. The MPI *envelope* is used to distinguish between messages. By the MPI ordering rule, if a sender sends two messages with the same *envelope* to a destination, the second matching *Recv* on the destination can not be satisfied by the first message if the first *Recv* is still pending.

The second property of MPI semantics is that MPI guarantees the progress of communication operations. For a pair of matching *Send* and *Recv* operations initiated on two processes, if neither of them is consumed by another matching operation, they are guaranteed to complete.

2.2 NetVM

2.2.1 Overview

NetVM is a novel network interface that supports user-mode, zero-copy, remote network memory without pinning the source or the destination memory. The NetVM prototype is implemented in firmware for the Myrinet and is integrated with the FreeBSD virtual memory system. NetVM's remote-write latencies are 13.3 μs and 53.3 μs for 4-byte and 4-KB transfers respectively, with a maximum throughput of 94.6 MB/s.

NetVM was designed with the following goals:

1. Read and write to/from remote virtual and physical memory.

2. Optionally notify a remote application.

3. Enforce protected operations.

operation	description
register	initialize with NetVM
export	declare segment for data transfer
unexport	revoke exported segment
import	locate remote exported segment
write	write to remote segment
read	read from remote segment
writeP	write to remote physical segment
readP	read from remote physical segment
handler	define user notification handler
arm	invoke handler on notification

Table 2.1: NetVM API

4. Provide reliable delivery.

5. Support out-of-order delivery networks.

2.2.2 API

Table 2.1 lists the key operations available for an application using NetVM.

The application initializes NetVM by calling *register*. It then calls *export* to declare each virtual address range it uses for data transfers. An exported segment is guarded by a 64-bit protection key. When the application calls *import* to locate remote segments, NetVM sends a request to the remote node's kernel driver which responds with the segment address limits and protection key. An application can revoke an *export* by calling *unexport* to immediately disable all remote access to the segment. There is no need to ensure that other applications revoke their *import*s first.

The *write* and *read* operations transfer data to and from a remote segment. The application simply specifies the source and target virtual addresses. NetVM also supports statically pinned transfers. *WriteP* and *readP* are similar to their virtual counterparts except the application specifies physical addresses instead. They also require that both source and target buffers have been statically pinned.

The application notifies a remote application by specifying a *signal* in the *write/writeP* calls. To enable asynchronous notification, the remote application first calls *handler* to register a notification handler and then calls *arm* to prime it. Once armed, NetVM will invoke the handler exactly once when it receives a notifying *write/writeP* to that application. Notifications that arrive when the handler is unarmed are ignored.

2.3 Myrinet

Myrinet is a local area network based on the technology used for packet communication and switching within massively parallel processors [3]. A Myrinet link consists of a full-duplex pair of 1.25Gb/s channels. It is a highly reliable network with a very low error rate. Variable-length packets are supported in Myrinet which uses wormhole routing to deliver the packets. The topology of Myrinet ensures that the capacity of the network grows with the number of nodes.

Each machine has a Network Interface Card (NIC) which contains a LANai network processor, on-board SRAM and three DMA engines. The SRAM is fast and relatively small. It is accessed by the host from the I/O bus. The three DMA engines are used for transferring data from the NIC to the network, from the network to NIC, and between the host and the NIC. In addition, the host can access the SRAM using programmed I/O (PIO). The DMA engines are controlled by the Myrinet Control Program (MCP) running on the LANai network processor. The interesting feature of Myrinet is that the LANai network processor is programmable. Its programmability motivated the design of User-Level Network Interface Protocol [11].

2.4 Related work

We review a selected list of MPI implementations in this section. They differ in both the MPI implementation and the underlying communication system.

2.4.1 LAM

LAM features a full implementation of MPI. It is originated from the Trollius Operating System developed at the Ohio Supercomputer Center for transputers.

LAM runs as a single UNIX daemon on each computer. The daemon is uniquely structured as a nano-kernel and hand-threaded virtual processes. It is transparent to users and system administrators.

The LAM library is written in two layers. The lower layer is the communication system which provides communication primitives for the upper layer. The upper layer is portable and independent of the underlying communication system. LAM uses a Request Progression Interface (RPI) to drive the underlying communication system. RPI is the most sophisticated message-advancing engine in the MPI library. It handles the progress of non-blocking communication requests.

LAM provides two modes of passing messages between processes: LAMD (daemon) mode and C2C (client-to-client) mode. In LAMD mode, all MPI messages are passed between processes via the LAM daemons. C2C mode intends to use the highest performance of the underlying communication system by bypassing the LAM daemons and passing messages directly between processes. LAM includes a TCP/IP implementation of C2C mode. LAMD mode is typically slower than C2C mode in message passing but provides extensive monitoring capabilities.

A set of LAM nodes are started by *lamboot*. LAM supports a dynamic resource environment in which both LAM nodes and MPI processes can be added and removed at runtime with *lamgrow* and *lamshrink*.

2.4.2 MPICH

MPICH is a freely available, complete implementation of the MPI specification, designed to be both portable and efficient. The portability and some features of MPICH are based on three existing systems: P4, Chameleon, and Zipcode.

The software architecture of MPICH supports the goals of portability and high performance. The central mechanism of achieving these goals is the Abstract Device Interface (ADI). ADI separates the communication device dependent part of an implementation from the MPI specific implementation. *Channel interface* is a small subset of interfaces in ADI. It provides a way to transfer data from one process's address space to another's. It can be expanded gradually to include specialized implementation of more of the ADI functionality. ADI provides portability for MPICH. Various versions of MPICH were implemented with their ADIs based on different hardware. ADI was designed to provide a portable MPI implementation, but it can be used to implement any high-level message-passing library.

2.4.3 MPI_BIP

MPI_BIP is a port of MPICH over Myrinet using the BIP (Basic Interface for Parallelism) [10] as underlying communication system. BIP is a network communication interface designed for message-passing parallel computing. The goal of the design was to exploit the high speed Myrinet by bypassing system calls or memory copies. It delivers to the application the maximal performance achievable by the hardware. The BIP interface provides send and receive, blocking or non-blocking communication primitives. There are separate protocols for long messages and short messages.

BIP uses a rendezvous protocol in sending and receiving long messages. With this protocol a receive has to be posted before a send. For short messages BIP stores them in a circular queue on the receiver's side. Send can complete without a receive being posted. A send will block if the queue becomes full.

BIP was designed to achieve zero-copy in MPI. MPI_BIP includes a header in small messages; big messages are transmitted on a different channel than the MPI header. BIP provides only raw flow control capabilities, and relies on the upper layer protocol to implement flow control. A credit-based flow control is implemented by MPI_BIP based on BIP queues to avoid message overflow on the receiver's side.

The BIP interface could be directly used by specialized applications, but the main usage of BIP is through other well established protocol layers such as MPI.

2.4.4 MPI_NP

MPI_NP [1] is a message-passing layer designed for Myrinet. The goal is to reduce host communication overhead by offloading communication related tasks from the host processor based on the assumption that the network processor is able to handle more than average workloads. MPI_NP uses LAM as the MPI library. It implemented a specific Myrinet Control Program (MCP) and a Request Progression Interface (RPI) to LAM.

MPI_NP implemented *channels*, a bidirectional communication path, to communicate between two processes. *Channels* are implemented on the NIC. They consist of buffer space to store the body of messages and rings to hold MPI envelopes. Messages in a channel follow the MPI ordering rules and cannot overtake each other. MPI_NP maintains a global channel queue on the NIC. Similar to BIP, MPI_NP uses a credit-based flow control mechanism to control message overflow on channels.

MPI_NP supports blocking and non-blocking MPI communication operations and various modes for the *Send* operation. It provides three protocols for transferring messages between two processes: full credit, message rendezvous, and eager sending of small messages. The full credit protocol requires that a sender has enough credit before it sends a message. There are two cases that need to be considered in the message rendezvous protocol: unexpected messages when the *Send* operation happens earlier than a matching *Recv* operation, and expected messages when the *Recv* operation occurs earlier than a matching *Send.* Eager sending of small messages is used to transfer small messages quickly. Messages are sent without credit using this protocol. If there is no space on the receiver the message will be dropped. In this case the receiver notifies the sender, and the message will be sent when the sender gets enough credits.

MPI_NP supports zero copy. It uses memory on the NIC as its system buffer. When data being transferred does not fit into the system buffer, MPI_NP uses either sender buffering or address translation and page pinning to achieve zero copy.

Chapter 3

MPIN - MPI on NetVM

MPIN is an implementation of the MPI standard. It uses NetVM as the underlying communication system. The objective of MPIN is to take advantage of NetVM to remove the communication overhead associated with communication stacks in traditional operating systems to achieve better performance for parallel computation using MPI.

The *remote write* interface provided by NetVM changes the semantics of the traditional send and receive communication functions. It allows applications to directly access the memory of a remote node. While achieving better performance

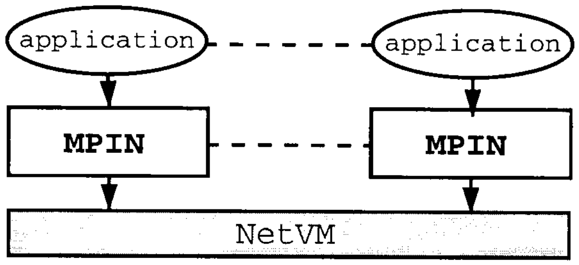

Figure 3.1: System Architecture of MPIN

in transferring data, it makes implementing MPI on top of it a challenge.

In this chapter we introduce the system architecture of MPIN, address the challenges of using NetVM's *remote write* interface to implement MPI communication operations, and describe the design approach taken.

3.1 MPIN System Overview

MPIN is an MPI implementation on NetVM. We introduce the system architecture of MPIN in this section. We also describe the challenges of implementing MPIN using the NetVM interface, and the advantages of doing so.

3.1.1 System Architecture

Figure 3.1 shows the system architecture of MPIN. The arrows show the direction of function calls in each layer. There are three layers shown in the figure:

NetVM, the low layer, provides the communication system on which MPIN is built. The *remote write* interface of NetVM allows direct writes to the memory of a remote node, but the lack of synchronization between the sending and receiving nodes makes implementing MPIN communication operations a challenge.

MPIN layer is the implementation of MPI. It interacts directly with the underlying communication system - NetVM. NetVM is initialized when MPIN starts. Communication operations in MPIN, for example MPI_Send() and MPI_Recv(), are built on top of the NetVM layer using the *remote write* interface. MPIN is responsible to synchronize the communication entities.

The top layer is the application layer. Applications are parallel programs written in MPI. They can run on any MPI implementation without code change.

3.1.2 Data Structures

We briefly introduce the primary MPIN data structures in this section. They consist of a control message area (CMA), a request, and a request list. These three data structures form the basis of the communication functions in the MPIN system. The detailed descriptions for each of them is provided in the following sections.

The key idea of having a *CMA* is that a process can send control messages by writing the messages directly to the appropriate region of the CMA on other process. CMA is implemented as a two-dimensional array in the memory of every process. Each CMA has a designated space for every other process involved in the communication. Each process knows the memory addresses of CMAs on other processes, thus it can send control messages directly to it.

A *request* is a system object to identify communication operations and match the operation that initiates the communication with the operation that terminates it, such as MPI_Test() and MPI_Wait(). A request object identifies various properties of a communication operation such as the send mode, the context, the tag and the destination arguments to be used for a *Send*, or the tag and the source arguments to be used for a *Recv*. In addition, this object stores information that records the current status of the operation.

MPIN creates a request object for each *Send* or *Recv* operation initiated. The request is destroyed when the communication is done. For a blocking operation, this occurs when the function returns; for a non-blocking operation, it is when the communication finishes, and MPI_Test() or MPI_Wait() is called to complete this request.

A *request list* is a double linked list which stores all the pending requests during the execution of an application. It is a global variable in MPIN. Each request, after it is created, is appended to the *request list.* Pending requests keep pointers to the previous and next requests. Within the *request list*, pending requests that relate to each other form lists of special purpose, called *sub list*s or *virtual list*s. We describe *sub list*s and *virtual list*s in Section 3.3.4 and Section 3.4.1. Using the *request list* MPIN manages communication operations executed by an application.

3.1.3 Design Challenges

The challenges in building MPIN are mainly related to synchronization in sending and receiving data. We describe the problems and our design approach in this section.

The MPI model uses the *message* as the fundamental unit for data transfer, the semantics of which is stream-oriented communication. Messages are used for both transferring data and synchronizing senders and receivers. On the other hand, NetVM allows an application to reliably transfer data to a remote node without synchronizing with the receiver process and without the intervention of the remote CPU. Building MPIN requires translating remote memory network semantics into the MPI stream-oriented communication model. There are two challenges in doing so.

First, we know that one advantage of NetVM is that it separates control and data. This separation substantially improves the performance of network operations that require only data flow, but the data transfer happens asynchronously between the sender and receiver. MPIN must provide synchronization between *Send* and *Recv* operations. In doing so, MPIN uses different types of control messages and the CMA data structure on each process to store control messages sent from remote processes.

Second, MPIN must provide a form of stream-oriented communication in transferring control messages and data. We develop a state machine to control the sending and receiving of data, and to ensure the order and progress of communication operations.

The remainder of this chapter describes our design approach and the implementation. Section 3.2 and Section 3.3 explore the details of the design and implementations for the two challenges mentioned above. Section 3.4 focuses on the design when the wildcard MPI_ANY_SOURCE is used in *Recv* operation, which is an extension of the design for normal operations.

3.2 The Control Message Area

A control message area, CMA, is created in each MPI process to store control messages sent to this process. There are different types of control messages in MPIN used in communication operations. We introduce the types of control messages used, the functionality of the CMA, and how CMA works for the key communication operations.

3.2.1 Control Messages

A control message is 32 bytes long, as shown in Figure 3.2. It contains the MPI envelope (source or destination process ID, tag, communicator or context ID), the type of the control message, a count of the elements being sent or received, the MPI data type, the receiver's buffer address, and the index of the remote RFA data structure. The RFA data structure is used for recording *Recv* from MPI_ANY_SOURCE information and will be introduced in Section 3.4.

Control messages are used to synchronize sending and receiving operations

process id	tag	ctrl msg type	count	context id	MPI data type	recver's buf addr	remote RFA indx

Figure 3.2: Control Message

types of control messages
UNUSED
RTR
SC
ENA_ANY
RTR_ANY
UNABLE_ANY
DISABLE_ANY

Table 3.1: Types of Control Messages

in MPIN. Control message types are shown in Table 3.1. Type UNUSED is used in the CMA to mark unused control message slots. RTR (Request To Receive) is a control message sent from the receiver when an *Recv* operation is posted. It informs the sender that the receiver is ready to receive data. SC (Send Complete) is an acknowledgement from the sender that the data has been sent. Upon getting this message, the receiver knows that the data it received is complete. The remaining four control messages: ENA_ANY, RTR_ANY, UNA_ANY, and DISABLE_ANY, are used only when MPI_ANY_SOURCE is used in a *Recv* operation. The description is provided in Section 3.4.

3.2.2 CMA Data Structure

CMA is a data structure which stores the incoming control messages from every process involved in the communication. It is created in the memory of every process when MPIN is initiated. In this section we introduce the data structure and explain

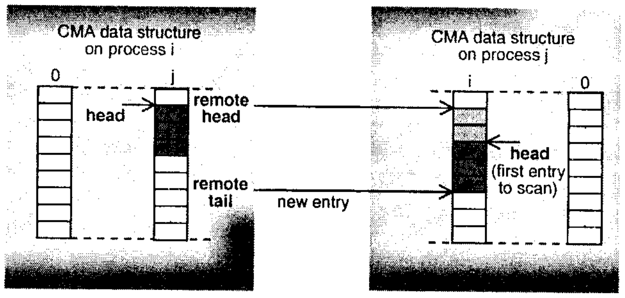
CMA data structure on process i
0
j
head
remote head
remote tail
new entry
CMA data structure on process j
i
0
head (first entry to scan)

Figure 3.3: Control Message Area (CMA)

the operations on it.

Data Structure

The CMA is a two-dimensional array created and initialized in the memory of each process when an MPI application starts. The first dimension is the processes, the second is the control messages sent from each process. Figure 3.3 shows the CMA data structure on processes *i* and *j*. CMA keeps an array for each process, including itself. Each array implements a circular list. Each circular list holds up to 128 control messages. If there are *n* processes involved in the communication, for example, there are *n* circular lists in the CMA. When a process writes control messages to a destination process, it writes the control message to its designated circular list in the CMA on the destination process. In addition to storing control messages, CMA on a process also contains information about the CMAs on other processes it writes to.

CMA works in the following way: consider the CMAs in Figure 3.3, one on process *i* and the other on process *j*. The CMA on process *i* contains a circular list for incoming control messages from process *j* (and vice versa). Process *i* writes outgoing control messages directly into its remote CMA list in process *j*'s CMA. The message is appended to the list. Since process i is the only process adding to this circular list, the pointer to the tail of the circular list can be maintained in process i's CMA, as shown by the *remote tail* in Figure 3.3. Additionally, to prevent process i from overrunning its remote circular list on process j, it is necessary to store a local copy of the *head* pointer on process j, as shown as the *remote head*, which is used to compare with *remote tail* before sending a control message to process j. Since the *head* pointer on process j is being updated by j when it processes control messages, it is necessary for process j to periodically update i's outdated copy of *remote head* pointer. MPIN refreshes this pointer using *remote write* whenever process i's designated circular list in process j's CMA reaches half of its capacity.

Operations on the CMA

There are five primary operations on the CMA: the initialization of CMA, adding a control message to a circular list, scanning for a matching control message, removing a control message, and freeing a CMA.

Initializing a CMA: The CMA is created in the memory of an MPI process when MPI_Init() is called. Each process gets the address of the CMA in every other process.

Adding a control message: When a process needs to send a control message to a destination process, it writes the message to the designated CMA circular list in the destination process. As we know, every process keeps the remote tail (see Figure 3.3) of its designated CMA circular list in other processes, the process can calculate the destination memory address based on the remote tail and the address of the remote CMA, and write the control message using *remote write*.

MPIN assumes that the number of control messages pending in the CMA for each process does not go beyond the threshold of the circular list, which is 128 control messages in total for each process. The system crashes if a process tries to write a control message to the destination CMA when the list is full.

Scanning for a control message: Scanning for a matching control message in a circular list starts from the CMA list head, which is maintained by the local process. It goes through every control message in the list before it reaches a slot marked as UNUSED, or reaches the head of the list when the list is full. The process repeats till the matching control message is found.

Removing a control message : When a control message is matched, it is removed from the CMA list. If the message is the head of the list, its slot is marked as UNUSED. Otherwise, control messages ahead of it in the CMA list are shifted one slot toward the tail. The head pointer is updated in both cases.

Freeing a CMA : The memory space allocated for the CMA is freed when MPI_Shutdown() is called.

3.2.3 A Model for *Send* and *Recv* Operations in MPI

In this section we describe how the CMA works for the key communication operations in MPI. We present a model for *Send* and *Recv* operations and explain how we use the CMA and control messages to synchronize the operations.

We use Figure 3.4 to explain the model. There are two processes in the figure: a receiver and a sender. In each process a buffer and the designated CMA circular list for the other process is shown. A *Recv* operation on the receiver begins the operation by writing a control message RTR into its designated CMA list on the sender (step 1 in Figure 3.4). The receiver then spin-waits on its local CMA list till it gets the control message SC from the sender.

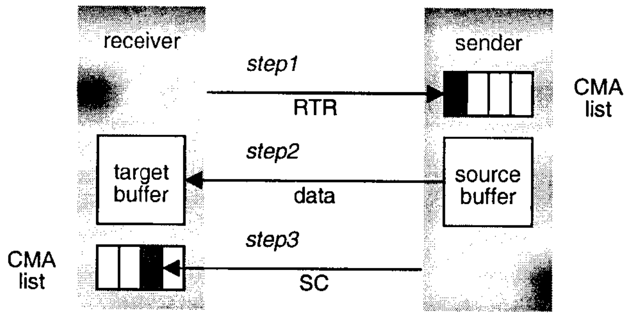

Figure 3.4: Model for *Send* and *Recv* Operation

If *Send* operation is called before the RTR arrives to the sender's CMA, *Send* spin-waits on the CMA list for the matching RTR from the receiver. Once the RTR is found, the sender extracts the destination address and size from it, and ensures that the buffer size is large enough to hold the data. The sender then transfers the data in its source buffer directly to the target buffer using NetVM (step 2 in Figure 3.4). Finally, the sender writes a SC control message to the receiver's CMA to inform it that the data transfer is complete (step 3 in Figure 3.4). The *Recv* operation on the receiver returns when it gets the matching SC from the sender.

This model applies to both blocking and non-blocking *Send* and *Recv* operations, but the implementations differ. The details will be explained in Section 3.3.

The main difference between this model and traditional MPI implementation is that in this model, transferring of control message and data are completely separated.

In the situation where MPI_ANY_SOURCE is used as the source parameter of a *Recv* operation, the receiver does not know where to write the RTR message to when it starts. An extended model which covers this special situation will be described in Section 3.4.

3.3 Order and Progress

MPI semantics guarantee certain general properties of point-to-point communication. This section describes the order and progress of message passing and how they apply to blocking and non-blocking operations. We list the key points which need to be considered to maintain the MPI semantics, and describe the state machine we developed in MPIN for this task.

3.3.1 MPI Semantics for Order and Progress of Message Passing

MPI semantics define the order and progress of message passing, as described in Section 2.1.2. A valid MPI implementation must ensure MPI semantics. The MPI semantics are easy to maintain for blocking operations. The nature of blocking operations retains the execution order and there are no other operations interrupting their progress. Maintaining the ordering of non-blocking operations is more complex, because they return without completing the communication they started. Once started, the progress of non-blocking operations are independent. MPIN must then manage pending operations to ensure that order is maintained, and each operation can make progress properly.

In the design of MPIN, control and data are separated. Three things must be guaranteed to maintain the order and make progress for blocking and non-blocking operations.

- The order of control messages stored in CMA list for each process must be maintained. Control messages must be stored in the order they are written and this order must be maintained until the message is removed from the list.
- The sequence of writing control messages by different operations must be kept

so that message overtaking does not happen.

- For pending identical requests that were created by the same communication operations with same arguments, scanning for matching control messages in the CMA circular list must be ordered. By doing so an operation does not get a control message that is destined to be taken by another operation, assuming that the two operations are waiting for the same control message.

3.3.2 Control Message Matching in MPIN

Control messages are used to synchronize communication operations. In blocking operations, scanning for matching control messages and sending data are done without interruption. A non-blocking operation does not have to be finished before it returns. Considering the situation where there are multiple non-blocking operations initiated but not finished, the sequence for the pending operations to send control messages and scan for matching control messages has to be kept.

Two things must be ensured in MPIN. Firstly, when identical control messages are going to be sent by pending operations, they have to be sent in the sequence that the operations started. Secondly, if pending operations need to scan the CMA list for identical control messages, they also have to do it in this sequence. An operation can not scan the CMA list until the identical operation initiated ahead of it has found the message.

3.3.3 State Machine for MPI Request

In Section 3.1.3 we explained that implementing MPIN on top of NetVM requires a form of stream-oriented communication for transferring control messages and data. In this section we describe our design approach, as well as the state machine we

states for *Send*	description
PENDING	Waiting to be activated
WAITRTR	Scanning CMA list for matching RTR
STARTSC	SC sent
DONE	*Send* completed
states for *Recv*	description
STARTRTR	RTR sent to source process
WAITSC	Scanning CMA list for matching SC
DONE	*Recv* completed

Table 3.2: Request States for *Send* and *Recv* Operations (Normal Case)

developed for this task.

Request State

The request state is a property of a request object. It records the current status of the request. Table 3.2 shows the request states for *Send* and *Recv* operations, and a brief description for each state. MPIN determines the state transition for *Send* and *Recv* operations. It will be explained in the following section.

The request states listed in Table 3.2 fall into two categories:

- *static state* : PENDING, STARTRTR, DONE
- *automatic state* : WAITRTR, STARTSC, WAITSC

Processing a request in static state does not change the state nor push the communication forward. A request is usually put in static state when it is ready to move to the next state, e.g., to start scanning for a control message, but has to give way to another pending identical request which is in the same state. Messages in static state can be activated and moved to the next state by other pending requests ahead of them in the *request list.*

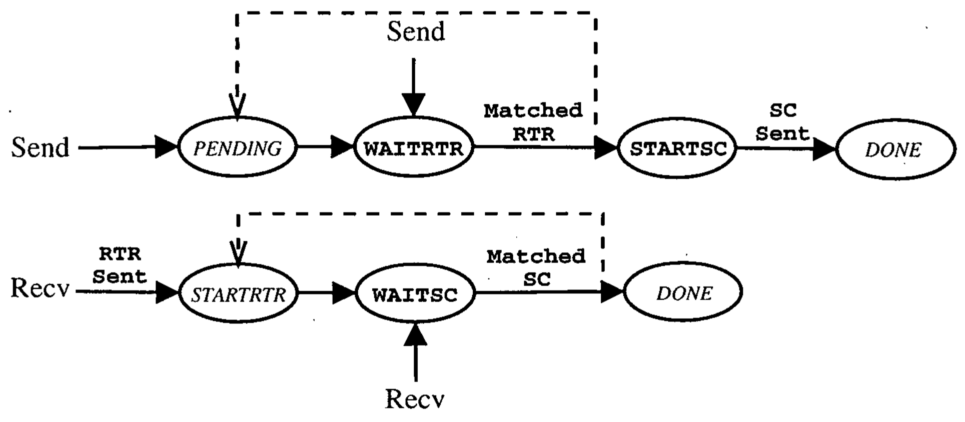

Figure 3.5: State Transition of Requests

On the other hand, when MPIN processes a request in an automatic state, it extracts information from the request object and pushes forward the communication according to the current state.

State Transition

Figure 3.5 demonstrates the state transition process of *Send* and *Recv* operations. Two fonts are used in the figure to distinguish static states (PENDING, STARTRTR, DONE) and automatic states (WAITRTR, STARTSC, WAITSC). We use the solid arrows to show the progressing direction of a request. The dashed arrows point out the activation points where a request in a static state is activated by a request in an automatic state which just finished a state transition. The requirements of state transitions of automatic states are also shown.

In this figure, a *Send* operation could start from either PENDING or WAIT-RTR state. Normally, when a request for a *Send* operation is created, if there are no identical requests pending ahead of it, or all of the pending identical requests are in state STARTSC or DONE, it starts in the state WAITRTR. Otherwise, it starts in PENDING. For the *Recv* operation without wildcard MPI_ANY_SOURCE in use, the operation starts by sending an RTR message to the sender. The state of the request is put in WAITSC when there is no identical requests pending, or all of the pending identical requests are in state DONE. Otherwise, the request state starts with STARTRTR. Processing requests of both operations moves the state in the direction of the solid arrows till it reaches state DONE. The state transition for *Recv* operations with MPI_ANY_SOURCE is explained in Section 3.4.

State transition may be triggered by remote or local events. It may depend on the behavior of a remote process, for example, when a control message is sent. If the request is in a state of waiting for this control message, the state transition happens when this control message arrives in the CMA and is found in the processing of the request. In Figure 3.5, a *Send* request in state WAITRTR moves to STARTSC when the matching RTR is found in the CMA list, and a *Recv* request in state WAITSC moves to DONE when the SC is found. State transitions may also happen on a local event. For example, the state STARTSC of a *Send* operation moves to DONE after the SC control message is sent.

Requests in automatic states are responsible for activating identical requests pending after them and which are in static states. Usually the activation happens at the point of state transition of a request in an automatic state, as shown with the dashed arrows in Figure 3.5. When a *Send* request gets the matching RTR, its request state moves from WAITRTR to STARTSC. The identical request pending after it, which must be in state PENDING at this moment, is activated and the request state moves to WAITRTR. For the same reason, a *Recv* request in state WAITSC, when it transits to state DONE, activates its following pending identical request, which must be in state STARTRTR, to state WAITSC.

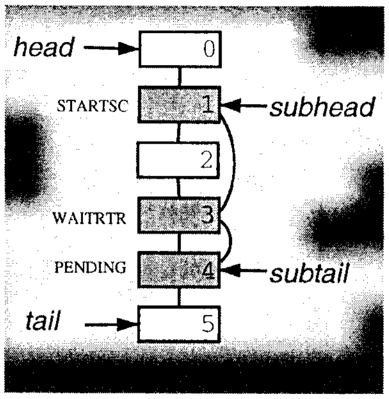

Figure 3.6: Sub List in Request List

3.3.4 Request Management

MPIN stores all the pending requests in the *request list* during the execution of an application. We introduce a special list in the *request list* called a *sub list*, and describe how MPIN manages the progressing of pending requests stored in the *request list*.

Sub List

The *sub list* is a doubly linked list which links pending identical requests in the *request list*. The reason we introduce this special list is that when there are identical requests pending in the *request list*, the dependency among them has to be considered to maintain the order of the operations they represent. *Sub list*s are used to do that.

Each pending request keeps pointers to the previous and next requests in the *request list*. In addition to these pointers, a request which belongs to a *sub list* also keeps pointers to its previous and next requests in its *sub list*.

Figure 3.6 shows requests in the *request list* at some point in the execution of an application. There are six requests pending in total. The shaded boxes represent identical requests which belong to a *sub list.* The *head* and *tail* pointers in the figure are for the *request list*, while *subhead* and *subtail* pointers keep track of the start and end of the *sub list.*

Progress of Requests

Progressing of a request starts from the request being appended to the *request list*, and stops when it is removed. In this section we describe how MPIN processes the pending requests in three steps:

- After a request is appended to *request list*, all pending requests ahead of it are searched to find out if it belongs to a *sub list.* If so, it is appended to the *sub list.*

- MPIN processes each request in the *request list* according to its current request state. Figure 3.5 shows the state transition for *Send* and *Recv* operations. Using the example in Figure 3.6 again, requests 1, 3, and 4 are identical requests of *Send* operations which belong to a *sub list.* Their states are STARTSC, WAITRTR, and PENDING respectively. Request 4 is blocked by request 3 because only one request is allowed to scan for RTR at a time. According to Figure 3.5, when request 3 moves to state STARTSC, it activates request 4 from state PENDING to WAITRTR.

- The state of a request eventually moves to DONE. At this time, a request of a blocking operation is removed from the *request list*; if the request represents a non-blocking operation, it is removed from the *request list* when an MPI_Test() or MPI_Wait() function is called. Pointers to both *request list* and *sub list* are updated accordingly.

Message passing in MPIN progresses with the state transition of requests, and the states are only transited by calls into MPI communication functions. When a communication function is called, MPIN goes through all the pending requests in the *request list* and processes them according to their current state. For each pending request, doing so is equal to getting the communication function called once more. It is necessary to do so in MPIN because sometimes a pending request might block requests following it which are supposed to move on no matter what state this request is in. Going through all the pending requests and processing them whenever possible is the only way for MPIN to guarantee the progress of communication functions.

3.4 Wildcard in Receive Operation

MPI allows a *Recv* operation to accept messages from an arbitrary sender if the *Recv*'s source parameter is MPI_ANY_SOURCE. A *Send* operation from any process which has the matching tag and context parameters is considered to be a matching operation. If there are multiple matching *Send* operations, only one can be matched.

In a sender-initiated system like LAM, message passing can happen without the matching *Recv* being posted. Messages from different sources are queued up on the receiver's side. Source matching happens locally on the receiver's side when a *Recv* is posted. For a *Recv* operation with MPI_ANY_SOURCE, the first message in the queue is picked up. Source matching in MPIN is difficult. Since NetVM is used as the underlying communication system, system copying is removed. Instead of being queued up in the receiver's buffer, message data is written directly into the receiver's memory when the matching *Recv* is posted. In MPIN, source matching must be finished before data is sent to the receiver. To do this, the receiver must cooperate with the potential senders using control messages. Data transfer then happens between the *Recv* and the *Send* on the matched sender.

The model we describe in Section 3.2.3 applies only to the case in which the source parameter of a *Recv* operation is specific. In that model, a *Recv* operation always starts by sending an RTR control message to the source. When MPI_ANY_SOURCE is used, the receiver does not know in the beginning which source it is going to send the RTR to. We extended the old model to cover the situation where MPI_ANY_SOURCE is used as the source parameter in *Recv* operations.

We describe our design in three steps: data structures, state transition, and request management.

3.4.1 Data Structures

Source matching is done through control information exchanged between the receiver and the potential senders. MPIN creates two data structures: RFA(Receive From Any source), and SRFA(Sender's information for Receive From Any source). The receiver creates an entry in its RFA when a *Recv* operation is initiated. In addition, the receiver writes an entry to the SRFA on every other process for this *Recv*. We describe the new data structures and control messages used when MPI_ANY_SOURCE is specified as the source parameter in *Recv* operations, and a new model based on them.

Data Structures

RFA and SRFA are created in the memory of each process and are used for the receiver and sender to exchange control information. Figure 3.7 shows that both the RFA and SRFA are implemented as arrays. RFA is used by the receiver and SRFA by the sender. There are two fields in an element of RFA. One is the tag parameter of the *Recv* operation, the other is an array of flags for each process. The address of the RFA on each process is exported to other processes when MPIN starts. An element of the SRFA consists of four integers: the destination and the tag parameters of the *Send* operation, the index of the RFA entry on the destination process, and a flag. The usage of flags in both RFA and SRFA are described later.

In addition to the RFA and SRFA data structures, four new types of control messages are used; Table 3.1 shows them: ENA_ANY, RTR_ANY, UNA_ANY and DISABLE_ANY. ENA_ANY is used for *Recv* with MPI_ANY_SOURCE to notify the potential senders. RTR_ANY works the same as RTR except that it is sent from a *Recv* operation with MPI_ANY_SOURCE. UNA_ANY is sent from a *Send* operation to let the receiver know that the sender does not intend to send data to it. DISABLE_ANY is used to revoke the information that a *Recv* operation with MPI_ANY_SOURCE has broadcast to the potential senders.

We develop a new model for the situation where MPI_ANY_SOURCE is used in *Recv* operations. This model is an extension of the one we describe for the normal case. We use the data structures and new control messages introduced in this section to describe this model.

A Model for *Recv* with MPI_ANY_SOURCE

We use Figure 3.7 to explain the model. There are two processes in the figure: a receiver and a sender. On the receiver's side, an RFA and a CMA list for the sender's process are shown. The SRFA and a CMA list for the receiver's process are shown on the sender's side. The exchange of control information and data between

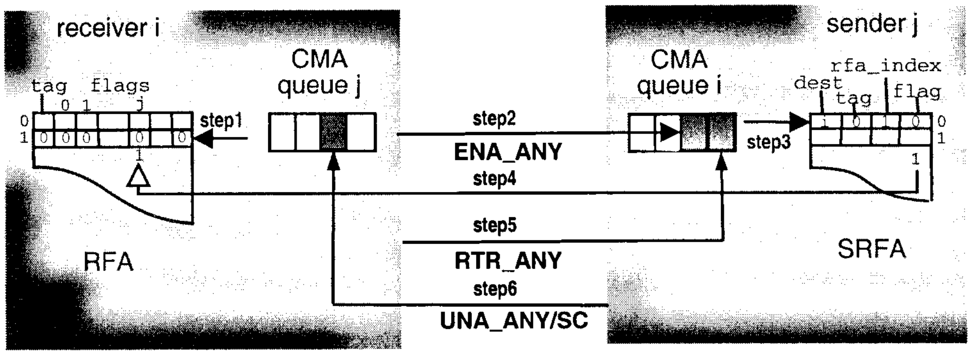

Figure 3.7: *Recv* from MPI_ANY_SOURCE

these two processes uses NetVM *remote write.*

When a *Recv* operation with MPI_ANY_SOURCE starts, the tag parameter and an array of flags are added as a new entry to the local RFA. All the flags are set to 0 initially (step 1 in Figure 3.7). The *Recv* operation then writes an ENA_ANY control message to the CMA list in every process (step 2 in Figure 3.7). A flag in an RFA entry will be turned on to 1 by the corresponding sender if the sender wishes to send data. The receiver scans the flag array to find out which flag has be turned on. The first corresponding sender is picked up and the flag turned off.

When a process finds an ENA_ANY in its CMA list it adds a new entry to its SRFA (step 3 in Figure 3.7). The new entry consists of the destination and the tag parameter of the *Recv* operation, an index of the RFA entry on the receiver, and a flag 0. The first three integers are carried by the ENA_ANY control message.

When a *Send* operation is called, the sender checks the SRFA to see if there is a *Recv* from MPI_ANY_SOURCE entry that has matching destination and tag, and if the flag is 0. If so, the *Send* sets the flag to 1, then uses NetVM to write a 1 to the corresponding flag in the RFA entry on the receiver process (step 4 in Figure 3.7). The destination address of the flag is calculated by the remote RFA's address, the index of the RFA entry, and the ID of the sender's process.

The *Recv* operation scans the array of flags in the RFA entry. The first potential sender is picked up if the flag for the sender is 1. The receiver then turns the flag to 0 and sends an RTR_ANY control message to the sender (step 5 in Figure 3.7).

When the sender finds the RTR_ANY control message in its CMA list, it sets the corresponding SRFA flag to 0 and writes the data and a SC control message to the receiver (step 6 in Figure 3.7). At this point, the *Send* operation completes and returns. Upon getting the SC control message, the *Recv* from MPI_ANY_SOURCE also completes.

MPIN does not remove the RFA entry when the *Recv* operation returns. This reduced communication overhead of broadcasting ENA_ANY control messages for every *Recv* by having identical *Recv* from MPI_ANY_SOURCE use the same entry. However, it left a problem that there might be RFA flags set by previous senders that have already completed.

We solve this problem as follows. When a receiver thinks that the sender is interested in sending data to it, and the sender has already been satisfied, it sends an RTR_ANY to the sender. The sender responds to it by sending an UNA_ANY back to the receiver.

Upon getting the UNA_ANY control message, the *Recv* operation knows that this sender is not interested in sending data to it anymore. It then starts to scan the RFA entry for the next potential sender.

The DISABLE_ANY control message is not displayed in the figure. It is used when the RFA does not have a space for a new entry. The oldest entry is removed and a DISABLE_ANY control message carrying information about the removed entry is written to every process. When a DISABLE_ANY control message is found in the CMA list, a process cleans up the corresponding SRFA entry. If a new *Recv* from MPI_ANY_SOURCE starts and finds that there is no entry for it, it will create a new one.

This model works for both blocking and non-blocking communication operations where the source parameter of *Recv* operations can be either specific or a wildcard. Suppose there are N processes in the communicator. In the normal case, the number of control messages required for a *Recv* from MPI_ANY_SOURCE to finish is:

N(ENA_ANY) + 1(Setting RFA flag) + 1(RTR_ANY) + 1(SC) = N + 3

In the best case where an RFA entry already exists, the ENA_ANY control message needs not to be sent. The total control messages involved are 3.

We extended the state machine to cover the situation where the source parameter of *Recv* operation is a wildcard. The new state machine is based on the model we just described.

State Transition

New states are added to the old state machine. Table 3.3 lists all the states with brief descriptions. States for a *Recv* operation are also categorized as for normal requests (PENDING_NOR, STARTRTR, WAITSC) and for requests with wildcard source parameters (PENDING_ANY, SOURCE_MAT, SOURCE_MAT_AUTO, STARTRTRA, WAITSC_UA). State DONE is for both cases.

Figure 3.8 shows the state transitions. Two fonts are used as in Figure 3.5 for

states for *Send*	description
PENDING_NOR	Waiting to be activated
WAITRTR	Scan CMA list for matching RTR
WAITRTRA	Scan CMA list for matching RTR_ANY
STARTSC	SC sent
DONE	*Send* completed
states for *Recv*	description
PENDING_ANY	Waiting to be activated. For *Recv* with MPI_ANY_SOURCE
PENDING_NOR	Waiting to be activated. For normal *Recv*
SOURCE_MAT	Scan RFA for potential source
SOURCE_MAT_AUTO	Scan RFA for potential source
STARTRTR	RTR sent to source process
STARTRTRA	RTR_ANY sent to potential source process
WAITSC	Scan CMA list for matching SC
WAITSC_UA	Scan CMA list for matching SC or UNA_ANY
DONE	*Recv* completed

Table 3.3: Request States for *Send* and *Recv* Operations (MPI_ANY_SOURCE)

static states and automatic states respectively. Solid arrows show the progressing direction of a request. Dashed arrows show the activation points where static states for a *Recv* request with a specific source parameter are activated. Static states of a *Recv* with MPI_ANY_SOURCE request are activated in a different way and will be described later.

Request Management

MPIN processes pending requests in the *request list.* We introduced the *sub list* in Section 3.3.4. *Recv* requests in a *sub list* are requests with a specific source parameter. For a *Recv* with MPI_ANY_SOURCE request, the *sub list* it belongs to is only determined when a matching source is found from one of the potential sources. For requests corresponding to *Recvs* with MPI_ANY_SOURCE, another special list called a *virtual list* is created inside the *request list.* Next we describe the *virtual list* and how MPIN processes requests in the *virtual list.*

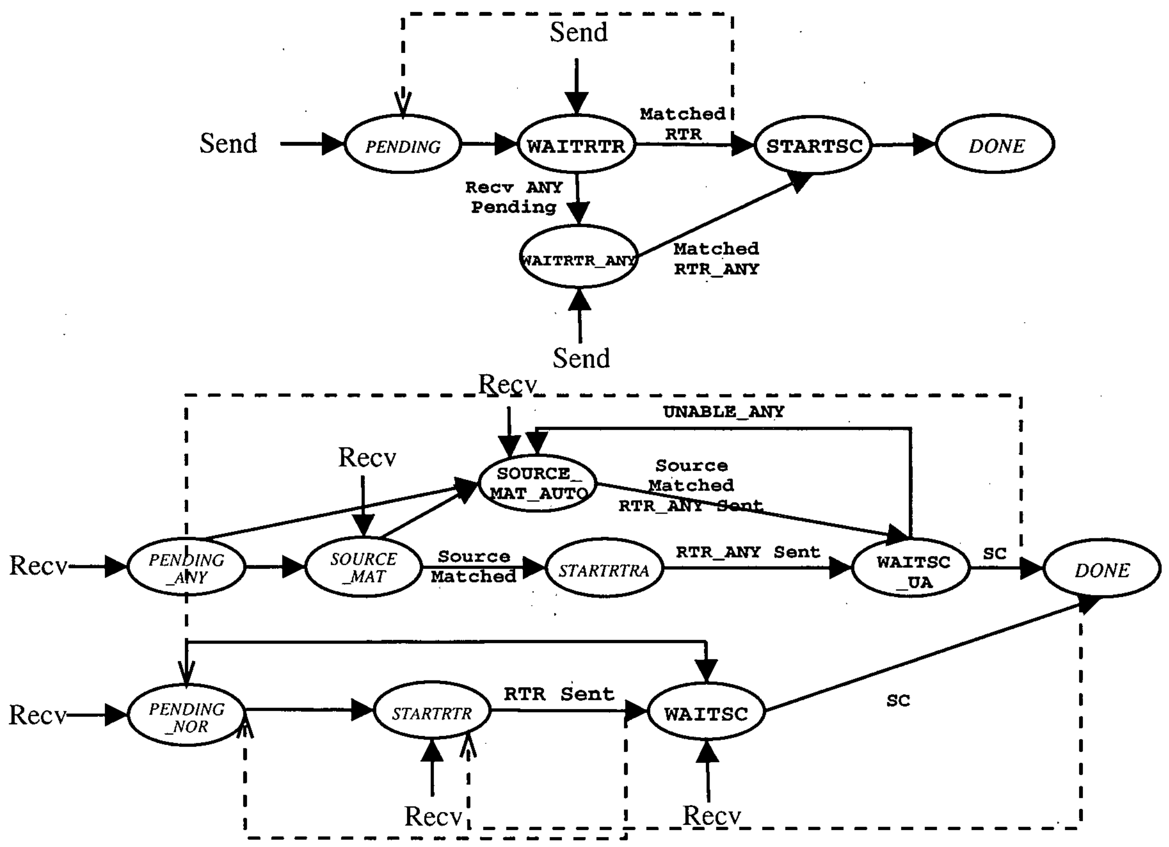

Figure 3.8: State Transition for *Recv* from MPI_ANY_SOURCE

Virtual List

The *virtual list* is a special list created in the *request list* and used to store identical pending *Recv* with MPI_ANY_SOURCE requests. For those requests, source matching happens before the data transfer. Before the real source is determined, the request can not be added to a *sub list*. After the source matching, the request becomes a normal *Recv* request and could be added to one of the *sub list*s pending ahead of it in the *request list*. Thus, its existence has to be known by the other pending requests in order to maintain the order of all the pending operations.

A *virtual list* connects identical *Recv* requests with MPI_ANY_SOURCE. In addition, it also connects a *Recv* request with MPI_ANY_SOURCE and the requests that are the tails of *sub list*s with matching tag and context parameter pending in front of it. When the real source is determined, the *Recv* request becomes a normal one and will be added to a *sub list* pending ahead of it if there is one.

Figure 3.9 shows a *virtual list* in a *request list*. Boxes with the same shades represent identical *Recv* requests in a *sub list*. Requests 1 and 6 belong to a *sub list*, and requests 2, 3, and 5 belong to another *sub list*. Boxes labeled *ANY* indicates identical *Recv* requests with MPI_ANY_SOURCE in use (requests 4, 7, 8). They form a *virtual list*. From the figure we can see that besides the pointers connecting the *ANY* requests, requests 4 and 7 keep pointers linking to the tails of the *sub list*s pending ahead of them, assuming that the requests in the *sub list*s have matching tag and context parameters with the *ANY* requests in the *virtual list*.

Progress of Requests

The progressing of *Recv* requests with MPI_ANY_SOURCE happens with the state transition shown in Figure 3.8. When a request is created and appended to the

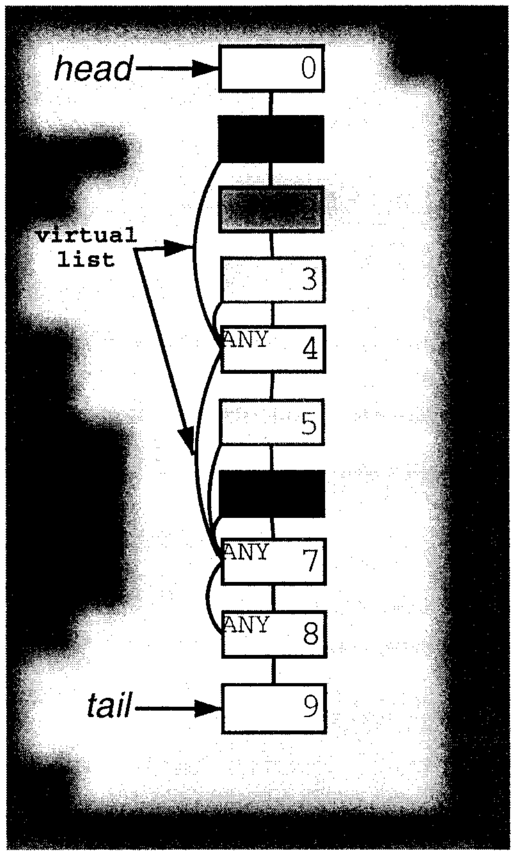

Figure 3.9: Virtual List in Request List

request list, it is also appended to the *virtual list* it belongs to. MPIN processes each request according to its current request state. In MPIN, the real source of the request is determined only when the request state reaches state DONE, then the request becomes a normal *Recv* request. It is removed from the *virtual list* and joins the *sub list* it belongs to. Processing of the request continues as it does for a normal *Recv* request. The request is eventually removed from the *request list.*

In transiting a static state of a *Recv* request with MPI_ANY_SOURCE, the pending requests in *sub lists* ahead of it and connected to it in the *virtual list* suggest which state this request can move forward to. The suggested state is put into a temporary variable. Every request ahead of it in the *virtual list* has to make a suggestion, and the lowest state is chosen for the request to continue from when the request is processed.

3.4.2 Limitations and Future Work

MPIN is not a fully implements MPI system. The current prototype of MPIN implemented basic blocking and non-blocking *Send* and *Recv* operations. Some of the collective communication operations such as MPI_Bcast() and MPI_Barrier() are implemented based on the point-to-point operations. Communications in MPIN are currently limited to one communicator. Wildcard in *Recv* operations is implemented only for MPI_ANY_SOURCE. But our design can be easily extended to implement *Recv* operations with MPI_ANY_TAG.

Our future work can be summarized as follows:

- Fully implement all four modes for *Send* operations;
- Implement associate operations such as MPI_Probe();

- Implement the rest of the collective operations;
- Implement the wildcard: MPI_ANY_TAG;
- Extend the message passing to inter-communicator communication.

Chapter 4

Performance

MPIN was developed and tested on the FreeBSD operating system. We compare the performance of MPIN to LAM running on TCP over Myrinet. We compare our system with LAM because LAM is a widely used MPI implementation running on TCP/IP. MPIN uses NetVM as the underlying communication system which allows direct access to a remote node's memory without TCP intervention.

We present performance results for both microbenchmarks and real applications. Microbenchmarks test the latency and throughput for various sizes of data. A MPI application, *RSimp*[5], was used to test the performance of a real application running on MPIN and LAM. The results show that compared to LAM, MPIN improves the performance for both the microbenchmarks and the real-world application.

4.1 Experimental Setup

Our experiments were conducted on a cluster of 266-MHz Pentium Pro PCs with 128-MB of memory running FreeBSD 2.2.2 and with a page size of 4-KB. The

Data (bytes)	Latency(MPIN) (μs)	Latency(LAM) (μs)	Throughput(MPIN) (MB/sec)	Throughput(LAM) (MB/sec)
4	29.91	201.66	0.13	0.02
8	30.00	202.08	0.27	0.04
16	30.09	202.21	0.53	0.08
32	30.42	209.14	1.05	0.16
64	32.10	210.83	2.00	0.32
128	33.78	254.66	3.78	0.53
256	35.00	221.46	7.31	1.22
512	37.36	233.41	13.68	2.31
1024	43.30	257.33	23.71	4.17
2048	52.85	320.09	38.80	6.81
4096	69.89	459.50	58.43	9.42
4100	78.85	460.36	51.53	9.43
5120	90.17	492.46	56.97	11.15
6144	99.59	533.79	61.85	12.51
7168	108.52	578.25	65.91	13.56
8192	116.58	649.54	70.16	13.96

Table 4.1: Microbenchmarks for MPIN and LAM using MPI_Send and MPI_Recv Operations

PCs are connected by the Myrinet network that uses 33-MHz LANai 4.1 network processors with 1-MB on-board SRAM.

Timing in the experiment is measured by counting machine cycles and dividing it by the clock speed of the CPU to convert it to microseconds.

4.2 Microbenchmarks

In this section we present microbenchmarks for MPIN. We tested the latency and throughput of the system for various sizes of data, and compared the performance with LAM running on TCP over Myrinet.

The MPI blocking *Send* and *Recv* operations are used in testing microbenchmarks. The data set we chose ranges from 4 bytes to 8K bytes. Data being sent started at page boundary. The largest size we used covers two pages.

Table 4.1 shows the latency and throughput of MPIN and LAM for various sizes of data. The results are also shown in Figure 4.1 and Figure 4.2 for latency and throughput respectively.

4.2.1 Latency

We tested one-way latency using MPI *Send* and *Recv* operations. Two nodes are used in the test with one node sending data and the other node receiving. The procedure is repeated 1000 times. The median value is reported.

In MPIN, when MPI_ANY_SOURCE is not used, a receiver starts the communication by sending a RTR control message to the sender. The communication on the receiver's side finishes when the matching control message SC from the sender is found in the CMA list. A *Send* is spin-waiting on the CMA if a matching *Recv* hasn't been posted. The blocking time of *Send* depends on the starting time of *Recv*, thus it should not be added to the overall latency. On the other hand, LAM is not receiver-initiated. The sender does not need the *Recv* operation to be posted before it sends the data.

To get the correct measurement, we use a pingpong test to measure latency. The method we used is shown in Figure 4.3. One node starts the timer, sends the data using MPI_Send(), then uses MPI_Recv() to receive the response. The other node sends the data back as soon as it gets it. The timer starts on the first node before MPI_Send(), and stops when the MPI_Recv() returns. We get the one way latency for a specific size of data by dividing the difference of the two times by two.

From Figure 4.1 we see that the latency of MPIN is more stable than that of LAM. As the data size gets bigger, the time spent for sending it in LAM grows faster than MPIN. The figure shows a drop in LAM when the data size is 256 bytes.

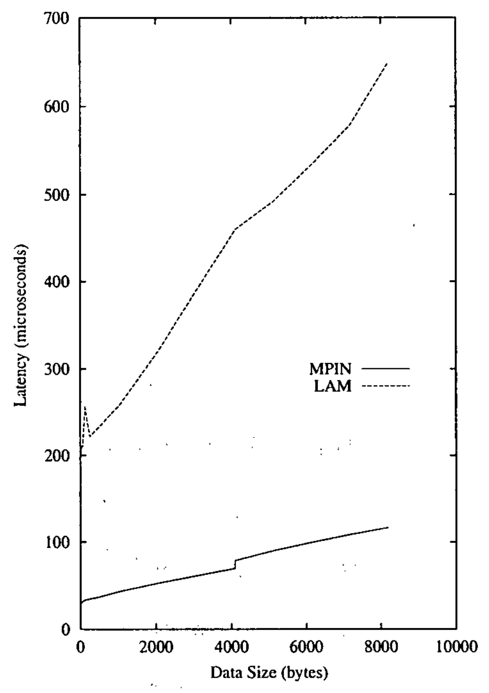

Figure 4.1: Comparison of Latency for MPIN and LAM

After that, the latency grows constantly. In MPIN, there is a small delay when data crosses the page boundary, which is at 4K bytes.

4.2.2 Throughput

Throughput was also tested for different sizes of data on LAM and MPIN. For a specific size of data, throughput is calculated by dividing the data size by the time spent in sending it. The time of sending data is measured using the same method we used in testing latency.

Figure 4.2 shows that compared to LAM, MPIN has greatly improved the throughput for the various sizes of data used in the test. The throughput for 8K bytes data is 70.16MB/second in MPIN. For both MPIN and LAM, as the data size grows, the network gets more saturated and thus has better throughput. There is a

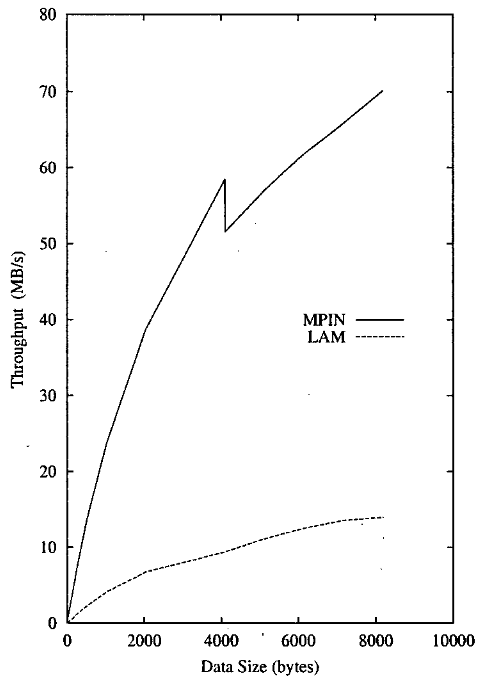

Figure 4.2: Comparison of Throughput for MPIN and LAM

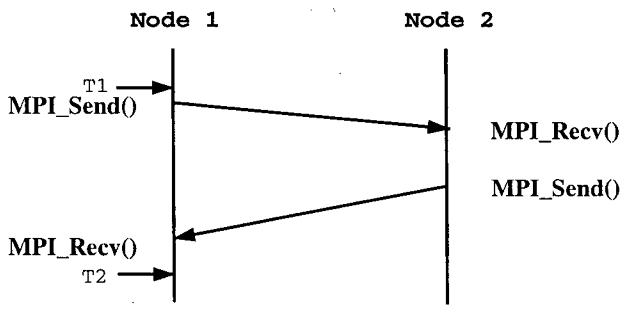

Figure 4.3: Measurement of Latency/Throughput

drop when the data crosses the page boundary at 4096 bytes. After that point the throughput goes up again as the data size grows to 8K.

In the figure, LAM on TCP over Myrinet shows a much lower throughput than MPIN. The reason is that the TCP uses system calls which go through the kernel and incurs extra copy to and from user/kernel memory. LAM internal buffering also has a big contribution to the overhead. LAM can return from MPI_Send after buffering the message. Messages could stay in the system buffer before they get sent out.

4.3 Receive From MPI_ANY_SOURCE Test

We measured MPIN's performance of *Recv* operation with MPI_ANY_SOURCE and compared it with LAM. The test runs with one receiver and various number of senders. The receiver calls *Recv* from MPI_ANY_SOURCE to receive data from one of the senders. The test runs with 1, 2, 3, 4 senders respectively. The results are shown in Figure 4.4. We measured the time the receiver takes to receive the data in each case. The result shows that even though more control messages are involved in *Recv* from MPI_ANY_SOURCE in MPIN, the performance is comparable with LAM.

4.4 Application-level Tests

To test MPIN we used an application from the area of computer graphics, specifically a polygonal mesh simplification algorithm. The purpose of these algorithms is to reduce the complexity of polygonal models so that computer graphics hardware is able to render them at interactive rates. These algorithms are both memory and

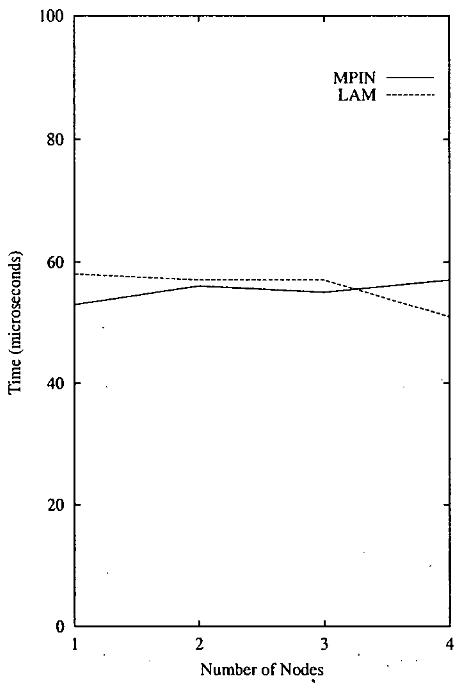

Figure 4.4: Comparison of *Recv* from MPI_ANY_SOURCE for MPIN and LAM

computation intensive and thus are ideal candidates for parallelization.

We used a parallel version of an algorithm called *RSimp* [5] to evaluate the performance of MPIN. The algorithm operates by obtaining a very coarse approximation of a polygonal model and then iteratively refining it until the required level of detail is obtained. For our purposes the algorithm can be viewed as consisting of a priority queue and an n-ary tree where a node can have two, four, or eight children.

The algorithm begins with a root node having eight leaf nodes. The leaf nodes are inserted into the priority queue. During each iteration a leaf node is removed from the priority queue and is internalized by creating two, four, or eight leaf nodes. These leaf nodes are then inserted into the priority queue. The algorithm stops when the required number of leaf nodes are obtained.

Several modifications were required to transform the sequential version of *RSimp* into its current parallel incarnation *PRSimp* [4]. *PRSimp*'s design is based on the master worker architecture. There is a single master node that controls the entire simplification process and partitions out work to the worker nodes. The first main modification consisted of adding MPI communication operations to enable the master node to send and receive data from the worker nodes. The second modification involved modifying *RSimp* such that it could determine whether it was a master or a worker node and performing as such. A master node performs all the operations performed in *RSimp* plus additional operations of sending and receiving data from the worker nodes. The worker nodes have considerably less responsibility. Their main task is to receive a node, expand it into 2, 4, or 8 leaf nodes and return them to the master.

The changes to the simplification process are as follows. Instead of retrieving one node from the priority queue, expanding it, and inserting the new leaf nodes back in, the master node retrieves n nodes from the priority queue; where n is the number of worker nodes. The master node then sends a node to each of the workers to be processed. It then removes one more node from the priority queue, processes the node itself, and inserts the sub nodes back in. Then it receives the processed nodes from the workers and inserts the results back into the priority queue. This process is repeated until the required number of leaf nodes is reached.

The application uses non-blocking *Send* in distributing the work to workers and non-blocking *Recv* in receiving results. Workers get the distributed work, do the computation, and send the results back. The workers use blocking communication calls.

4.4.1 Measurements

We report three aspects of the performance of running *PRSimp* on MPIN and LAM: total time, latency and overhead. The measurements are obtained on the master node only, because its performance determines the overall application performance. The performance for the sequential version *RSimp* is provided as a reference.

The running time of the application includes reading data from the disk, simplifying data (running the algorithm), and writing the results back to disk. The communication between the master and workers happens only in the second phase.

Figure 4.5 compares the performance of running *PRSimp* on MPIN and LAM over Myrinet. The total time, latency and overhead of running *PRSimp* are shown in this figure.

The total time we report is the time of the simplication phase. It includes the time of performing the computation, the latency and the overhead of the communication functions.

The time of latency and overhead forms the communication time. The timer starts before the communication operations are called and stops after they finish. Latency covers the time of calling MPI_Test to test if the *Send* or *Recv* operation finishes. The time for each MPI_Test is totaled up as the overall latency of the communication operations in the application. Overhead is obtained by subtracting the communication time of the application by the latency.

By comparing the performance in running *PRSimp* on MPIN and LAM, we get the overall performance as well as the breakdown of latency and overhead of the communication operations in each system. With the message size of 10K to 16K bytes sent from the master node, MPIN achieved a very low latency in waiting for the communication operations to complete, and the overall communication time is

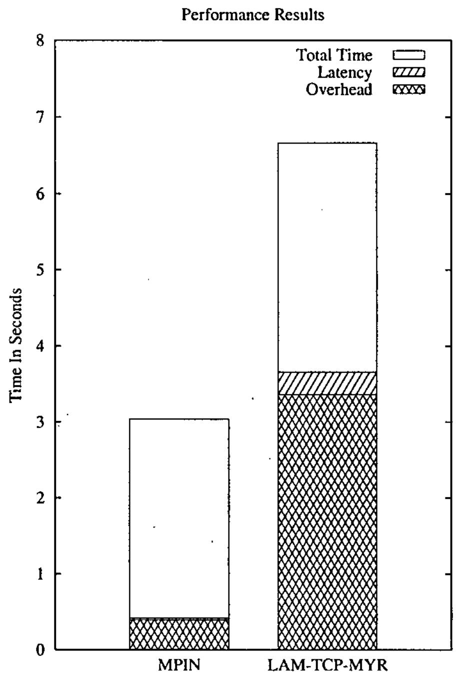

Figure 4.5: Comparison of Running *PRSimp* on MPIN and LAM

considerably lower than LAM. As a result, the overall cost of running *PRSimp* on MPIN is improved about 54% compared to LAM over Myrinet.

Chapter 5

Conclusions

MPIN is an MPI implementation on Myrinet using the NetVM as the underlying communication system. It aims to provide parallel applications an MPI interface based on fast *remote memory* technology.

Several recent research projects have explored the benefits of using programmable network interfaces provided by current gigabit networks [3]. These benefits include lower message overhead possible when interfaces are directly accessible at user level [6, 9, 13], lower large-message latency possible when interfaces fragment and pipelining data transfers between host memory and the network [2, 7, 14], higher throughput possible when fragmentation and pipeline are adaptive to message size [10, 12], and lower overheads possible when interfaces implement sender-based flow control [6]. Our work differs from each of these projects in that we focus on functionality at a higher-level than the network protocol layer.

This thesis focuses on the design and implementation of blocking and non-blocking MPI point-to-point communication operations using the *remote write* interface provided by NetVM. Based on different semantics of *remote write* and MPI, our design mainly focused on providing a synchronization mechanism for message passing, and translating the *remote write* semantics to stream-oriented semantics. MPIN creates a Control Message Area (CMA) on each process, and uses different types of control messages to synchronize message passing. A state machine was developed to control the progress of message passing between processes, and to ensure the order and progress of communication operations.

MPIN is compared with LAM running on Myrinet. Performance showed that MPIN improved the performance of both the microbenchmark and real world applications. This thesis proved that the *remote memory* technology can be a powerful tool in implementing a stream-oriented MPI library.

Bibliography

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

[2] Darrell Anderson, Jeff Chase, Syam Gadde, Andrew Gallatin, Ken Yocum, and Mike Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the 1998 USENIX Technical Conference*, June 1998.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L Seitz, J. N. Seizovic, and Wen King Su. Myrinet: a gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[4] Dmitry Brodsky. R-simp to pr-simp: Parallelizing a model simplification algorithm. Technical Report TR-00-02, University of British Columbia, 2000.

[5] Dmitry Brodsky and Benjamin Watson. Model simplification through refinement. In Sidney Fels and Pierre Poulin, editors, *Graphics Interface '00*, page To appear in GI '00. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 2000.

[6] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovitch, and John Wilkes (HP Labs). An implementation of the Hamlyn sender-managed interface architecture. In *2nd USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 245–60, October 1996.

[7] Jeffery S. Chase, Andrew J. Gallatin, Alvin R. Labeck, and Kenneth G. Yokum. Trapeze messaging API. *Technical Report CS-1997-19, Duke University, Department of Computer Science*, November 1997.

[8] Cezary Dubnicki, Angelos Bilas, Kai Li, and Jim F. Philbin (Princeton). Design and implementation of virtual memory-mapped communication on Myrinet. In *International Parallel Processing Symposium*, April 1997.

[9] Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnick, Liviu Iftode, and Kai Li. Early experience with message-passing on the Shrimp multicomputer. In *Proc. of the 23rd International Symposium of Computer Architecture*, May 1996.

[10] Loic Prylli and Bernard Tourancheau. Bip: a new protocol designed for high performance networking on Myrinet. In *Workshop PC-NOW, IPPS/SPDP98*, 1998.

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

[12] Randolph Y. Wang, Arvind Krishnamurthy, Richard P Martin, Thomas E Anderson, and David E Culler. Modeling and optimizing communication pipelines. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems*, 1998.

[13] Matt Welsh, Anindya Basu, and Thorsten von Eiken. Incorporating memory management into user-level network interfaces. In *Hot Interconnects V*, Aug 1997.

[14] Kenneth G. Yokum, Jeffery S. Chase, Andrew J. Gallatin, and Alvin R. Labeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 243–52, August 1997.