TR-97-011

MPICH-PM: Design and Implementation of Zero Copy MPI for PM

Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori,
and Yutaka Ishikawa
Real World Computing Partnership
{tezuka, ocarroll, hori, ishikawa}@rwcp.or.jp

Received 18 March 1998

Tsukuba Research Center, Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono
Tsukuba-shi, Ibaraki 305, Japan

Abstract

This report describes the design and implementation of a high performance MPI library using a zero copy message transfer primitive supported by PM.

MPICH-PM consists of the MPICH implementation of the Message Passing Interface (MPI) standard, ported to the high-performance, communications library PM.

The zero copy message transfer primitive requires a memory area pinned down to physical memory, which is a restricted resource quantity under a paging memory system. Allocation of pinned down memory by multiple simultaneous requests for sending and receiving without any control can cause deadlock. To avoid this deadlock, we have introduced: i) separate of control of send/receive pin-down memory areas to ensure that at least one send and receive may be concurrently processed, and ii) delayed queues to handle the postponed message passing operations which could not be pinned-down.

Contents

1 Introduction **2**

2 Design of the Zero Copy MPI Implementation **4**
2.1 PM 4
2.2 MPICH 5
2.3 Eager Protocol using PM asynchronous message transfer 6
2.4 Rendezvous Protocol using PM zero copy message transfer 7
2.4.1 Performance Consideration 11

3 Implementation of the ADI for PM **14**
3.1 Eager Protocol 15
3.1.1 Short Blocking Send 15
3.1.2 Short Non-Blocking Send 16
3.1.3 Short Synchronous Send 17
3.1.4 Long Blocking Send 18
3.1.5 Long Non-Blocking Send 20
3.1.6 Long Synchronous Send 22
3.1.7 Unexpected Messages 24
3.2 Rendezvous Protocol 25
3.2.1 Short Blocking Send 25
3.2.2 Short Non-Blocking Send 25
3.2.3 Short Synchronous Send 25
3.2.4 Long Blocking Send 26
3.2.5 Long Non-Blocking Send 30
3.2.6 Effect of Delayed Queue 31
3.3 Details of the CH_SCORE implementation 35
3.3.1 Interfaces to PM message and RMA routines 35

4 Evaluation **41**
4.1 MPI primitive performance 41
4.2 NASPAR performance 43

5 Conclusion **46**

Chapter 1

Introduction

High performance network hardware, such as Myrinet and Gigabit ethernet, make it possible to build a high performance parallel machine using commodity computers. To realize a high performance communication library in such a system, a remote memory access mechanism or so called zero copy message transfer mechanism has been implemented, such as in PM[15], VMMC-2[6], AM[1], and BIP[2]. In the zero copy message transfer mechanism, user data is transferred to the remote user memory space with neither any data copy by a processor nor kernel trapping. The mechanism may be implemented in a network interface which has a DMA engine, such as Myrinet.

The zero copy message transfer mechanism requires that both sender and receiver memory areas must be pinned down to physical memory during transfer because the network interface can only access physical addresses. The pin-down operation is a very risky primitive in the sense that malicious users' pin-down requests may exhaust physical memory. Thus, the maximum pin-down area size is limited in a trustworthy operating system kernel.

The implementation of a higher level message passing library such as MPI using the zero copy message transfer mechanism is complicated if the maximum pin-down area size is smaller than the total size of messages being processed by the higher level message passing library at any one time. For example, assume that an application posts several asynchronous message send and receive operations whose total message size is more than the maximum pin-down area size at some time. In this case, the message passing library runtime must be responsible for controlling the pin-down area without deadlock.

The reader may think that the system's parameter for the maximum pin-down area size must be tuned so that a parallel application can run without exceeding the maximum pin-down area size. This approach, of course, cannot be accepted because the system tuner cannot predict the maximum pin-down area size for future applications and also under a multiuser environment.

In this paper, the MPI implementation using a zero copy message transfer mechanism, called *Zero copy MPI*, is designed and implemented based on the MPICH implementation. The PM communication driver is used as the low level communication layer, which supports not only a zero copy message transfer but also message passing mechanisms. An overview of our design to avoid deadlock due to starvation of the pin-down area is: i) separate control of send/receive pin-down memory areas to ensure that at least one send and receive may be processed concurrently, and ii) when the message area can not be

pinned down to the physical area, the request is postponed. The detailed protocol will be introduced in section 2.

Zero Copy MPI has been running on RWC PC Cluster II, consisting of 64 Pentium Pro 200 MHz processors with Myricom Myrinet. Performance is compared with low level benchmarks and the results of higher level NAS parallel benchmarks.

The organization of this paper is as follows: the design and implementation is presented after introducing MPICH and PM in section 2. The basic performance and the results of NAS parallel benchmarks are shown and discussed with other implementations in section 4. Finally, we conclude this paper in section 5.

Chapter 2

Design of the Zero Copy MPI Implementation

Our Zero Copy MPI is implemented using MPICH[7] on top of our lower level communication layer, PM. An overview of PM and MPICH is firstly introduced and then the design of our Zero Copy MPI is presented in this section.

2.1 PM

PM[15, 16] has been implemented on the Myrinet network whose network interface card has an on-board processor with a DMA engine and memory[11]. PM consists of a user-level library, a device driver for a Unix kernel, and a Myrinet communication handler which exists on the Myrinet network card. The Myrinet communication handler controls the Myrinet network hardware and realizes a network protocol. PM 1.2[14] realizes a user memory mapped network interface and supports a zero copy message transfer as well as message passing primitives as shown in Table 2.1. Messages are asynchronous and are delivered reliably and in posted send order with respect to any pair of processors.

The PM 1.2 API[14] for a zero copy message transfer provides pin-down and release

Table 2.1: PM Primitives

Remote Memory Access primitives	
`_pmMLock`	Pin-down some memory on the local host
`_pmMUnlock`	Unpin some memory on the local host
`_pmVWrite`	Start transfer from pinned memory on local host to destination host using DMA
`_pmWriteDone`	Test for completion of all pending writes (on sender side)
Message passing primitives	
`_pmGetSendBuf`	Allocate a send buffer on the network interface
`_pmSend`	Send a filled send buffer
`_pmSendDone`	Test for completion of pending sends
`_pmReceive`	Return address of a received-message buffer on the network interface
`_pmPutReceiveBuf`	Free the receive buffer

operations for the application specified memory area, namely _pmMlock and _pmMUnlock. The _pmVWrite primitive, whose parameters are the address and length of the data buffer on the sender and the receiver, transfers data without any copy operation by the host processor. The _pmWriteDone primitive reports the completion of the _pmVWrite operation. A _pmVWrite followed by a _pmSend to the same destination preserves order at the receiver. There is no remote memory read facility in PM.

The PM 1.2 API[14] for message passing has six primitives: i) message buffer allocation, ii) message buffer deallocation, iii) message send post, iv) testing send completion, and v) message receive post.

2.2 MPICH

Zero Copy MPI is designed and implemented based on MPICH. In the MPICH implementation, the program is divided into two layers, the machine independent part and the machine dependent part which is called the ADI (Abstract Device Interface)[7].

Each type of hardware needs its own implementation of the ADI, and the highest performance implementations of MPICH on each platform have highly tuned the internals of the ADI. However, MPICH also provides a highly simplified general purpose implementation of the ADI called the *channel device*. The channel device is almost all software emulation and the hardware dependent code has been distilled into the following three functions: `MPID_ControlMsgAvail` is a non-blocking check for any waiting control message. `MPID_RecvAnyControl` is a blocking receipt of any incoming control message . `MPID_SendControl` is a non-blocking send of a short control message. The order of sending control messages between two processors must be preserved at the receiver. We note here that PM's message passing primitives satisfy all the requirements for implementing these functions.

In addition to the functional interface, there are two protocols that may be enabled, the *eager* and *rendezvous* protocols. In the *eager* protocol, as soon as a message is posted, all of the message data is sent to the receiver. On the receiver side, if the receive has not already been posted, the data must be first buffered in memory. Later when the receive is posted the buffered data can be copied to the final user message buffer. Thus there is a single copy from the receive message buffers and through the processor if the message is expected, and a double memory copy if the message is unexpected.

In the *rendezvous* protocol, when a send is posted, only the message envelope is sent and buffered on the receiver. When the receive is posted, then the receiver informs the sender that it can send the message. Because the receiver is aware of the location of the user buffer, it can always copy the data direct to the user buffer, without any intermediate buffer.

In the discussion that follows we will refer to two queues internal to MPICH, the *unexpected message queue* and the *expected message queue* or *posted queue*. Because the message passing paradigm is one of data communication plus synchronization, the synchronization can happen in two ways: either the receiver or the sender arrive at the synchronization point first and wait for the other. In MPICH, we have a queue for each case.

When MPICH executes a nonblocking receive before a send has arrived, such as

Figure 2.1: Message Copy Transfer

MPI_Irecv, it places receive request object on the *posted queue*. When a message arrives, MPICH can check for a matching receive in the *posted queue*. If it is there, the message is said to be matched. Conversely, if a send arrives before the receive has been posted, the send is put in the *unexpected message queue*. When an MPI_Irecv is posted, it will check the *unexpected queue* first, and only if there is no match will it be placed on the *posted queue*.

Once a send and receive are matched, that does not mean that the data communication has completed. A short message with data that arrives all in one packet will be completed, but a large message sent in multiple *eager* packets will not yet be complete, and in the *rendezvous* protocol, the receiver must now ask the sender to go ahead before the message is complete. Even when data communication is complete, message is not completely delivered until MPI_Wait is called on the associated MPI_Request object.

2.3 Eager Protocol using PM asynchronous message transfer

To realize MPI using standard message passing facilities, the *eager* protocol is employed and the Message Copy Transfer technique is used.

The technique of "Message Copy Transfer", illustrated in 2.1 will be used. In Message Copy Transfer, the SRAM of the Myrinet is mapped into the address space of the application program. To send a message, the CPU copies the user's send buffer into the PM buffer on the network interface. The Myrinet network interface asynchronously transmits the date to the receiver's network interface, and is placed in the SRAM. The receiver's

CPU copies the data from the network interface to the application receive buffer.

In the *eager* protocol The sender only knows the address of the user send buffer and does not know the address of the receiver's buffer, or even if the receiver has posted a matching receive. This implies that if the matching recieve has been posted, the data will be copied by the cpu into the user receive buffer, but if the matching recieve has not been posted, the receiver will temporarily buffer the received data until the the receive has been posted. At that thime the location of the user recieve buffer will be known and it can be copied from the temporary buffer to the user buffer.

This simple eager protocol has the advantage of no negotiation before sending, hence lower latency, but if the receiver is not ready for the data, there will be higher latency due to the extra buffering and memory copy at the reciever. Additionally, buffering of really large messages may exhaust memory at the reiever.

A possible example execution flow of the *eager* protocol is:

1. Sender sends a *do_get* message to the receiver using message passing primitives when the MPI send operation is issued. The *do_get* message contains the first part of the data.

2. Sender sends many *cont_get* messages to the receiver using message passing primitives. Each message contains a successive part of the data.

3. The last *cont_get* message is marked as being the last part of the data. The MPI send operation is now complete and the user buffer can be returned to the user, and control returns to the user.

4. When the *do_get* message is received, a matching MPI recv operation is searched for on the *posted queue*. If found, the address of the user buffer is now known. If not, a temporary buffer is placed on the *unexpected message queue*.

5. The first part of the data is copied from the message to the user or temporary buffer.

6. When the *cont_get* messages are received, each successive part of the data is copied to the buffer.

7. When the final *cont_get* message is received, the final part of the data is copied and the receive is marked complete.

8. If the receive was blocking, control can return to the user.

2.4 Rendezvous Protocol using PM zero copy message transfer

To realize Zero Copy MPI, the *rendezvous* protocol is employed and the Zero Copy Transfer technique is used.

The technique of "Zero Copy Transfer", illustrated in 2.3 will be used. In Zero Copy Transfer, the SRAM of the Myrinet is not mapped into the address space of the application program. To send a message, the user buffer is pinned donw to physical memory. Next

Sender:

- When a MPI send operation is posted, the *send_request* message is sent to the receiver using the _pmSend primitive.
- When the *send_ok* message, containing the address of the buffer in the remote machine, is received from the receiver using the _pmReceive primitive.
  1. The message area sent to the receiver is pinned down to physical memory using _pmMlock.
  2. Data is transferred to the receiver using the _pmVWrite primitive.
  3. The sender polls until _pmWriteDone indicates the _pmVWrite has completed (on the sender side)
  4. The pin-down area is freed using the _pmMUnlock.
  5. The *send_done* message is sent to the receiver.

Receiver:

- When the *send_request* is received, check whether or not the corresponding MPI receive operation has been posted using the *posted queue*. If the operation has been posted:
  1. The posted request is dequeued from the *posted queue*.
  2. The receive buffer area is pinned down using the _pmMLock primitive.
  3. The *send_ok* is sent to the sender.

  else
  1. The *send_request* is enqueued on the *unexpected queue*.
- When the *send_done* message is received, the pin-downed area is freed using the _pmMUnlock.
- When a MPI receive operation is posted, check whether or not the corre-
sponding *send_ok* message has arrived using the *unexpected queue*.
If the message has arrived:
  1. The request is dequeued from the *unexpected queue*.
  2. The receive buffer area is pinned down using the _pmMLock primitive.
  3. The *send_ok* is sent to the sender.

  else
  1. The posted request is enqueued into the *posted queue*.

Figure 2.2: Simple Protocol

Figure 2.3: Zero Copy Transfer

the CPU asks the Myrinet network interface to DMA copy the data from the user buffer to the Myrinet SRAM. The The Myrinet network interface transmits the data to the receiver's network interface and SRAM. The Myrinet DMA engine on the receiver uses DMA transfer to place the data directly into the user buffer, provided it has already been pinned down.

That is, the sender must know the address of both user buffers before data can be sent. This implies that both the send and receive must have first been posted, and that the receiver can inform the sender the address of its buffer. This negotiation protocol is implemented using the PM message passing primitives. First of all, a simple protocol, with no consideration of exceeding the maximum pin-down area size, is introduced in Figure 2.2

As illustrated in Figure 2.4, an possible example execution flow of the simple protocol is:

1. Sender sends a *send_request* to send to the receiver using message passing primitives when the MPI send operation is issued.

2. When the MPI receive operation is posted at the receiver and the operation matches the *send_request* sent by the sender, the buffer is pinned down and the *send_ok* message, containing the address of the buffer, is sent back to the sender using message passing

3. Sender receives the destination buffer address, and pins down its own memory.

4. Sender calls `_pmVWrite` as many times as necessary to transfer the entire buffer (the amount written per call is limited by a maximumum transfer unit)

5. Sender waits for completion of the writes by calling `_pmWriteDone`

Figure 2.4: An Example Protocol

Figure 2.5: An Example Deadlock in the Naive Protocol

6. Sender sends a *send_done* message to the receiver

7. Sender unpins the send buffer

8. Receiver unpins the receive buffer.

This naive protocol inherently causes deadlock. Referring to figure2.5, for example, suppose two nodes are concurrently sending a message larger than the maximal pin-down memory size to each other. Each node acts as both a sender and receiver. Each node first issues MPI_ISEND operations followed by MPI_IRECV operations. It is possible that they each pin-down their entire area for receiving, and inform the other of their readiness to receive. However, when they try to send they find they have no pin-down area left to use for sending. They can't unpin some of the receive buffer area and repin it on the send buffer because they must assume that the remote processor is writing to their receive buffer.

To avoid deadlock, the following implementation are introduced:

1. Separation of maximum pin-down memory size for send and receive
   Our ADI, not PM, manages two types of maximum pin-down memory size, one for sending and one for receiving. Doing so ensures that we can always be processing at least one send and receive concurrently, preventing deadlock. Calls to _pmMLock and _pmMUnlock are only responsible for managing the pin-down area and do not distinguish between whether the user will use it for sending or receiving.

2. Delayed Queue
   When requests to pin down memory for receiving exceed the maximum allowed for receiving by our ADI, those requests must be delayed by the ADI in a special queue, and executed in order as soon as resources become available. This queue is different from MPICH's internal posted message queues. The delayed queue must be frequently polled to ensure progress of communication. Writes are executed immediately the sender is informed that the receiver is ready, and no delayed write queue is necessary as no more than a certain maximum number of bytes (actually, memory pages) are allowed to be pinned-down on send buffers. This guarantees that it is always possible to pin down at least some memory on a send buffer. (If it's not big enough for the whole send buffer, we can send in parts by unpinning and repinning several times, resynchronizing with the receiver if necessary.) A design with a delayed write queue as well as delayed receive queue, would also be possible.

The revised algorithm is defined in Figure 2.6

2.4.1 Performance Consideration

It can be seen that *rendezvous* protocol using our zero copy implementation in the previous subsection requires three control messages to be exchanged, compared to just one for the *eager* protocol which uses one message. Thus the latency for small messages will be three times that of an *eager* short message. However, long message bandwidth improves greatly due to the much higher bandwidth of the remote memory write primitive.

In the current implementation on the Pentium Pro 200 MHz cluster, a message of less than 2 Kbytes uses the *eager* protocol while a message greater than 2 Kbytes uses the *rendezvous* protocol determined by our MPI experimentation. This is mentioned with the performance result in the next section.

Sender:

- When a MPI send operation is posted, the *send_request* message is sent to the receiver using the _pmSend primitive.
- When the *send_ok* message, containing the address of the buffer in the remote machine, is received from the receiver using the _pmReceive primitive.

  1. The message area sent to the receiver is pinned down to the physical memory using _pmMlock.
  2. Data is transferred to the receiver using the _pmVWrite primitive.
  3. The sender polls until _pmWriteDone indicates the _pmVWrite has completed (on the sender side)
  4. The pin-down area is freed using _pmMUnlock.
  5. The *send_done* message is sent to the receiver.

  Note: Since the maximum pin-down area size for the *receive* operations is less than the total amount that may be pinned down by PM, this guarantees that we can lock at worst some of the message area to be sent in step 1 above.

Receiver:

- When the *send_request* is received, check whether or not the corresponding MPI receive operation has been posted using the *posted queue*.
  If the operation has been posted:

  1. the posted request is dequeued from *posted queue*.
  2. perform the **PDOP** procedure described later.

  else

  1. The request is enqueued to the *unexpected queue*.
- When the *send_done* message is received, the pin-downed area is freed using the _pmMUnlock.
- When a MPI receive operation is posted, check whether or not the coresponding *send_ok* message has arrived using the *unexpected queue*.
  If the message has already arrived:

  1. The request is dequeued from the *unexpected queue*.
  2. perform for the **PDOP** proceedure described later.

  else

  1. The request is enqueued into the *posted queue*.
- Whenever a check for incoming messages is made in the ADI, dequeue a request from the *delayed queue* and perform for the **PDOP** proceedure.

Figure 2.6: Revised Protocol

Receiver:

PDOP If the maximum pin-down are size for the receiving operation is not exceeded,

1. The receive buffer area is pinned down using the _pmMLock primitive.
2. The *send_ok* is sent to the sender.

else

1. The request is enqueued into the *delayed queue*.

Figure 2.7: PDOP procedure for Revised Protocol

Chapter 3

Implementation of the ADI for PM

As explained in the previous chapter, the ADI implements two separate protocols called

- Eager (based on message passing)
- Rendezvous (based on Remote Memory Access)

These two protocols must implement the standard MPI types of communication, such as

- Blocking Send
- Non-blocking Send
- Synchronous Send

In addition, for each of the above communication types, each protocol has separate cases to consider

- Short Messages
- Long Messages

This chapter, based on the algorithms outlined in the previous chapter, will first illustrate the internal protocols for each message type, and then will discuss the actual implementation source code.

Figure 3.1: Typical Eager Short Blocking Message Execution Flow

3.1 Eager Protocol

3.1.1 Short Blocking Send

- Sender
  1. A short message packet, which is exactly large enough to hold the short message header and the short message data, is allocated in the PM send buffer.
  2. The message type is marked as short and the header information is filled in
  3. The data is copied by the CPU to the packet in the buffer using _memcpy
  4. the *short* message is sent to the reciever using _pmSend
  5. The MPI_SEND is marked as complete
  6. Control returns to the user program
- Receiver
  1. When a message is received by _pmReceive
  2. The message type is decoded and the receive buffer is located
  3. The data is copied by the CPU to the user receive buffer
  4. The MPI_RECV is marked as complete
  5. Control returns to the user program

Figure 3.2: Typical Eager Short Non-Blocking Message Execution Flow

3.1.2 Short Non-Blocking Send

- Sender
  1. A MPI_REQUEST structure is filled in for this message and marked not complete
  2. A short message packet, which is exactly large enough to hold the short message header and the short message data, is allocated in the PM send buffer.
  3. The message type is marked as short and the header information is filled in
  4. The data is copied by the CPU to the packet in the buffer using _memcpy
  5. the *short* message is sent to the reciever using _pmSend
  6. The Request argument of MPI_ISEND is marked as complete
  7. Control returns to the user program
  8. The user program calles MPI_WAIT on the request
  9. Since the request was marked as complete, control returns immediately to the user program
- Receiver
  1. When a message is received by _pmReceive
  2. The message type is decoded and the receive buffer is located
  3. The data is copied by the CPU to the user receive buffer
  4. The MPI_RECV is marked as complete
  5. Control returns to the user program

Figure 3.3: Typical Eager Short Synchronous Message Execution Flow

3.1.3 Short Synchronous Send

- Sender
  1. A *short_sync* message packet, which is exactly large enough to hold the short synchronous message header and the short message data, is allocated in the PM send buffer.
  2. The message type is marked as short_sync and the header information is filled in
  3. The data is copied by the CPU to the packet in the buffer using _memcpy
  4. The *short* message is sent to the reciever using _pmSend
  5. Messages are polled until at *sync_ack* acknowledgement is received with _pmReceive
  6. The MPI_SSEND is marked as complete
  7. Control returns to the user program
- Receiver
  1. When a message is received by _pmReceive
  2. The message type is decoded and the receive buffer is located
  3. The data is copied by the CPU to the user receive buffer
  4. A *sync_ack* packet is allocated in the PM send buffer.
  5. The message type is marked as short_sync and the header information is filled in
  6. The *short_sync* message is sent to the reciever using _pmSend
  7. The MPI_RECV is marked as complete
  8. Control returns to the user program

Figure 3.4: Typical Eager Long Blocking Message Execution Flow

3.1.4 Long Blocking Send

- Sender
  1. A *do_get* message packet, which is exactly large enough to hold the long message header and the first part of the message data, is allocated in the PM send buffer.
  2. The message type is marked as *do_get* and the header information is filled in. The header includes the total length of the message and the length and offset of this part.
  3. The first part of the data is copied by the CPU to the packet in the buffer using _memcpy
  4. the *do_get* message is sent to the reciever using _pmSend
  5. A *cont_get* ("continue get") message packet, which is exactly large enough to hold the long message header and the first part of the message data, is allocated in the PM send buffer.
  6. The message type is marked as *cont_get* and the header information is filled in. The header includes the total length of the message and the length and offset of this part.
  7. The second part of the data is copied by the CPU to the packet in the buffer using _memcpy
  8. the *cont_get* message is sent to the reciever using _pmSend
  9. As many *cont_get* messages are sent in the above way until all the data has beens sent.
  10. The MPI_SEND is marked as complete
  11. Control returns to the user program
- Receiver
  1. When *do_get* message is received by _pmReceive

2. The message type is decoded and the receive buffer is located
3. The first part of the data is copied by the CPU to the user receive buffer
4. The network is polled until a *cont_get* arrives and is received by _pmReceive
5. The second part of the data is coppied to the user receive buffer
6. If the offset and message lengths of the *cont_get* indicate that there are more packets, further *cont_get* messages are waited for as above.
7. If the offset and message lengths of the *cont_get* indicate that this is the last *cont_get* packet, the MPI_RECV is marked as complete
8. Control returns to the user program

Figure 3.5: Typical Eager Long Non-Blocking Message Execution Flow

3.1.5 Long Non-Blocking Send

- Sender

1. A *do_get* message packet, which is exactly large enough to hold the long message header and the first part of the message data, is allocated in the PM send buffer.
2. The message type is marked as *do_get* and the header information is filled in. T he header includes the total length of the message and the length and offset of this part.
3. The first part of the data is copied by the CPU to the packet in the buffer using _me mcpy
4. the *cont_get* message is sent to the reciever using _pmSend
5. A *cont_get* ("continue get") message packet, which is exactly large enough to ho ld the long message header and the first part of the message data, is allocated in the PM se nd buffer.
6. The message type is marked as *cont_get* and the header information is filled in. The header includes the total length of the message and the length and offset of this part.
7. The second part of the data is copied by the CPU to the packet in the buffer using _m emcpy
8. the *cont_get* message is sent to the reciever using _pmSend
9. As many *cont_get* messages are sent in the above way until all the data has beens sent.
10. The MPI_SEND is marked as complete
11. Control returns to the user program

- Receiver

  1. When *do_get* message is received by _pmReceive
  2. The message type is decoded and the receive buffer is located
  3. The first part of the data is copied by the CPU to the user receive buffer
  4. The network is polled until a *cont_get* arrives and is received by _pmReceive
  5. The second part of the data is coppied to the user receive buffer
  6. If the offset and message lengths of the *cont_get* indicate that there are more packets, further *cont_get* messages are waited for as above.
  7. If the offset and message lengths of the *cont_get* indicate that this is the last *cont_get* packet, the MPI_RECV is marked as complete
  8. Control returns to the user program

Figure 3.6: Typical Eager Long Synchronous Message Execution Flow

3.1.6 Long Synchronous Send

- Sender

1. A *do_get_sync* message packet, which is exactly large enough to hold the long message header and the first part of the message data, is allocated in the PM send buffer.
2. The message type is marked as *do_get_sync* and the header information is filled in. T he header includes the total length of the message and the length and offset of this part.
3. The first part of the data is copied by the CPU to the packet in the buffer using _me mcpy
4. the *do_get_sync* message is sent to the reciever using _pmSend
5. A *cont_get* ("continue get") message packet, which is exactly large enough to ho ld the long message header and the first part of the message data, is allocated in the PM se nd buffer.
6. The message type is marked as *cont_get* and the header information is filled in. The header includes the total length of the message and the length and offset of this part.
7. The second part of the data is copied by the CPU to the packet in the buffer using _m emcpy
8. the *cont_get* message is sent to the reciever using _pmSend
9. As many *cont_get* messages are sent in the above way until all the data has beens sent.
10. The sender waits for the *sync_ack* acknowledgement message from the receiver with _pmReceive.

11. The MPI_SEND is marked as complete
12. Control returns to the user program

- Receiver

1. When *do_get_sync* message is received by _pmReceive
2. The message type is decoded and the receive buffer is located
3. The first part of the data is copied by the CPU to the user receive buffer
4. A *sync_ack* message packet is created to acknowledge the start of receipt
5. The *sync_ack* is sent to the sender with _pmSend
6. The network is polled until a *cont_get* arrives and is received by _pmReceive
7. The second part of the data is coppied to the user receive buffer
8. If the offset and message lengths of the *cont_get* indicate that there are more packets, further *cont_get* messages are waited for as above.
9. If the offset and message lengths of the *cont_get* indicate that this is the last *cont_get* packet, the MPI_RECV is marked as complete
10. Control returns to the user program

Figure 3.7: Typical Eager Unexpected Message

3.1.7 Unexpected Messages

Figure 3.7 shows the effect of a message arriving before the receiver is ready for it, in the Eager protocol. In this case example, messages are probed for with MPI_PROBE and only after a message has arrived is the receive posted. The same situation can occur in other cases, for example, when the sender sends messages in the order `MPI_SEND(A); MPI_SEND(B);` and the receiver receives them in the opposite order, `MPI_RECV(B); MPI_RECV(A);`. In that case, while MPI is waiting for message B on the sender, it will buffer message A on the unexpected list.

- Sender
  1. A short message packet, which is exactly large enough to hold the short message header and the short message data, is allocated in the PM send buffer.
  2. The message type is marked as short and the header information is filled in
  3. The data is copied by the CPU to the packet in the buffer using _memcpy
  4. the *short* message is sent to the reciever using _pmSend
  5. The MPI_SEND is marked as complete
  6. Control returns to the user program
- Receiver
  1. Any MPI call which checks for messages is called, such as MPI_PROBE
  2. When a message is received by _pmReceive
  3. The message type is decoded and the receive buffer cannot located because the receive has not been posted yet
  4. A temporary buffer is allocated on the Unexpected Message Queue

5. The data is copied by the CPU to the temporary buffer on the Unexpected Queue
6. The MPI_PROBE completes and reports back to the user program that a message is waiting
7. Control returns to the user program
8. The user program calls MPI_RECV to receive the probed message
9. The temporary buffer is located on the unexpected message queue
10. The temporary buffer is copied to the user receive buffer
11. The temporary buffer is disposed of and dequeued from the Unexpected Message Queue.
12. The MPI_RECV is marked as complete
13. Control returns to the user program

3.2 Rendezvous Protocol

3.2.1 Short Blocking Send

Even when the Rendezvous protocol is used, short messages are carried in exactly the same way as in the Eager protocol. See figure 3.1.

3.2.2 Short Non-Blocking Send

Even when the Rendezvous protocol is used, short non-blocking messages are carried in exactly the same way as in the Eager protocol. See figure 3.2.

3.2.3 Short Synchronous Send

Short Synchronous messages are carried in exactly the same way as Rendezvous Long Synchronous messages. See figure 3.8. Since the Rendezvous protocol requires a pin-down synchronization between sender and receiver, no explicit synchronization acknowledgement is required.

This causes the latency of Short Synchronoous Messages to be much longer than Short Blocking Messages.

Figure 3.8: Typical Rendezvous Long Blocking Message Execution Flow

3.2.4 Long Blocking Send

Because the amount of memory that may be pinned down at once is limited by PM (currenly to about 4MB), and since there may be other simultaneous recieves also consuming some of the pin-down limit, there are two cases to consider:

1. The receiver can pin down the entire recieve buffer at once
2. The receiver can pin down only part of the receive buffer

Full Receive Buffer Pin-Down In the case that the Receiver can pin down the entire receive buffer, only one pin down synchronoization is necessary. This is illustrated in figure 3.8.

1. Sender sends a *send_request* to send to the receiver using message p assing primitives when the MPI send operation is issued.
2. When the MPI receive operation is posted at the receiver and the operation matches the *send_request* sent by the sender, the buffer is pinned down and the *send_ok* message, containing the address of the buffer, is sent back to the sender using message p assing
3. Sender receives the destination buffer address, and pins down its own memo ry.
4. Sender calls `_pmVWrite` as many times as necessary to transfer the e ntire buffer (the amount written per call is limited by a maximumum transfer uni t)
5. Sender waits for completion of the writes by calling `_pmWriteDone`
6. Sender sends a *send_done* message to the receiver
7. Sender unpins the send buffer
8. Receiver unpins the receive buffer.

9. Control returns to the user program

Figure 3.9: Typical Rendezvous Very Long Blocking Message Execution Flow

Partial Receive Buffer Pin-Down In the case that the Receiver can only pin down part of the receive buffer, the protocol is extended to allow for several resynchronizations, one for each part of the part of the receive buffer that is pinned down. See figure 3.9.

1. Sender sends a *send_request* to send to the receiver using message p assing primitives when the MPI send operation is issued.

2. When the MPI receive operation is posted at the receiver and the operation matches the *send_request* sent by the sender, the first part of the buffer is pinned down and the *send_ok* message, containing the address of the buffer, is sent back to the sender using message p assing

3. Sender receives the destination first part of the buffer address and length, and pins down the corresponding part of the send buffer.

4. Sender calls `_pmVWrite` as many times as necessary to transfer the e ntire buffer (the amount written per call is limited by a maximumum transfer uni t)

5. Sender waits for completion of the writes by calling `_pmWriteDone`

6. Sender sends a *done_send_cont* message to the receiver

7. Receiver unpins the first part of the receive buffer
8. Receiver sends *send_ok_cont* message, containing the address and length of the next part of the buffer
9. Sender pins the second part of the receive buffer, and transfers each part of the buffer in the same way.
10. For the final part of the message, sender sends a *send_done* message to the receiver
11. Sender unpins the last part of the send buffer.
12. Receiver unpins the receive buffer.
13. control returns to the user program in each case

Figure 3.10: Typical Rendezvous Long Non-Blocking Message Execution Flow

3.2.5 Long Non-Blocking Send

Non-blocking eager protocol (figure 3.5) and non-blocking rendezvous protocol (figure ??) differ fundamentally in timing. In the eager protocol, all data is delivered when the MPI_ISEND is posted. The subesequent MPI_WAIT is essentially a no-op. In contrast, the rendezvous protocol, only the first *send_request* is delivered and then user computation is performed. The data transfer does not actually take place until the MPI_WAIT operation.

1. Sender sends a *send_request* to send to the receiver using message p assing primitives when the MPI_ISEND operation is issued.
2. Control returns to the sender's user program
3. When the MPI receive operation is posted at the receiver and the operation matches the *send_request* sent by the sender, the buffer is pinned down and the *send_ok* message, containing the address of the buffer, is sent back to the sender using message p assing
4. Eventually, the sender calls MPI_Wait
5. Sender receives the destination buffer address, and pins down its own memo ry.
6. Sender calls `_pmVWrite` as many times as necessary to transfer the e ntire buffer (the amount written per call is limited by a maximumum transfer uni t)
7. Sender waits for completion of the writes by calling `_pmWriteDone`
8. Sender sends a *send_done* message to the receiver
9. Sender unpins the send buffer
10. Receiver unpins the receive buffer.
11. Control returns to the user program

Figure 3.11: Typical Rendezvous Message Execution Flow Without Delayed Queue

3.2.6 Effect of Delayed Queue

The effect of the delayed queue for delayaing receive pin-downs is only seen when multiple senders are sending to the same reciever. In these examples, two senders are sending to a single receiver.

Concurrent Receives without Invoking Delayed Queue In figure 3.11

1. The receiver has posted two non blocking receives with MPI_IRECV.
2. The receiver calls MPI_WAITALL to wait for both to complete
3. When the receiver gets the first *send_request* from sender 1, it locks the first recieve buffer and responds in the usual way.
4. While the receiver is waiting for the *done_send* from sender 1, it continues to process messages
5. The receiver encounters a second *send_request* from sender 2.
6. The receiver checks it's resources for pinning down receive buffers, and finds that it can pin down the second buffer without running out of resources. It locks the second buffer and responds as usual.
7. Receiver waits until a *done_send* message is received from both senders

8. Receiver returns to user program.

Note the following points

- Pin-down resources were not exhausted in this example, hence the delayed queue was not invoked
- We note that data transfer, effected by _pmVWrite, can happen in parallel (although total bandwidth is shared between senders)
- We note that the smaller message finishes first in this case, even though processing started later.

Figure 3.12: Typical Rendezvous Message Execution Flow With Delayed Queue

Concurrent Receives with Delayed Queue In figure 3.12

1. The receiver has posted two non blocking receives with MPI_IRECV.
2. The receiver calls MPI_WAITALL to wait for both to complete
3. When the receiver gets the first *send_request* from sender 1, it locks the first recieve buffer and responds in the usual way.
4. While the receiver is waiting for the *done_send* from sender 1, it continues to process messages
5. The receiver encounters a second *send_request* from sender 2.
6. The receiver checks it's resources for pinning down receive buffers, Resources are exhausted.
7. The receiver places the second request on the delayed queue

8. Receiver waits until a *done_send* message is received from sender 1.
9. Before receiver returns to user program, it finds a delayed request and activates it, removing it from the delayed queue.
10. When the second transfer is over, control returns to application

Note the following points

- Pin-down resources were exhausted in this example, hence the delayed queue was invoked
- There was no parallel receiving of data
- The reception of messages was serialized.

3.3 Details of the CH_SCORE implementation

MPICH-PM is implemented as an MPICH "channel device". The implementation of the device is in the directory mpid/ch_score and is called the CH_SCORE implementation of the Abstract Device Interface.

Figure 3.13 shows the overall interrellation ships of the source code files. As shown in the figure, all MPI calls first pass through the upper layers of the MPI Library which contain the routines for high level abstractions such as communicators and datatypes and collective communication. Withing the MPI library are also routines for posted and unexpected queue management. Communcation calls are translated by the upper layers into the approprivate calls to the ADI. When sending a message, the upper layer translates all send calls into a call to a routine in the file scoresend.c, and all receive calls go to scorerecv.c. scoresend.c and scorerecv.c implement actions which are common to sending and receiving regardless of protocol.

Depending on the length of the message and whether Message Copy or Zero Copy has been selected, scoresend.c and scorerecv.c dispatch their calls to the appropriate module. scoreget.c implements the Message Copy (Eager) protocol and scorerndv.c implements the Zero Copy (Rendezvous) protocol. scoreget and scorerndv implement protocols specific actions, that is the detailed message passing decisions for each protocol.

Scorepriv.c implements the private routines of the ADI that are most specific to PM. Part of the module contains message passing routines which call the message passing functions in PM, and the other part of the module handles Zero Copy routines, which call the RMA routines in PM, and implement the Delayed Queue which is necessary to manage the pin-down resources and prevent deadlock when using RMA.

3.3.1 Interfaces to PM message and RMA routines

The file `scorepriv.c` contains the only routines that call PM directly.

3.3.1.1 Basic Abstract Device Interface

```
int MPID_ControlMsgAvail(void)
void MPID_RecvAnyControl(MPID_PKT_T ** pkt, int maxsize, int *from)
void MPID_RecvAnyControl_rma(MPID_PKT_T ** pkt, int maxsize, int *from)
void MPID_RecvAnyControl_get(MPID_PKT_T ** pkt, int maxsize, int *from)
void MPID_SendControl(MPID_PKT_T * pkt, int size, int dest)
void MPID_SendControlBlock(MPID_PKT_T * pkt, int size, int dest)
MPID_PKT_T     *MPID_SCORE_GetSendPkt(int len)
```

These routines provide the basic message passing interface to to PM

- MPID_ControlMsgAvail
  This is a non blocking check for messages. It checks the network interface by calling _score_peek_network

- MPID_RecvAnyControl
  This is a blocking receive for the first message available. There are two versions, one

Figure 3.13: Source file structure of the CH_SCORE implementation of the ADI in MPICH-PM

for each protocol case. In the eager (message) protocol, it calls MPID_RecvAnyControl_get in the rendezvous (zero copy) protocol, it calls MPID_RecvAnyControl_rma.

1. MPID_RecvAnyControl_get
   Eager protocol only. Blocking check for messages by calling _score_recv_message. Also reports idle status to SCORE by calling _scored_become_idle or _scored_become_busy.
2. MPID_RecvAnyControl_rma
   Rendezvous protocol only. In the rendezvous protocol, it is sometimes necessary to buffer incoming packets. If packets are buffered on the packet queue, we remove from that queue, else check the network as for MPID_RecvAnyControl_get.

- MPID_SCORE_GetSendPkt
  Allocates a send packet in the PM buffer.
- MPID_SendControl
  Sends the allocated packet.
- MPID_SendControlBlock Same as MPID_SendControl.

3.3.1.2 Initialization

```
int MPID_SCORE_init(int *argc, char ***argv)
int MPID_SCORE_finalize(void)
void MPID_SCORE_usage_warn(char *option)
```

These routines initialize the ADI and interface to the operating system.

- MPID_SCORE_init
  Initialize data structures. This analyzes the `-score` command line options to select `mpizerocopy` or not.
- MPID_SCORE_finalize
  Finalize processing.
- MPID_SCORE_usage_warn
  Report incorrect option usage.

3.3.1.3 Remote Memory Access Interface

```
void MPID_CreateSendTransfer(void *buf, int size,
void MPID_CreateRecvTransfer(void *buf, int size,
void MPID_StartRecvTransfer(void *buf, int size,
int MPID_StartSendTransfer(void *buf, int size,
void MPID_EndRecvTransfer(void *buf, int size,
void MPID_EndSendTransfer(void *buf, int size,
int MPID_TestRecvTransfer(int rid)
int MPID_TestSendTransfer(int sid)
```

These functions interface to PM's remorte memory access in a high level way. Because the design concetrates on managing receives, some of these operations are no-ops for sending.

- MPID_CreateSendTransfer
  Not used.
- MPID_CreateRecvTransfer
  Creates a pin-down region for the receive operation and returns it.
- MPID_StartRecvTransfer
  Marks the receive operations data structure as started.
- MPID_StartSendTransfer
  Marks the send operations data structure as started.
- MPID_EndRecvTransfer
  Marks the receive operations finished and unlocks the region.
- MPID_EndSendTransfer
  Marks the send operation finished and unlocks the region.
- MPID_TestRecvTransfer
  Calls MPID_SCORE_Process_RMA_Queues to start any other waiting operations.
- MPID_TestSendTransfer
  Not used.

3.3.1.4 Pin Down Memory Management

```
int MPID_SCORE_Can_Lock(int lock_type)
void MPID_SCORE_Lock(MPID_SCORE_PIN_T * pin)
void MPID_SCORE_Unlock(MPID_SCORE_PIN_T * pin)
void MPID_SCORE_VWrite(int dst_node, u_int dst_addr,
void MPID_SCORE_TryLock(void *buf, int size, MPID_SCORE_PIN_T * pin)
int MPID_SCORE_TryWrite(int dest, char *buf, int size, MPID_SCORE_PIN_T * recv_pin)
```

These routines interface directly to PM's RMA functions. They try to pin down the correct amount of memory, but if it is not available, they reduce their request. Creates the concepts of a "receive pin" or a "send pin"

- MPID_SCORE_Can_Lock
  Check that there are sufficient locks available (for receiving or sending). This is the key routine that implements the delayed queue test.
- MPID_SCORE_Lock
  Lock an exact sized region by calling _pmMLock, or return error.
- MPID_SCORE_Unlock
  Unlock a send or receive region.

- MPID_SCORE_VWrite
  Write from one locked region to another. Call _pmVWrite as many times as necessary to write the entire region.

- MPID_SCORE_TryLock
  Try to lock a whole region, or as much as possible. If not enough, modify the request by reducing it.

- MPID_SCORE_TryWrite
  Tries to lock a region on the local PE and write it to the receiver's already locked region. Won't write more bytes than the receiver has locked. Returns number of bytes actually written. Sender's region will be be unlocked at the end of the call.

3.3.1.5 Message Packet Queuing

```
void MPID_SCORE_QueueConstruct(MPID_PKT_QUEUE * q)
void MPID_SCORE_Enqueue(MPID_PKT_QUEUE * q, MPID_PKT_T * in_pkt, int len)
MPID_PKT_T     * MPID_SCORE_Dequeue(MPID_PKT_QUEUE * q)
void MPID_SCORE_EnqueueControl(MPID_PKT_QUEUE * q, MPID_BLOCKING_TYPE is_blocking)
```

Only used when Rendezvous protocol is used. In the Rendezvous protocol, incoming packets need to be queued during congestion.

- MPID_SCORE_QueueConstruct
  Initialize the packet queue.

- MPID_SCORE_Enqueue
  Enqueue a packet given as an argument.

- MPID_SCORE_Dequeue
  Dequeue a packet.

- MPID_SCORE_EnqueueControl
  Get a packet from the network interface and enqueue it. Can be blocking or non blocking.

3.3.1.6 Delayed Queue

```
int MPID_SCORE_Do_Request_Enqueue(MPID_RNDV_T recv_handle,
int MPID_SCORE_Ack_Request_Enqueue(MPIR_RHANDLE * dmpi_recv_handle,
int MPID_SCORE_Do_Request_Dequeue(void)
int MPID_SCORE_Ack_Request_Dequeue(void)
int MPID_SCORE_Process_RMA_Queues(void)
```

These routines are only used by the Rendezvous (Zero Copy) protocol.

- MPID_SCORE_Do_Request_Enqueue
  Defer a request on the sender side.

- MPID_SCORE_Ack_Request_Enqueue
  Defer a request on the receiver side.
- MPID_SCORE_Do_Request_Dequeue
  Remove and execute a defered request on the sender side. used)
- MPID_SCORE_Ack_Request_Dequeue
  Remove and execute a defered request on the receiver side.
- MPID_SCORE_Process_RMA_Queues
  If the RMA queues are not empty, dequeue one operation and execute it.

Chapter 4

Evaluation

Table 4.1 shows the specifications of the machines considered. The ZeroCopyMPI/PM is our Zero Copy MPI implementation in this table.

4.1 MPI primitive performance

We have two (integrated) MPI implementations, one uses only the PM message passing feature and another is Zero Copy MPI using PM. Figure 4.1 shows MPI bandwidth of two implementations. This result is obtained by the MPICH performance test command, `mpirun -np 2 mpptest -pair -blocking -givdy` for sizes from 4 bytes to 1 megabyte.

Examining the graph, the point-to-point performance slope of MPI using only the PM message passing feature drops off for messages larger than 2 Kbytes, although it is better than the zero copy strategy for smaller messages. Hence our Zero Copy MPI implementation uses PM the message passing feature for messages of less than 2 Kbytes in length. That is why the same performance is achieved on the graph for messages smaller than 2 Kbytes. In our implementation the message size at which the protocol switch is made can be set at run-time by a command line argument or environment variable. For example, the trade-off point on a 166 MHz Pentium Myrinet cluster is different to that of a 200 MHz Pentium Pro Myrinet cluster. This allows the same compiled programs to run optimally on different binary-compatible clusters.

The low point on the message passing-only graph at about 8 Kbytes is due to the message exceeding the maximum transfer unit of PM, so more than one packet is required to send the message.

Table 4.1: Machines

System	ZeroCopyMPI/PM	MPI/FM	MPI/GAM	MPI/BIP	MPI/SR2201
Processor	Pentium Pro	Pentium Pro	UltraSPARC	Pentium Pro	HP-RISC
Clock (MHz)	200	200	167	*not reported*	150
Bandwidth(MBytes/sec)	160	160	160	160	300
Network Topolgy	-	-	-	-	3D crossbar
Network Interface	Myrinet	Myrinet	Myrinet	Myrinet	*dedicated*

Figure 4.1: MPI bandwidth

Table 4.2: MPI Point-to-point Performance

System	ZeroCopyMPI/PM	MPI/FM	MPI/GAM	MPI/BIP	MPI/SR2201
Min Latency (usec)	13.16	17.19	15.49	12	25.95
Max Bandwidth (MB/s)	98.81	69.34	38.62	113.68	214.5

Other MPI implementations are compared in Table 4.2. Considering the implementation on commodity hardware and network, our Zero Copy MPI is better than MPI on FM[10] and MPI on GAM[3] but worse than MPI on BIP[12]. Moreover, MPI latency is better than a commercial MPP system Hitachi SR2201, which also uses a form of remote memory access, called Remote DMA, in Hitachi's implementation of MPI.

Our Zero Copy MPI runs in the multi-user environment. In other words, many MPI applications may run simultaneously on the same nodes. This nature is inherited from a unique feature of PM[15]. As far as we know, MPI/FM and MPI/BIP may only used by a single user at a time. In that sense, our Zero Copy MPI achieves good performance while supporting a multi-user environment.

4.2 NASPAR performance

Figure 4.2 graphs the results, in Mops/second/process, of running the eight NASPAR 2.3b2 bencharmarks class A, from 16 to 64 processors, under Zero Copy MPI/PM, and compares them to the reported results of the Hitachi SR2201 and the Berkeley NOW (using MPI/GAM or similar AM based MPI) network of workstations from the NASPAR web site [4]. Some combinations of benchmark and size lack published results for the other two systems. We have no native FORTRAN compiler, so we used the `f2c` fortran to C translator, and compiled the programs with `gcc -O4`. Unfortunately `f2c` is widely regarded to be inferior in performance of generated code to a native compiler such as the `f77` available on Unix workstations.

The graphs reveal that LU, MG, and BT scale well with fairly flat curves to 64 processors on Zero Copy MPI/PM and our hardware and software environment. SP stands out as increasing performance substantially as processor numbers increase, especially as the NOW performance decreases over the same range. CG and IS decline in relative performance per processor.

There are many MPI implementations, e.g., MPI/FM[10], MPI/GAM[3], MPI/BIP[12], and so on, and many Zero-Copy message transfer low level communication libraries, e.g., AM[1], BIP[2] and VMMC2[6]. As far as we know, however, only MPI/BIP implements zero-copy message transfer on a commodity hardware and network combination.

BIP forces upon the users that a virtual memory area be pinned down to the contiguous physical area before communication. Because the implementation of MPI on BIP has not been published as far as we know, we cannot compare the design. According to the data published on the web, only IS and LU benchmarks of NAS parallel benchmarks have been reported. Table 4.3 shows the comparison of MPI/BIP and ours, indicating as Zero Copy MPI/PM. As shown in Table 4.2, MPI/BIP point-to-point performance is better than ours. However, our MPI performance for NAS parallel benchmarks are better than on MPI/BIP.

In all the above graphs, the X-axis is the number of processors while the Y-axis is Mops/s/processor.

Figure 4.2: NAS Parallel Benchmarks (Class A) Results

Table 4.3: NAS parallel benchmarks with 4 Pentium Pro on MPI/BIP

(Mop/s/proc)

Program	MPI/BIP	ZeroCopyMPI/PM
IS (Class S)	0.49	2.23
IS (Class A)	2.02	2.45
LU (Class S)	21.47	28.23
LU (Class A)	*not reported*	24.26

Chapter 5

Conclusion

In this report, we have designed the implementation of MPI using a zero copy message primitive, called Zero Copy MPI. A zero copy message primitive requires that the message area must be pinned down to physical memory. We have shown that such a pin-down operation causes deadlock in the case that multiple simultaneous requests for sending and receiving consume pin-down area from each other, preventing further pin-downs. To avoid such a deadlock, we have introduced the following techniques i) separation of maximum pin-down memory size for send and receive to ensure that at least one send and receive may always be active concurrently, and ii) delayed queues to handle the postponed message passing operations.

The performance results show that 13.16 usec latency and 98.81 MBytes/sec maximum bandwidth is achieved on 200 MHz Intel Pentium Pro with Myricom Myrinet. In comparison with other MPI implementations using not only point-to-point performance but also NAS parallel benchmarks, we would conclude that our Zero Copy MPI achieves good performance comparing with other MPI implementations on commodity hardware and the myrinet network.

This report contributes to the general design of the implementation of a message passing interface not only for MPI but also others on top of a lower level communication layer that has a zero copy message primitive with pin-down memory. The assumption of this lower level communication primitive is getting to be common practice. In fact, there are some standardization activities such as VIA, the Virtual Interface Architecture[5], and ST, Scheduled Transport[13].

Bibliography

[1] http://now.cs.berkeley.edu/AM/lam_release.html.

[2] http://lhpca.univ-lyon1.fr/bip.html.

[3] http://now.cs.berkeley.edu/Fastcomm/MPI/.

[4] http://science.nas.nasa.gov/Software/NPB/.

[5] COMPAQ, Intel, and Microsoft. Virtual Interface Architecture Specification Version 1.0. Technical report.

[6] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos Damianakis, and Kai Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *HOT Interconnects V*, pages 37–46, 1997.

[7] W. Gropp and E. Lusk. Mpich working note: Creating a new mpich device using the channel interface. Technical report, Argonne National Laboratory, 1995.

[8] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. User-level Parallel Operating System for Clustered Commodity Computers. In *Proceedings of Cluster Computing Conference '97*, March 1997.

[9] Yutaka Ishikawa. Multi Thread Template Library – MPC++ Version 2.0 Level 0 Document –. Technical Report TR–96012, RWC, September 1996.

[10] Mario Lauria and Andrew Chien. MPI-FM: High Performance MPI on Workstation Clusters. In *Journal of Parallel and Distributed Computing*, 1997. http://lhpca.univ-lyonl.fr/PUBLICATIONS/pub.

[11] 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.

[12] Loic Prylli and BernardTourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *Workshop PC-NOEW, IPPS/SPDP'98*, 1998. http://lhpca.univ-lyonl.fr/PUBLICATIONS/pub.

[13] T11.1. Information Technology – Scheduled Transfer Protocol(ST), Working Draft. Technical report.

[14] Hiroshi Tezuka. PM Application Program Interface Ver. 1.2. http://www.rwcp.or.jp/lab/pdslab/pm/pm4_api_e.html.

[15] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In Peter Sloot Bob Hertzberger, editor, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717. Springer-Verlag, April 1997.

[16] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. April 1998. To appear at IPPS '98.