Implementing Message Passing Communication with a Shared Memory Communication Mechanism

by

Kenji Morimoto

A Master Thesis

Submitted to the Graduate School of the University of Tokyo in Partial Fulfillment of the Requirements for the Degree of Master of Science in Information Science

March 1999

Abstract

This thesis describes a high-performance implementation of the Message Passing Interface (MPI) library based on a shared memory communication mechanism. Our implementation, called MPI/MBCF, combines two protocols to utilize the shared memory communication mechanism: the write protocol and the eager protocol. In the write protocol, *Remote Write* is used for communication with no buffering. In the eager protocol, *Memory-Based FIFO* is used for buffering by the library. These two protocols are switched autonomously according to the precedence of send and receive functions.

The performance of the MPI/MBCF was evaluated on a cluster of workstations. We measured the round-trip time and the peak bandwidth, and executed the NAS Parallel Benchmarks. The results show that a message passing library achieves high performance by using a shared memory communication mechanism.

Acknowledgments

I am very grateful to Professor Kei Hiraki for guiding me with a lot of helpful advice. I would like to express my gratitude with all my heart to Mr. Takashi Matsumoto. He gave me many useful suggestions. He built the whole infrastructure of my work. And he supplied much functionality to the infrastructure in response to my requests. I also wish to give my thanks to all the members of Hiraki Laboratory for their help and encouragement.

Contents

1 Introduction **1**

2 Preliminaries **4**

2.1 Message Passing Interface Standard 4

2.1.1 Overview of the MPI Standard 4

2.1.2 Definitions and Requirements of the MPI Standard for Point-to-point Communication Functions 5

2.2 MBCF: Memory-Based Communication Facilities 8

2.2.1 Features of the MBCF 8

2.2.2 Functionality of the MBCF 10

2.2.3 Performance of the MBCF 12

3 Message Passing Library on Shared Memory **14**

3.1 Implementation of Point-to-point Communication Functions . . . 14

3.1.1 Ideas for Point-to-point Communication Functions 15

3.1.2 Details of Implementation of Point-to-point Communication 17

3.1.3 Execution Sequence of Point-to-point Communication . . 25

3.2 Implementation of Other Functionality 26

4 Performance Evaluation **30**

4.1 Conditions of Evaluation 30

4.2 Fundamental Performance 31

4.2.1 Round-trip Time 31

4.2.2 Peak Bandwidth 32
4.3 Performance for the NAS Parallel Benchmarks 35
4.3.1 NAS Parallel Benchmarks 35
4.3.2 Conditions for the NPB 36
4.3.3 Results of Execution 37
4.3.4 Summary of Results for the NPB 40

5 Related Works **43**
5.1 MPI Implementations on General Platforms 43
5.1.1 MPICH 43
5.1.2 LAM/MPI 43
5.1.3 CHIMP/MPI 44
5.2 MPI Implementations on MPPs 45

6 Conclusion **46**

Chapter 1

Introduction

There are two communication models widely used for parallel machines with distributed memory: the message passing model and the shared memory model. In the message passing model, a communication path is established between each pair of tasks, and communication among tasks is performed by applying *send* and *receive* operations to those paths. This model is an abstraction of transmission media, that is, interprocessor communication networks. In this model, interprocessor communication networks are considered as communication paths among tasks. In the shared memory model, on the other hand, address spaces of all tasks are mapped into a unified address space, and *read* and *write* operations[1] to that shared space correspond to communication. This model is an abstraction of communication targets, that is, processors' memory spaces. In this model, data transmission among processors is considered as accesses to remote processors' memory. This model considers address spaces as targets, and is called 'memory-based'.

When these two models are considered as communication models, they are exchangeable; one can emulate the other. Thus the two models are equivalent in expressiveness. So far, the message passing model is widely used because the

[1] These operations are not necessarily fine-grain memory accesses by a *load* or *store* machine instruction.

message passing model is believed to be more efficient, and many implementations based on that model are supplied as libraries. This is because (1) usual shared memory communication mechanisms provide fine-grain shared memory access methods alone, and translate each fine-grain access like wordwise load and store into fine-grain actual transmission, (2) shared memory communication mechanisms are abstracted with the message passing model when viewed from users, and (3) not remote write but remote read is mainly used because of the use of invalidation-based coherent caches, so data transmission is delayed until the data becomes needed. These problems are not caused by the shared memory model itself but by implementations, and can be settled with an improved implementation of the model. In addition, in order to improve communication performance such as latency and bandwidth, it is essential to utilize memory-oriented architectural supports, like MMU and cache memory. The communication based on the shared memory model can receive benefit from these supports more directly than the communication based on the message passing model. For this reason, parallel machines should provide high-performance communication functionality based on the shared memory model, rather than based on the message passing model.

From the above consideration, we hold conjecture that message passing communication implemented with a high-performance shared memory communication mechanism gives better performance than that implemented with a message passing communication mechanism. In this study, we verify our claim through implementation and experiments. We chose the Message Passing Interface (MPI) Ver. 1.2 [7, 8] for a message passing communication library to be implemented, and used the Memory-Based Communication Facilities (MBCF) [13] as a base communication mechanism. The MPI library, called MPI/MBCF, has been implemented on the general-purpose scalable operating system SSS–CORE [10]. Shared memory communication operations are not applicable directly to message passing communication, since the message passing model does not contain the idea of 'remote address'. We supply notification of remote addresses for message passing communication with small overhead.

The rest of this thesis is organized as follows. Chapter 2 provides a summary of the standard of the message passing interface we have implemented, and describes the shared memory communication mechanism we employed for the basis of the MPI/MBCF. Chapter 3 explains the key point of the MPI/MBCF in detail, and then describes other parts of the implementation in brief. The performance evaluation of the MPI/MBCF is shown in Chap. 4. Related works are described in Chap. 5, and we conclude with a summary in Chap. 6.

Chapter 2

Preliminaries

This chapter first presents an outline of the standard of the message passing interface we have implemented. And then, the shared memory communication mechanism is introduced which we employed for the basis of our implementation.

2.1 Message Passing Interface Standard

2.1.1 Overview of the MPI Standard

The Message Passing Interface (MPI) [7, 8] is a widely used standard interface for writing message passing programs, especially on parallel machines with distributed memory. It is standardized at the Message Passing Interface Forum, and the standardization process is still in progress. There are two standard interfaces for now: MPI-1.2 and MPI-2; the latter standard includes the former.

Unlike the Parallel Virtual Machine (PVM) [22], which is another widely used standard for writing message passing programs, the MPI does not define an executing environment but just defines an interface for programming. The interface is defined for C and FORTRAN 77 in MPI-1.2, and for C++ in MPI-2 additionally. MPI-1.2 includes:

- Point-to-point communication

- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Environmental management and inquiry
- Profiling interface

In addition, MPI-2 includes:

- Process creation and management
- One-sided communication
- External interfaces
- Parallel I/O

For this study, we have implemented MPI-1.2 but not MPI-2. The reasons of this choice are (1) 'process creation and management', 'external interfaces', and 'parallel I/O' are not important factors specific to the message passing model, (2) 'one-sided communication' is not a component of the message passing model at all, and is introduced into the MPI standard in very limited and unnatural form, and (3) MPI-2 is not a widespread standard yet, and there are few applications which practically use functions specific to MPI-2.

2.1.2 Definitions and Requirements of the MPI Standard for Point-to-point Communication Functions

Among 129 functions defined in MPI-1.2, about 40 functions have effects of starting or completing communication. The performance of these functions dominates the performance of the entire MPI library. Especially, the point-to-point non-blocking send and receive functions, `MPI_Isend()` and `MPI_Irecv()`[1], and

[1] In this thesis, notation of MPI functions and constants follows the C manner.

the completion function, `MPI_Wait()`, are fundamentals of other communication functions. 'Non-blocking' means that the function just starts an operation and should be paired with such a completion function as `MPI_Wait()`. Most communication functions in the standard can either be implemented with these three functions or be explained on the analogy of them.

In the MPI standard, `MPI_Isend()` and `MPI_Irecv()` are declared as follows.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request);
```

Almost every MPI function returns an integer-typed error code. `buf` is the address of the send [receive] buffer. `count` is the number of elements in the send [receive] buffer. `datatype` is the type of each element in the send [receive] buffer. `dest` [`source`] is the rank of the destination [source] process. `tag` is the identification tag of the message to be sent [received]. `comm` is the *communicator* (described below), and `request` is a pointer to the identifier to be assigned for the pending send [receive] request.

The objects typed `MPI_Datatype`, `MPI_Comm`, or `MPI_Request` are 'opaque objects'. Each opaque object is either an integer indexing an array of structures or a pointer to a structure. Memory spaces for actual structures are allocated by the MPI library.

The *communicator* specifies the communication context for a communication operation. Messages are always received within the context they were sent, and messages sent in different contexts do not interfere. The communicator also specifies the group of processes which participate in the communication with this communication context. Processes in a group are ranked from 0 to $n-1$, where n is the number of processes in the group. The ranks of one process in two different communicators are not equal in general.

To specify a destination or a source process, the communicator and the rank of the target process in that communicator are used. In order to distinguish messages exchanged between a certain pair of a sender and a receiver, an identification tag is attached to a message. A send operation matches a receive operation only if all of the following four conditions are satisfied:

1. Two operations specify the same communicator
2. The send operation specifies the process as a destination which handles the receive operation in question
3. The receive operation specifies the process as a source which handles the send operation in question, or specifies `MPI_ANY_SOURCE` as a source
4. Two operations specify the same tag number, or the receive operation specifies `MPI_ANY_TAG` as a tag

The wild cards for a rank and a tag are allowed only in a receive operation. For a communicator, a wild card can be used in neither operations.

The MPI library must guarantee the arrival of messages. As to the order, the library must prevent overtaking of messages if they are equivalent in matching conditions. In other words, the library must guarantee:

- If a sender sends two messages in succession to the same destination, and both match the same receive operation, then this receive operation does not receive the second message if the first one is still pending
- If a receiver posts two receive operations in succession, and both match the same message, then the second receive operation does not be satisfied by this message if the first one is still pending

Furthermore, the library must not prevent overtaking of messages if it is not prohibited with the above two conditions, and must guarantee progress of communication.

Send operations are classified into four categories according to their behavior in the case where no matching receive operation has been posted.

Standard mode The library moves the message to the buffering area provided by the library if there is some room, and releases the send buffer as soon as possible

Buffer mode The library moves the message to the buffering area provided by the user, and releases the send buffer immediately

Synchronous mode The library does not need a buffering area, and waits for the issue of the matching receive before releasing the send buffer

Ready mode The library does not need a buffering area and may generate an error, since it assumes that the matching receive has already been posted

`MPI_Isend()` is a standard mode non-blocking send function. The MPI standard does not put a lower limit on the size of the buffering area provided by the library for standard mode send functions. It is possible for implementors not to provide the buffering area, and to treat standard mode functions as if they were synchronous mode ones. If the buffering area for standard mode is small or not provided, however, careless users' programs may fall into deadlock.

2.2 MBCF: Memory-Based Communication Facilities

2.2.1 Features of the MBCF

The Memory-Based Communication Facilities (MBCF) [13] is a software-based mechanism for accessing remote memory. It is based on the design of the Memory-Based Processor (which is a hardware-based mechanism for remote memory accesses) [12] and of the Strategic Memory System (which is a highly-functional distributed shared memory system) [11]. Details of the MBCF are as follows.

1. *Direct access to remote memory*

   The MBCF communicates by reading from and writing to remote memory

directly, not by sending messages to a fixed buffer for communication. This reduces excessive copy operations of messages.

The address of remote memory is specified in the logical address form. Therefore, <Node ID>:<Task ID>:<Logical Address> can be used as a global memory address.

2. *Efficient protection and address translation*

   By making good use of architectural memory management mechanisms such as MMU, protection of memory and translation of logical addresses are achieved at processor-clock-level speed.

3. *Use of off-the-shelf network hardware*

   The MBCF is a software-based mechanism, and does not need such dedicated communication hardware as many of MPPs have. In order to achieve high performance, however, it is preferable that the following mechanisms are provided.

   - Switching of address spaces with small overhead
   - TLB which allows many contexts to be mixed
   - Page alias capability
   - High-speed processor cache with physical address tags

   All of these mechanisms are available on most of the latest microprocessors.

4. *Channel-less communication*

   Unlike channel-oriented communication such as user memory mapping with the Myrinet [24], the MBCF achieves virtualization and protection dynamically by system calls for communication.

5. *Highly-functional operations for remote memory*

   Since a receiver of an MBCF packet handles it with software, such compound operations as *swap*, *FIFO write*, and *fetch and add* are available as well as *read* and *write*.

6. *Guarantee for arrival and order of packets*

   Because the MBCF system takes care of lost or out-of-order packets, users can make communication as if they were on a reliable network.

2.2.2 Functionality of the MBCF

From among various operations of the MBCF, *Remote Write* and *Memory-Based FIFO* are employed as the basis of our MPI library implementation.

Remote Write

Remote Write is used to write data directly to a remote address space.

To use *Remote Write*, the sending user should invoke an MBCF system call and issue an MBCF packet. The header of this packet holds the address to which the data is written as well as the receiver's node ID and task ID. This address is represented in the receiving user's logical address space. The packet can carry as long data as the network permits in one *Remote Write* invocation.

When the packet arrives at the receiver's communication hardware, the system-level trap handler awakes and tries to write that packet's data to the specified area. This trial to write fails if (1) the destination task does not exist, (2) the destination task does not permit other tasks to modify its memory space, or (3) the destination address is not within a valid memory page.

The trap handler returns a flag[2] to the sending user which indicates whether the write operation succeeded or not, if the flag is required by the user. The system call for *Remote Write* is non-blocking, and the sending user can examine this flag to know the completion of the write operation.

The receiving user finds out the completion of the write operation by observation of the contents of the memory area to be modified, or by some other implicit context.

[2]The mechanism for returning a flag is just the same as *Remote Write*.

Memory-Based FIFO

Memory-Based FIFO is used to send data to a remote FIFO-queue (ring buffer). The buffering area for a FIFO-queue is taken from the receiving user's address space, which is specified by the receiving user in advance via an MBCF system call. The user can register as many queues as space permits.

Memory-Based FIFO, a remote write operation to this queue, is one of variations of *Remote Write*. The sending user should issue an MBCF packet of *Memory-Based FIFO* in much the same way as the issue of *Remote Write*. These two operations are distinguished by the command field in the header. The destination address of the *Memory-Based FIFO* packet points to a structure which holds boundary information of the objective FIFO-queue.

The system-level trap handler on the receiver's node manages this boundary information, and tries to enqueue the packet's data. This trial fails if there is not enough room in the buffering area, in addition to for above-mentioned reasons of protection and security.

The trap handler returns a flag to the sending user which indicates whether the enqueue operation succeeded or not, if the flag is required by the user. The system call for *Memory-Based FIFO* is non-blocking, and the sending user can examine this flag to know the completion of the enqueue operation.

The receiving user dequeues data by invoking an MBCF system call. Boundary information of the FIFO-queue is managed in this system call.

In order to facilitate continuous issues of *Memory-Based FIFO*, the MBCF provides two options for its user. One option is named *reply on failure*, which suppresses the issue of the status flag if the operation (in this case, the enqueue operation) successfully completes. This option enables a user to issue a series of *Memory-Based FIFO* packets suppressing all flags except one for the last packet. The other option is named *enqueue eagerly*. This option enables a user to issue *Memory-Based FIFO* eagerly without confirming that the preceding enqueue operations complete successfully. This behavior is achieved by rejecting all the following enqueue operations but an explicit retrying enqueue operation once

an enqueue operation fails by the reason of exhaustion of the FIFO-queue.

Memory-Based FIFO can be considered as a highly-functional message passing mechanism accessible directly from the user-level.

2.2.3 Performance of the MBCF

The performance of the MBCF was evaluated on a cluster of workstations connected with a 100BASE-TX network. The following machines were used for measurement: Axil 320 model 8.1.1 (Sun Microsystems SPARCstation 20 compatible, 85 MHz SuperSPARC $\times$ 1), Sun Microsystems Fast Ethernet SBus Adapter 2.0 on each workstation, SMC TigerStack 100 5324TX (non-switching HUB), and Bay Networks BayStack 350T (switching HUB with the full-duplex mode). The operating system used for measurement is the SSS–CORE Ver. 1.1a [10]. The one-way latency and the peak bandwidth between two nodes were measured.

The one-way latency is the time from the invocation of a system call for a remote access at the source task to the arrival of the data at the destination task, including the overhead of reading the data. Table 2.1 shows the one-way latency of *Remote Write* (MBCF_WRITE) and *Memory-Based FIFO* (MBCF_FIFO) for various data-sizes on a non-switching HUB.

The peak bandwidth is measured by invoking remote accesses continuously. Table 2.2 shows peak bandwidth of *Remote Write* and *Memory-Based FIFO* for various data-sizes both on a non-switching HUB in the half-duplex mode and on a switching HUB in the full-duplex mode.

Both of the results show that the performance of the MBCF is very close to that of the network itself. The MBCF is superior in performance to the communication functions of MPPs, which have dedicated communication hardware of higher-potential [14].

Table 2.1: One-way latency of MBCF with 100BASE-TX in microseconds

data-size (bytes)	4	16	64	256	1024
MBCF_WRITE	24.5	27.5	34.0	60.5	172.0
MBCF_FIFO	32.0	32.0	40.5	73.0	210.5

Table 2.2: Peak bandwidth of MBCF with 100BASE-TX in Mbytes/s

data-size (bytes)	4	16	64	256	1024	1408
MBCF_WRITE, half-duplex	0.31	1.15	4.31	8.56	11.13	11.48
MBCF_FIFO, half-duplex	0.31	1.14	4.30	8.53	11.13	11.45
MBCF_WRITE, full-duplex	0.34	1.27	4.82	9.63	11.64	11.93
MBCF_FIFO, full-duplex	0.34	1.26	4.80	9.62	11.64	11.93

Chapter 3

Message Passing Library on Shared Memory

This chapter describes our implementation of the MPI called MPI/MBCF. Two point-to-point communication functions, send and receive, are first explained in detail, because their performance has a strong influence upon the performance of the entire library. And then, other parts of the implementation are summarized.

3.1 Implementation of Point-to-point Communication Functions

Two point-to-point communication functions, `MPI_Isend()` and `MPI_Irecv()`, are fundamentals of other communication functions, as mentioned above. In order to implement these two functions, we combined two protocols for actual communication by the library. One is the *write* protocol which uses *Remote Write* for communication, and the other is the *eager* protocol which uses *Memory-Based FIFO*. These two protocols are switched dynamically and autonomously according to the precedence of matching send and receive operations. The following subsections give the explanations of these protocols.

3.1.1 Ideas for Point-to-point Communication Functions

It is not a novel idea to apply a shared memory communication mechanism to message passing communication. The MPICH [9], which is implemented and distributed as a model MPI implementation at the Argonne National Laboratory and the Mississippi State University, employs the following three protocols.

Eager protocol The sender sends the message header and data to the receiver aiming at a pre-fixed location. The receiver takes out the header and examines matching of the message and pending receive requests. And then, the receiver takes out the data to the receive buffer if a matching receive has been posted, or else to a temporary buffer.

Rendezvous protocol The sender first sends the message header alone. The receiver examines matching of the header and pending receive requests. And then the receiver posts a request for the data to the source process if a matching receive has been posted, or else it waits for an issue of a matching receive. Finally the sender sends the message data to the receiver, with a remote memory operation if available[1]. This protocol restricts the behavior of a standard mode send operation as if it were a synchronous mode send operation[2].

Get protocol The sender first sends the message header alone. The receiver examines matching of the header and pending receive requests, and directly read the data from the source process with a remote read operation. The word 'get' is a dialect of 'remote read'. This protocol gets the same restriction and improvement as the rendezvous protocol gets.

Although the rendezvous and get protocols use shared memory communication operations, they inevitably bring an overhead of a round trip of control messages

[1] Although there is no description of the use of a remote write in [9], it is obvious that a remote write is applicable. The rendezvous protocol is originally introduced just for safety.

[2] There is a modified protocol to overcome this defect; the receiver allocates a temporary buffer if the message header comes from the sender before the matching receive, and then receives the data in that buffer.

after the data gets ready to be sent[3].

In order to utilize shared memory communication operations, it is essential to notify the address of the send [receive] buffer to the destination [source], because point-to-point communication functions in the MPI standard do not accept a remote address as an argument. There is one possible protocol; the receiver notifies the address of the receive buffer and the sender replies with a remote write operation. According to the above three protocols, matching of send and receive operations is examined only at the receiver's side. By allowing matching at the sender's side, we introduce this new protocol.

Write protocol The receiver sends the header to the source process. The sender examines matching of the header and pending send requests, and sends the message data to the receiver with a remote write operation.

This protocol is not self-contained for the following two reasons.

1. In the same way as the rendezvous protocol, the write protocol restricts the behavior of a standard mode send operation as if it were a synchronous mode send operation. Although this defect can be overcome by allocating a temporary buffer at the sender, it is not efficient, and it does not solve the problem of delayed transmission of the data until the issue of the matching receive.

2. If the receiver specifies `MPI_ANY_SOURCE` as a source, it cannot send the header to just one source process. It is still possible for the receiver to broadcast the header to all processes in the specified communicator, but this needs complicated interprocess arbitration[4].

The solution for these problems described in this thesis is to combine the write protocol with the eager protocol. Under the combined protocols, the sender and the receiver act as follows.

[3] A remote read operation can be considered as a set of a control message and a replying remote write message.

[4] *Interprocess* arbitration usually implies *internode* communication.

- The receiver sends the header to the source process as a request for sending (according to the write protocol), if the matching message has not arrived yet and if the source process is uniquely specified.

- The sender sends the message data to the receiver with a remote write operation (according to the write protocol) if the matching request has arrived. Or else the sender sends the message header and data to the receiver aiming at a pre-fixed location (according to the eager protocol).

The following is a straightforward explanation of our strategy; when the data gets ready to be sent, the sender immediately sends that data anyway; the receiver encourages the source process to send data directly.

The MPICH's three protocols are exclusive. They are not used together at the same time, and one is chosen according to some static conditions such as the length of data. The MPI/MBCF's two protocols are switched dynamically and autonomously. It depends on the dynamic precedence of the send and the receive operations which protocol is actually chosen. Each of the sender and the receiver may choose a different protocol from the other's at the same time. Even if so, finally one protocol becomes rejected and the other accepted in an autonomous way, as described in the next subsection.

3.1.2 Details of Implementation of Point-to-point Communication

Based on the above strategy, the standard mode non-blocking send function, `MPI_Isend()`, and the non-blocking receive function, `MPI_Irecv()`, are implemented with the MBCF.

Communication in the Write Protocol

The send function first examines whether the matching request for sending has arrived or not. If it has arrived already, the sender obtains the address of the receive buffer. And then the sender transmits the data to that buffer directly

with *Remote Write* of the MBCF, as illustrated with Fig. 3.1. Solid lines denote data transfer triggered by the send function. In this case, there is no message buffering by the MPI/MBCF.

The receive function, on the other hand, first examines whether the matching message has arrived or not. If it has not arrived yet, the receiver posts a request for sending to the source process and notifies the sender of the address of the receive buffer. This request is sent using *Memory-Based FIFO*. The MPI/MBCF provides as many FIFO-queues for requests as MPI processes exist. The receiver posts a request to the i-th FIFO-queue, where i is the identification number of the receiver process[5]. The set of FIFO-queues for requests is separated from that for communication messages. By arranging multiple FIFO-queues, the MPI/MBCF prevents various packets from mixing, and makes it easy to handle each queue.

Communication in the Eager Protocol

If the send function is invoked before the receive function, the sender cannot obtain the address of the receive buffer, and so cannot send data to that buffer directly. In this case, the MPI standard recommends a library to move the data to a buffering area as soon as possible. In the MPI/MBCF, the sender sends the message with *Memory-Based FIFO* to the receiver, as shown in Fig. 3.2. Solid lines denote data transfer triggered by the send function, and a broken line by the receive function.

The FIFO-queue for messages at the receiver's side acts as a buffering area for standard mode send functions. The MPI/MBCF provides as many FIFO-queues for messages as MPI processes exist. In order to prevent mixing of messages from various source processes, the sender sends the message to the i-th FIFO-queue, where i is the identification number of the sender process. There is no requirement in the MPI standard for the order or the fairness of

[5]The identification number of a process is assigned from 0 to $N-1$ where N is the number of all existing processes. This number is equal to the rank of that process in the global communicator `MPI_COMM_WORLD`.

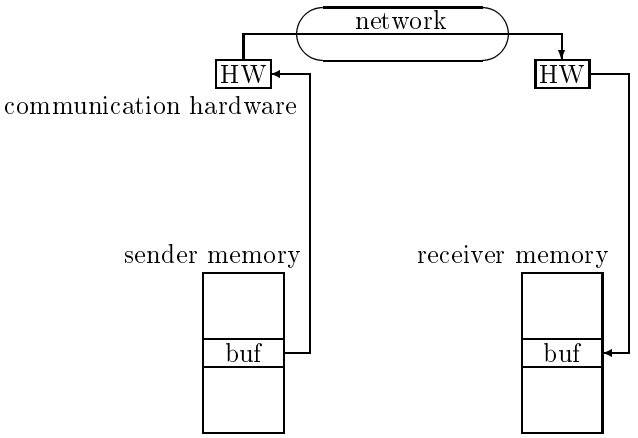

Figure 3.1: Communication with no buffering

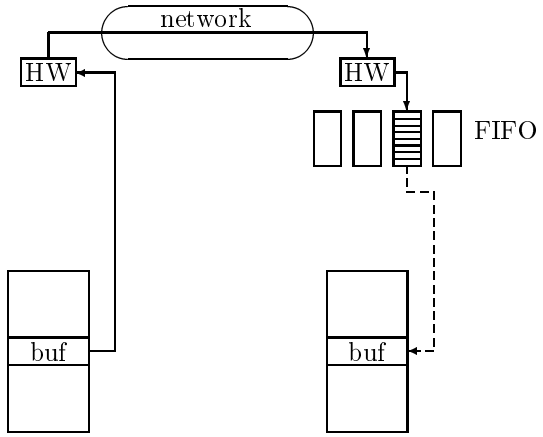

Figure 3.2: Communication with single buffering

handling messages from different sources. Therefore, it does not cause complicated arbitration to maintain multiple FIFO-queues for messages, even if a receive function specifies `MPI_ANY_SOURCE` as a source.

Buffering for Intertwined Communication

The above two methods can handle point-to-point communication successfully if the order of the sender's send operations corresponds with that of the receiver's receive operations. There may be two intertwined matching pairs of operations, however, when these two pairs specify different tags or communicators from each other. The following is an example of such operations, where `MPI_Send()` and `MPI_Recv()` are the blocking versions of `MPI_Isend()` and `MPI_Irecv()`, respectively.

```
if (my_rank == source) {
  MPI_Send(buf1, count, MPI_INT, dest, tag1, comm);
  MPI_Send(buf2, count, MPI_INT, dest, tag2, comm);
} else if (my_rank == dest) {
  MPI_Recv(buf2, count, MPI_INT, source, tag2, comm, &status);
  MPI_Recv(buf1, count, MPI_INT, source, tag1, comm, &status);
}
```

When the FIFO-queue for messages contains such disordered messages, the receive function should first dequeue non-matching messages, and then it dequeues the matching message to the receive buffer. The receiver allocates a temporary buffer for a message which is dequeued from the FIFO-queue but cannot be written to the receive buffer. The temporarily buffered message is linked to the i-th list of non-matching messages, where i is the identification number of the sender process. The receive function first searches the list of such messages for the matching one, and then it examines the FIFO-queue. The behavior of the temporarily buffered message is illustrated with Fig. 3.3. Solid lines denote data transfer triggered by the send function, a broken line by the matching receive function, and a dotted line by the non-matching receive

function.

This method increases the number of data copy operations and adds the overhead of managing a list of non-matching messages to the overhead of the receive function, and so reduces the performance of the receive function. In most of MPI applications and all of the MPI functions of the MPI/MBCF, however, the order of send operations corresponds with that of receive operations. Thus this method will not be used so often that it affects the total performance.

Elimination of Race Condition

When matching send and receive functions are invoked at about the same time, the message and the request pass each other. To solve this race condition, the sender assigns serial numbers to messages. Serial numbers are separately managed according to the identification number of the destination process. Thus a series of numbers is shared by the messages enqueued to the same FIFO-queue. By these numbers, the sender judges the freshness of requests, and does not reply to a stale request. The detailed usage of these serial numbers is described later.

Serial numbers are assigned also to requests for sending by the receiver. They are used for detecting disappearance of requests. Although the MBCF guarantees the arrival of packets, it cannot enqueue a request to a FIFO-queue if the queue is full, and then it discards the request. For simplicity, the receiver does not require the MBCF to return a status flag for the request, nor retry to post the discarded request. In addition, when the sender dequeues a non-matching request, it does not construct a list of requests but simply discards that request. These simple policies cause disappearance of requests. If any request has been disappeared, the sender must not reply to the succeeding requests, so it sends messages not with *Remote Write* but with *Memory-Based FIFO* to the FIFO-queue of the receiver. After dequeuing messages from the FIFO-queue, the receiver posts all unsatisfied requests again in order to keep them fresh, even if no request has disappeared. This enables the sender to resume replying to requests.

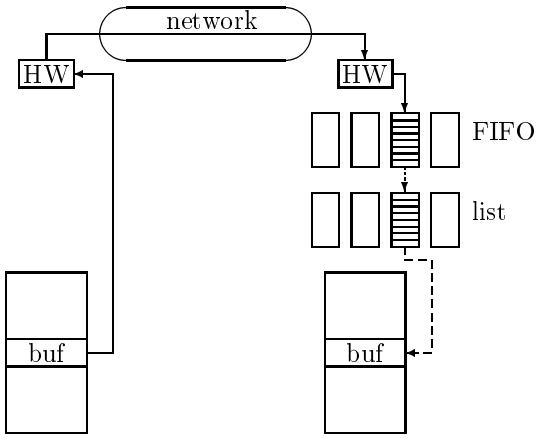

Figure 3.3: Communication with double buffering

The serial numbers assigned to requests are also used for handling receive functions with `MPI_ANY_SOURCE`. When `MPI_ANY_SOURCE` is specified as a source in a receive function, the receiver does not post a request even if a matching message has not arrived yet. This is because, as mentioned above, complicated interprocess arbitration becomes needed if the receiver broadcasts a request. Not the sender but the receiver serializes the messages from various sources which match the '`MPI_ANY_SOURCE` receive', and fixes the actual source. As long as the actual source is not fixed for the '`MPI_ANY_SOURCE` receive', the receiver must not post any more requests. This causes skipping of requests. Therefore, even after the actual source is fixed, the following requests must be ignored by the sender (or not be sent) until all requests are posted again. With serial numbers of requests, this skipping of requests can be detected at the sender.

The following is the detailed usage of serial numbers of messages and requests. A message header holds a serial number for a request as well as the serial number of itself. This extra number indicates the serial number which the receiver should assign to the next request, and is the copy of the number called *next request number*. *Next request number* is greater by one than the serial number of the request which the sender replied to most recently, and is equal to the serial number of the request which the sender expect to come next. A request for sending holds a serial number for a message as well as the serial number of itself. This extra number is equal to the serial number of the message which the receiver dequeued most recently, and is called *accepted message number*. When the sender dequeues a request, it examines:

- *Freshness*

  Whether the request's *accepted message number* is equal to the serial number of the latest sent message

- *Succession*

  Whether the request's serial number is equal to *next request number*

If these two conditions are satisfied, the sender finds that it can safely reply to the request with *Remote Write*. If not, the sender sends the message with

Memory-Based FIFO. For the receiver, the arrival of a message at a FIFO-queue implies that either of the two conditions is not satisfied. In such a case, the receiver goes through the list of pending receive operations and posts requests again.

Putting All Together

From the consideration in the previous paragraphs, the standard mode non-blocking send function, `MPI_Isend()`, is implemented as follows.

1. Examine the FIFO-queue for requests. If it contains a matching request, and the request is fresh and successive, transmit the data to the receive buffer with *Remote Memory* and finish. If not, go to step 2.

2. Transmit the message header and data to the FIFO-queue for messages of the destination with *Memory-Based FIFO*, and finish.

The non-blocking receive function, `MPI_Irecv()`, is implemented as follows.

1. Examine the list of non-matching messages temporarily buffered from the FIFO-queue at step 2 of the preceding receives. If it contains a matching message, move that message data to the receive buffer and finish. If not, append the current receive operation to the list of pending receive operations and go to step 2.

2. Dequeue a message header from the FIFO-queue for messages. If it matches one of pending receive operations, dequeue the message data to the matching receive buffer. Particularly if the dequeued message matches the current receive operation, then finish after dequeuing. If it does not match, allocate a temporary buffer, append that buffer to the list of non-matching messages, dequeue the message data to that buffer, and go to step 2. If the FIFO-queue for messages gets emptied while being consumed, go to step 3. If the FIFO-queue is empty from the beginning of step 2, go to step 4.

3. Go through the list of pending receive operations and post requests again, and finish.

4. Post just one request corresponding to the current receive operation, and finish.

The completion function, `MPI_Wait()`, has an effect of advancing communication as well as an effect of waiting for completion. For a send operation, `MPI_Wait()` retries to send the message with *Memory-Based FIFO* while the enqueue operations fail[6]. For a receive operation, `MPI_Wait()` consumes the FIFO-queue for messages and waits for the matching message by repeatedly executing step 2 and step 3 of `MPI_Irecv()`[7].

3.1.3 Execution Sequence of Point-to-point Communication

In the previous subsection, the implementation of point-to-point communication functions is described in detail, especially from the viewpoint of the data transmission algorithm for each of the sender and the receiver. As stated above, it depends on the precedence of the send and the receive functions which of the two communication operations, *Remote Write* or *Memory-Based FIFO*, is actually used for the data transmission. This subsection illustrates the execution sequence of point-to-point communication in the MPI/MBCF, according to the precedence of the two functions.

The Case Where Send Precedes Receive

In this case, the eager protocol is applied as shown in Fig. 3.4. The send function transmits the message with *Memory-Based FIFO* because no matching request has arrived. And then the receive function dequeues this message to the receive buffer.

[6] Data transmission with *Remote Write* always succeeds in the MPI/MBCF.

[7] The phrase 'go to step 4' in step 2 is replaced with 'finish'.

The Case Where Receive Precedes Send

In this case, the write protocol is applied as shown in Fig. 3.5. The receive function posts a request for sending to the sender with *Memory-Based FIFO* because no matching message has arrived. And then the send function dequeues this request, and transmits the message data in reply to the request directly to the receive buffer with *Remote Write*.

The Case Where Send Conflicts with Receive

In this case, each of the send and the receive functions first acts as if it preceded the other. The send function transmits the message with *Memory-Based FIFO*, and the receive function posts a request for sending. At the sender's side, the succeeding send function dequeues this request, and discards it since *accepted message number* of the request is older one. At the receiver's side, the succeeding receive function or the completion function dequeues the message to the proper receive buffer. The execution sequence follows the eager protocol after all, as shown in Fig. 3.6.

3.2 Implementation of Other Functionality

Most of the MPI functions implemented in the MPI/MBCF are independent of the MBCF. Only the following fundamental functions are dependent on the MBCF. The number in parentheses indicates the number of functions.

- Point-to-point blocking send or receive (5)
- Point-to-point non-blocking send or receive (5)
- Point-to-point completion (2)
- Point-to-point cancellation (1)
- Collective barrier (1)
- Initialization (1)

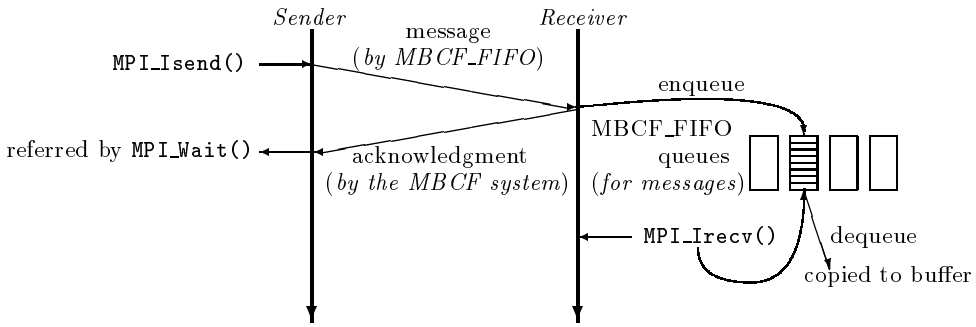

Figure 3.4: Execution sequence where send precedes receive (in eager protocol)

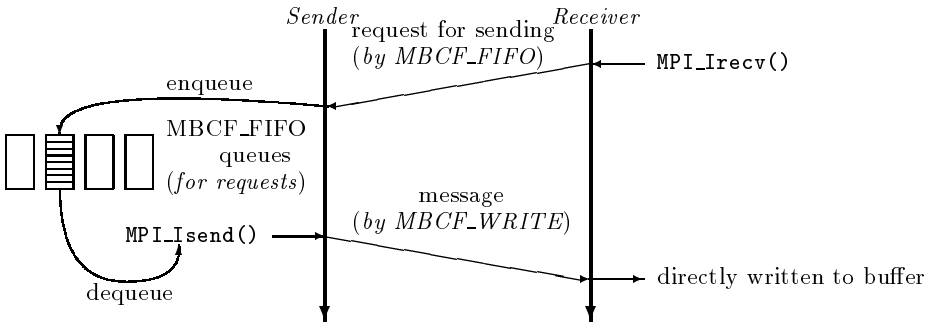

Figure 3.5: Execution sequence where receive precedes send (in write protocol)

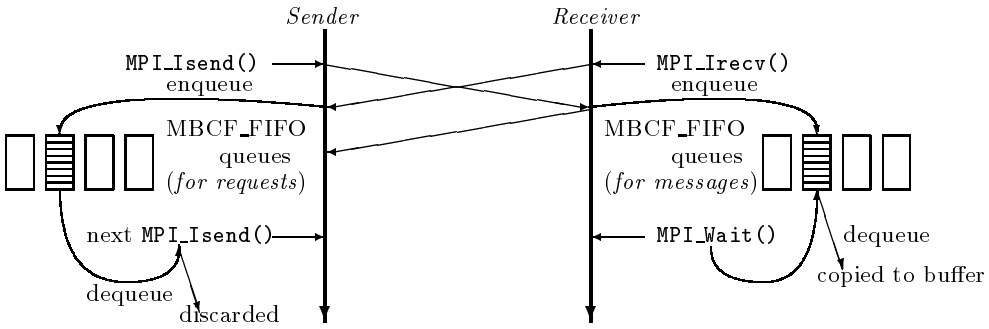

Figure 3.6: Execution sequence where send conflicts with receive (in eager protocol)

- Abortion (1)

All sorts of opaque objects are implemented as integers. Each integer is an index to a fixed-sized pre-allocated array of structures. Each structure is the entity of an opaque object, and holds information about the object. If the array of structures becomes fully used, the MPI/MBCF simply aborts. Thus the exhaustion of opaque objects causes a fatal error in the MPI/MBCF, but it rarely occurs in practical applications.

The point-to-point send functions other than the standard mode ones are implemented after the models of the standard mode ones. The point-to-point cancellation function employs *Memory-Based Signal*, which enables an MBCF user (the MPI library in this case) to interrupt another task, and to get that task to invoke a pre-registered interrupt handler. The interrupt handler for the cancellation function tries to discard the message or the request to be cancelled[8]. Other point-to-point functions are nothing but interfaces for the fundamental point-to-point functions.

The collective barrier function is implemented not by using point-to-point communication functions but by using the MBCF directly. Other collective functions are made up of point-to-point communication functions. The non-blocking receive function is utilized in order to allow the receive function to precede the send, and to make full use of *Remote Write*.

As mentioned in Sect. 2.1.2, a group or a communicator is an integer indexing to an array of structures since it is an opaque object. The same value is assigned to the communicators on all processes in the same context. This makes it easy to exchange communicators.

Process topologies are implemented by using the caching mechanism of communicators.

The abortion function uses *Memory-Based Signal* to kill other MPI processes.

[8]The MPI standard requires that a cancellation should complete whether it succeeds or fails, regardless of the status of the peer process. Thus it is not sufficient for cancelling to send a cancel notification to a remote FIFO, because the MPI/MBCF is a function-call-driven library.

Interfaces for FORTRAN 77 are provided in the form of wrapper functions written in C. Most of the wrapper functions just call the corresponding MPI functions in C, giving care to the difference between 'call by value' and 'call by reference'. Some of them need, however, conversion of arguments and returned values.

Chapter 4

Performance Evaluation

This chapter presents the results of performance evaluation of the MPI/MBCF. The conditions of the evaluation are first described. Next come the results of fundamental performance evaluation, that is, evaluation of the round-trip time and the peak bandwidth. And then the performance for arithmetic parallel applications is shown. For comparison, the results with a TCP-based MPI implementation are also shown.

4.1 Conditions of Evaluation

The same equipments as stated in Sect. 2.2.3 were used for measurement: a cluster of workstations connected with a 100BASE-TX network. The operating system used for measurement is the SSS–CORE Ver. 1.1a, which provides the MBCF.

To make a comparison, the performance of the MPICH Ver. 1.1 [9] on the SunOS 4.1.4 was also evaluated with the same equipments. The MPICH employs TCP sockets for communication when it is used on a cluster of workstations. Thus it can be said that the MPICH on a cluster of workstations is a message-based implementation of the MPI. For the SunOS 4.1.4, the network switch cannot be used in the full-duplex mode owing to limitations of the device driver.

In order to examine the effect of the write protocol, two different versions of the MPI/MBCF are used for the performance evaluation. One issues requests for sending as explained in Sect. 3.1, and the other does not. In the latter implementation, `MPI_Isend()` always transmits a message with *Memory-Based FIFO*, never with *Remote Write*. In the followings, the former implementation with requests for sending, or SendReqs, is denoted by SR for short, and the latter NSR.

4.2 Fundamental Performance

4.2.1 Round-trip Time

The round-trip time between two processes on different nodes was measured on a non-switching HUB, since there is large disadvantage and little benefit to the round-trip time on a switching HUB. In the evaluation program, two processes first invoke `MPI_Irecv()`s in advance. And then one process calls

1. `MPI_Send()`
2. `MPI_Wait()` (for `MPI_Irecv()`)

and the other

1. `MPI_Wait()` (for `MPI_Irecv()`)
2. `MPI_Send()`

repeatedly. The round-trip time is the time from the invocation of `MPI_Send()` to the end of `MPI_Wait()` on the former process. For the MPI/MBCF on the SSS–CORE, the round-trip time is measured for every iteration with a 0.5 μs-resolution counter, and the minimum value is adopted as the round-trip time. For the MPICH on the SunOS, unfortunately, only a 10 μs-resolution clock is available, so the round-trip time is measured by averaging the time for 1024 iterations, and the minimum value of the average times is adopted. Table 4.1 shows the round-trip time on every implementation for 0 byte to 4 Kbytes messages.

SR and NSR are much faster than MPICH. The difference between SR and NSR is caused by the difference of *Remote Write* and *Memory-Based FIFO*. *Remote Write* is faster as shown in Sect. 2.2.3, and is easier to manage because it does not require explicit acknowledgments. Communication with *Memory-Based FIFO* needs one more copy operation than *Remote Write* needs.

The round-trip time of SR for a 0 byte message is 71 μs. Since the one-way latency of the MBCF for a 4 bytes packet is 24.5 μs as shown in Sect. 2.2.3, the additional overhead for implementing an MPI library is small.

4.2.2 Peak Bandwidth

The peak bandwidth between two processes on different nodes was measured both on a non-switching HUB in the half-duplex mode and on a switching HUB in the full-duplex mode. In the evaluation program, one process repeatedly sends messages and the other receives. The communication time is the time from the first invocation of the send function to the barrier synchronization after the last invocation on the former process. The peak bandwidth is computed from the total message size divided by the communication time. Table 4.2 shows the peak bandwidth on every implementation for 4 bytes to 1 Mbytes messages. The results for 4 bytes to 4 Kbytes messages are extracted into a graph in Fig. 4.1. SRH and NSRH denote SR and NSR in the half-duplex mode, and SRF and NSRF denote SR and NSR in the full-duplex mode respectively.

In general, high bandwidth cannot be gained for small messages owing to the overhead of send and receive operations. The MPI/MBCF gains, however, much higher bandwidth than the MPICH, especially for small messages. This suggests that the MPI/MBCF is suitable for the applications necessarily or unnecessarily performing fine-grain communication, as well as for well-formed applications.

NSR achieves slightly higher bandwidth than SR in the half-duplex mode. This reveals that the additional packets for 'requests for sending' in SR interfere with the communication of actual data in that mode. In the full-duplex mode,

Table 4.1: Round-trip time of MPI with 100Base-TX in microseconds

message size (bytes)	0	4	16	64	256	1024	4096
SR	71	85	85	106	168	438	1026
NSR	112	137	139	154	223	517	1109
MPICH	968	962	980	1020	1080	1255	2195

Table 4.2: Peak bandwidth of MPI with 100Base-TX in Mbytes/s

message size (bytes)	4	16	64	256	1024	4096	16384	65536	262144	1048576
SRH	0.14	0.53	1.82	4.72	8.08	9.72	10.15	9.78	9.96	10.00
NSRH	0.14	0.54	1.89	4.92	8.54	10.21	10.34	10.43	10.02	9.96
SRF	0.14	0.57	1.90	5.33	10.22	11.68	11.77	11.85	11.85	11.86
NSRF	0.15	0.59	1.98	5.51	10.58	11.70	11.78	11.81	11.82	11.82
MPICH	0.02	0.09	0.35	1.27	3.54	6.04	5.59	7.00	7.77	7.07

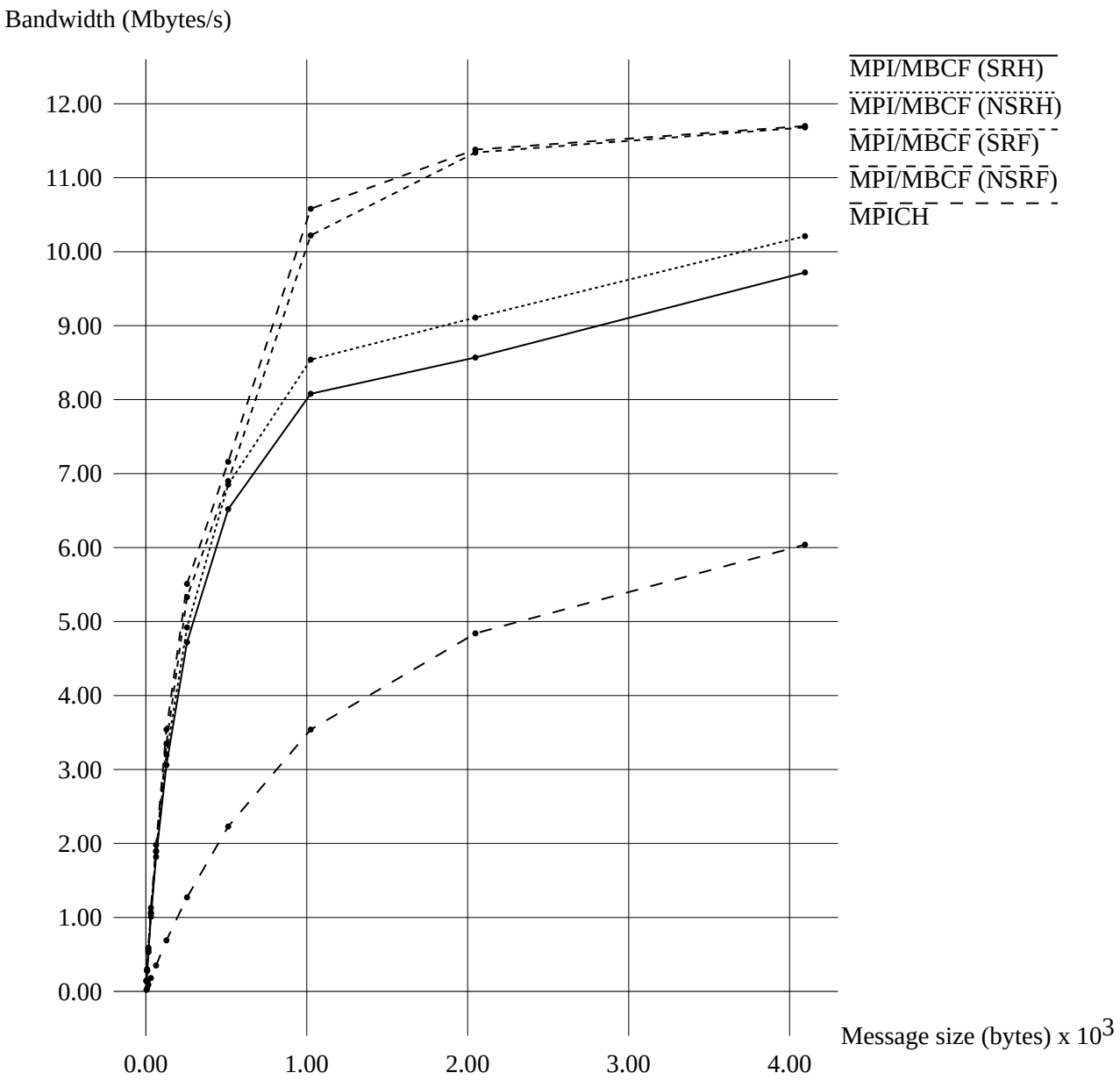

Figure 4.1: Peak bandwidth of MPI with 100Base-TX

on the other hand, there is little difference in bandwidth between SR and NSR, since requests do not interfere with actual data. Although NSR is inferior to SR with regard to the round-trip time, the difference of the round-trip time is absorbed to some extent where the peak bandwidth is concerned.

The peak bandwidth of SR is 10.15 Mbytes/s in the half-duplex mode and 11.86 Mbytes/s in the full-duplex mode. These values are close to the hardware limit of 100BASE-TX, 12.5 Mbytes/s, and to the bandwidth of the MBCF, 11.48 Mbytes/s and 11.93 Mbytes/s.

4.3 Performance for the NAS Parallel Benchmarks

4.3.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) [1, 2] is a suite of benchmarks for parallel machines. It is based on the Numerical Aerodynamic Simulation Programs, which is based at NASA Ames Research Center. The NPB 1.0 [1] defines problems, algorithms to solve problems, classes of problems, and a programming model. The NPB 2.x [2] provides solver programs written with the MPI. The NPB consists of the following five kernel programs and three computational fluid dynamics (CFD) applications.

- Kernel programs

  EP 'Embarrassingly parallel', generation of uniformly distributed random numbers and normally distributed random numbers by the multiplication congruence method

  MG Simplified multigrid kernel for solving a 3D Poisson PDE

  CG Conjugate gradient method for finding the smallest eigenvalue of a large-scale sparse symmetric positive definite matrix

  FT Fast-Fourier transformation for solving a 3D PDE

IS Large-scale integer sort

- CFD applications

  LU CFD application using the symmetric SOR iteration

  SP CFD application using the scalar ADI iteration

  BT CFD application using the 5×5 block size ADI iteration

In the NPB 2.x, IS is written in C with the MPI, and the others are written in Fortran 90 with the MPI. Each of the eight problems is classified into five classes according to its problem size configuration.

Class S For sample execution

Class W For a small-scale cluster of workstations

Class A For a medium-scale cluster of workstations

Class B For a medium-scale MPP system

Class C For a large-scale MPP system

4.3.2 Conditions for the NPB

The applications of the NPB Rev. 2.3 are executed on a switching HUB in the half-duplex mode, with each MPI process running on a different node from the others. We did not use a switching HUB in the full-duplex mode, because the MPICH on the SunOS cannot be run in that mode.

We used gcc-2.7.2.3 and g77-0.5.21 for compiling. The FT kernel program cannot be compiled with g77, and so is omitted in the followings.

The problem size is fixed to Class W, because the programs of Class A cannot be executed on SunOS 4.1.4 owing to the shortage of memory.

4.3.3 Results of Execution

Characteristics of Programs

Table 4.3 shows the characteristics of the benchmark programs. These were measured on SR with 8 (9 for SP and BT) processes by inserting counter codes into the MPI/MBCF.

The communication frequency is computed from the total amount of data (or the total number of messages) transmitted with MPI functions among all processes, divided by the execution time. The *Remote Write* availability rate is computed from the amount of data transmitted with *Remote Write*, divided by the total amount of data.

EP

Table 4.4 shows the execution time of EP. 2^{26} random numbers are computed.

In EP, processes communicate only for gathering results at the final stage of execution. Most part of the execution time is spent for floating-point calculations. Thus the result of EP is no more than the floating-point performance of the workstation.

MG

Table 4.5 shows the execution time of MG. The problem size is $64 \times 64 \times 64$, and the number of iterations is 40.

Point-to-point communication operations for messages of around 1 Kbytes are performed very frequently to exchange data across partitioning boundaries. Since the MPI/MBCF is suitable for fine-grain communication as mentioned in Sect. 4.2.2, the performance with the MPI/MBCF is much better than that with the MPICH.

All of the receive functions in MG specify a wild card `MPI_ANY_SOURCE` as a source. It causes low availability of *Remote Write* in SR. Properly speaking, however, the use of `MPI_ANY_SOURCE` is not essential in MG. By rewriting the

program without `MPI_ANY_SOURCE`, SR should achieve much better performance.

CG

Table 4.6 shows the execution time of CG. The problem size is 7000, and the number of iterations is 15.

Collective reduction operations and point-to-point communication operations for messages of around 10 Kbytes are performed. Although the communication frequency of CG is close to that of MG, the message size is larger than MG. Therefore the performance of CG is better than MG.

Additionally, the performance is improved in SR by applying *Remote Write* to the half of messages.

IS

Table 4.7 shows the execution time of IS. The problem size is 2^{20}, and the number of iterations is 10.

At each iteration, about 1 Mbytes messages are exchanged by collective all-to-all communication functions. Because the amount of computation is rather small, the performance of collective operations dominates the performance of IS more and more as the number of processes is increased. In the MPI/MBCF, functions for collective operations are written so that the receive functions are first invoked. Thus, in SR, *Remote Write* enables all-to-all functions to communicate directly, and improves the whole performance.

LU

Table 4.8 shows the execution time of LU. The problem size is $33 \times 33 \times 33$, and the number of iterations is 300.

Point-to-point communication operations for some hundred bytes messages are performed. Although the message size is small, the communication frequency is low as well. Thus the performance of LU is rather good both with

Table 4.3: Characteristics of NPB Programs

program	EP	MG	CG	IS	LU	SP	BT
communication frequency (Mbytes/s)	0.00	9.68	12.69	13.58	1.89	7.83	5.32
communication frpequency (# of messages/s)	4	4670	2138	466	1199	421	488
Remote Write availability rate (%)	51.10	0.01	53.33	99.22	13.37	49.01	47.24

Table 4.4: Execution time of NPB EP in seconds

# of processes	1	2	4	8
SR [speed-up]	121.14 [1.00]	60.51 [2.00]	30.30 [4.00]	15.15 [8.00]
NSR [speed-up]	121.15 [1.00]	60.59 [2.00]	30.30 [4.00]	15.15 [8.00]
MPICH [speed-up]	125.56 [1.00]	60.61 [2.07]	32.13 [3.91]	16.25 [7.73]

Table 4.5: Execution time of NPB MG in seconds

# of processes	1	2	4	8
SR [speed-up]	37.34 [1.00]	22.61 [1.65]	14.05 [2.66]	7.44 [5.02]
NSR [speed-up]	37.32 [1.00]	22.62 [1.65]	14.05 [2.66]	8.01 [4.66]
MPICH [speed-up]	38.81 [1.00]	31.30 [1.24]	21.01 [1.85]	13.72 [2.83]

Table 4.6: Execution time of NPB CG in seconds

# of processes	1	2	4	8
SR [speed-up]	69.16 [1.00]	37.69 [1.83]	20.94 [3.30]	11.24 [6.15]
NSR [speed-up]	69.13 [1.00]	38.54 [1.79]	21.44 [3.22]	11.81 [5.85]
MPICH [speed-up]	68.75 [1.00]	40.01 [1.72]	27.79 [2.47]	14.59 [4.71]

the MPI/MBCF and with the MPICH.

In LU, as well as in MG, the use of `MPI_ANY_SOURCE` hinders the use of *Remote Write*.

SP

Table 4.9 shows the execution time of SP. The problem size is $36 \times 36 \times 36$, and the number of iterations is 400.

Point-to-point communication operations for messages of around 10 Kbytes are performed. The communication frequency is low, and receive functions precede send functions, so SR achieves good performance above all.

BT

Table 4.10 shows the execution time of BT. The problem size is $24 \times 24 \times 24$, and the number of iterations is 200.

Point-to-point communication operations for messages of around 10 Kbytes are performed. Both of SR and NSR achieve good performance, though SR utilizes *Remote Write* at a high rate. This is because communication in BT is organized to hide latency in some degree.

4.3.4 Summary of Results for the NPB

For all programs, the MPI/MBCF achieves much better performance than the MPICH. Especially when small messages are exchanged frequently (e.g. in MG), the large overhead of the MPICH makes the performance worse, so the difference in performance between two libraries expands much more. The implementation of NSR uses *Memory-Based FIFO* alone, and has a resemblance to MPICH in that both of them are message-based implementations of the MPI. Since NSR achieves better performance than MPICH on the very same machines, *Memory-Based FIFO* on the SSS–CORE is superior to TCP on the SunOS for implementing an MPI library.

Table 4.7: Execution time of NPB IS in seconds

# of processes	1	2	4	8
SR [speed-up]	10.16 [1.00]	6.35 [1.60]	4.51 [2.25]	2.90 [3.50]
NSR [speed-up]	10.16 [1.00]	6.35 [1.60]	4.69 [2.17]	3.72 [2.73]
MPICH [speed-up]	10.25 [1.00]	7.09 [1.45]	5.61 [1.83]	4.81 [2.13]

Table 4.8: Execution time of NPB LU in seconds

# of processes	1	2	4	8
SR [speed-up]	1034.09 [1.00]	537.23 [1.92]	289.65 [3.57]	164.55 [6.28]
NSR [speed-up]	1034.56 [1.00]	541.21 [1.91]	294.00 [3.52]	169.63 [6.10]
MPICH [speed-up]	1081.51 [1.00]	611.92 [1.77]	320.70 [3.37]	185.04 [5.84]

Table 4.9: Execution time of NPB SP in seconds

# of processes	1	4	9
SR [speed-up]	1277.42 [1.00]	352.34 [3.63]	153.96 [8.30]
NSR [speed-up]	1276.39 [1.00]	352.77 [3.62]	165.01 [7.74]
MPICH [speed-up]	1391.16 [1.00]	475.27 [2.93]	231.66 [6.01]

Table 4.10: Execution time of NPB BT in seconds

# of processes	1	4	9
SR [speed-up]	617.67 [1.00]	155.19 [3.98]	67.13 [9.20]
NSR [speed-up]	617.44 [1.00]	155.21 [3.98]	67.65 [9.13]
MPICH [speed-up]	627.29 [1.00]	214.14 [2.93]	96.02 [6.53]

Compared with NSR, SR utilizes *Remote Write* to communicate directly when receive functions are invoked before send functions (e.g. in CG, IS, LU, and SP). Although the experiments were made on a half-duplex network, where requests for sending interfere with the communication of actual data in SR, SR achieves better performance than NSR. This proves that the combination of the eager protocol and the write protocol improves performance in applications, as well as fundamental performance. This also suggests that it is effective to implement a message passing library with a shared memory communication mechanism.

Chapter 5

Related Works

5.1 MPI Implementations on General Platforms

5.1.1 MPICH

The MPICH [9] is a widely used MPI implementation. It is implemented and distributed as a model MPI implementation simultaneously with the standardization process. The MPICH is available on various platforms, from a cluster of workstations to an MPP system.

As stated in Sect. 3.1.1, the MPICH employs three protocols: the eager, rendezvous, and get protocols. Although the latter two protocols can utilize shared memory communication operations, they inevitably bring an overhead of a round trip of control messages after the data gets ready to be sent. The MPICH uses, moreover, TCP sockets for communication when used on a cluster of workstations; they are not shared memory communicaton operations.

5.1.2 LAM/MPI

The LAM/MPI [5, 4] is another widely used MPI implementation available on various platforms.

The LAM/MPI is an upper layer of the LAM, which is a parallel software

environment. It uses a dedicated and complicated communication mechanism of the LAM by default. To achieve higher performance, however, it supplies TCP-based communication.

The LAM/MPI uses two protocols; the eager protocol for small messages and the rendezvous protocol (without shared memory communication) for large messages. Additionally it utilizes System V IPC on an SMP system. All of them are not internode shared memory communication operations.

5.1.3 CHIMP/MPI

The CHIMP/MPI [15, 16, 3] is an MPI implementation for various platforms with a lower layer, the CHIMP.

The CHIMP/MPI employs two protocols; PTAT and (PT)AT in their words. The basis for communication is TCP.

The PTAT protocol is a modified rendezvous protocol (without shared memory communication). In the PTAT protocol, the sender first send the message header with a part of the message data, not the message header alone. Then the receiver sends an acknowledgment message, and the sender sends the rest of the message data. If the message is small enough, the second transmission from the sender is omitted. The PTAT protocol is not identical to the eager protocol even if the message is small; the PTAT protocol requires an acknowledgment message from the library on the receiver's side, not from the communication system[1].

The (PT)AT protocol is similar to the write protocol, but does not use shared memory communication operations. In the (PT)AT protocol, the receiver posts a request for sending. And then the sender sends the message header and data aiming at a pre-fixed location, not at the receive buffer. When this protocol is applied, the receiver can safely take out the data to the receive buffer without a temporary buffer.

[1] The communication system will automatically send an acknowledgment packet for the first message from the sender anyway. The additional library-level acknowledgment is used to avoid race conditions.

These two protocols are not employed for shared memory communication operations.

5.2 MPI Implementations on MPPs

The MPIAP [20, 21] utilizes *put* and *get* of Fujitsu AP1000, AP1000+, and AP3000; it employs the eager protocol and the get protocol. The CRI/EPCC MPI [6] utilizes the Shared Memory Access library of Cray T3D; it employs the rendezvous protocol (with shared memory communication) as well as the above two protocols. Both of these two implementations cannot reduce the overhead of a round trip of control messages.

The MPI-EMX [23], which was implemented independently of but simultaneously with the MPI/MBCF [17, 18, 19], utilizes *remote memory write* of EM-X. It employs a similar protocol to the write protocol.

All of the above three are implementations with dedicated communication hardware on MPPs.

Chapter 6

Conclusion

We have implemented a fully equipped MPI library, the MPI/MBCF, with a shared memory communication mechanism, the MBCF, from scratch. The MBCF's *Memory-Based FIFO* is employed in the eager protocol for buffering by the library, and *Remote Write* is employed in the write protocol for direct transmission with no buffering. When a send function precedes a receive function, *Memory-Based FIFO* is used to send data without exchanging control messages. When a receive function precedes, *Remote Write* is used by the sender to send data directly to the receiver's address space.

The performance of the library was evaluated on a cluster of workstations connected with a 100BASE-TX network. The round-trip time was 71 μs for a 0 byte message, and the peak bandwidth was 10.15 Mbytes/s in the half-duplex mode and 11.86 Mbytes/s in the full-duplex mode. By executing the NAS Parallel Benchmarks, it was proved that the MPI/MBCF with *Remote Write* is superior in communication performance to the MPI/MBCF without *Remote Write* and to the MPICH/TCP. These results give corroborative evidence to our claim; a message passing library achieves higher performance by using a shared memory communication mechanism.

The experiments were made for two different operating systems and communication mechanisms on the very same machines with off-the-shelf hardware.

The results point out that it is possible to achieve large improvement of performance by improving the operating system and the communication library, without modifying applications. This also suggests the effectiveness of a cluster of workstations without dedicated communication hardware.

Bibliography

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994. `http://www.nas.nasa.gov/NAS/NPB/`.

[2] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995. `http://www.nas.nasa.gov/NAS/NPB/`.

[3] A. Bruce, J. Mills, and G. Smith. CHIMP version 2.0 design. Technical Report EPCC-KTP-CHIMP-V2-DESIGN 1.2, Edinburgh Parallel Computing Centre, February 1994. `http://www.epcc.ed.ac.uk/pg/CHIMP/`.

[4] G. Burns and R. Daoud. MPI primer / developing with LAM. `http://www.mpi.nd.edu/lam/`, November 1996.

[5] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. `http://www.mpi.nd.edu/lam/`, June 1994.

[6] K. Cameron, L. Clarke, and G. Smith. CRI/EPCC MPI for CRAY T3D. `http://www.epcc.ed.ac.uk/t3dmpi/Product/`, September 1995.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. `http://www.mpi-forum.org/`, June 1995.

[8] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. `http://www.mpi-forum.org/`, July 1997.

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

[10] T. Matsumoto, S. Furuso, and K. Hiraki. Resource management methods of the general-purpose massively-parallel operating system: SSS–CORE (in Japanese). In *Proc. of 11th Conf. of JSSST*, pages 13–16, October 1994.

[11] T. Matsumoto and K. Hiraki. Cache injection and high-performance memory-based synchronization mechanisms (in Japanese). In *IPSJ SIG Notes ARC-101-15, Vol. 93, No. 71*, pages 113–120, August 1993.

[12] T. Matsumoto and K. Hiraki. Distributed shared-memory architecture using Memory-Based Processors (in Japanese). In *Proc. of Joint Symp. on Parallel Processing '93*, pages 245–252, May 1993.

[13] T. Matsumoto and K. Hiraki. Memory-based communication facilities of the general-purpose massively-parallel operating system: SSS–CORE (in Japanese). In *Proc. of 53rd Annual Convention of IPSJ (1)*, pages 37–38, September 1996.

[14] T. Matsumoto and K. Hiraki. MBCF: A protected and virtualized high-speed user-level memory-based communication facility. In *Proc. of Int. Conf. on Supercomputing '98*, pages 259–266, July 1998.

[15] J. Mills, L. Clark, and A. Trew. CHIMP concepts. Technical Report EPCC-KTP-CHIMP-CONC 1.2, Edinburgh Parallel Computing Centre, June 1991. `http://www.epcc.ed.ac.uk/pg/CHIMP/`.

[16] J. Mills, L. Clark, and A. Trew. CHIMP concepts and development. Technical Report EPCC-TR94-14, Edinburgh Parallel Computing Centre, March 1994. `http://www.epcc.ed.ac.uk/pg/CHIMP/`.

[17] K. Morimoto, T. Matsumoto, and K. Hiraki. The general-purpose scalable operating system: SSS–CORE — implementation and evaluation of high performance MPI — (in Japanese). In *Proc. of 56th Annual Convention of IPSJ (1)*, pages 13–14, March 1998.

[18] K. Morimoto, T. Matsumoto, and K. Hiraki. Implementation of high performance MPI with the memory-based communication facilities (in Japanese). In *Proc. of Joint Symp. on Parallel Processing '98*, pages 191–198, June 1998.

[19] K. Morimoto, T. Matsumoto, and K. Hiraki. Implementing MPI with the memory-based communication facilities on the SSS–CORE operating system. In V. Alexandrov and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 223–230. Springer-Verlag, September 1998.

[20] D. Sitsky and K. Hayashi. Implementing MPI for the Fujitsu AP1000 / AP1000+ using polling, interrupts and remote copying. In *Proc. of Joint Symp. on Parallel Processing '96*, pages 177–184, June 1996.

[21] D. Sitsky and P. Mackerras. System developments on the Fujitsu AP3000. In P. Mackerras, editor, *Proc. of 7th Parallel Computing Workshop*, September 1997.

[22] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[23] O. Tatebe, Y. Kodama, S. Sekiguchi, and Y. Yamaguchi. Efficient implementation of MPI using remote memory write (in Japanese). In *Proc. of Joint Symp. on Parallel Processing '98*, pages 199–206, June 1998.

[24] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An operating system coordinated high performance communication library. In B. Hertzberger

and P. Sloot, editors, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717. Springer-Verlag, April 1997.