Implementation and Evaluation of MPI on an SMP Cluster

Toshiyuki Takahashi, Francis O'Carroll†, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Hiroshi Harada, Yutaka Ishikawa, Peter H. Beckman‡
Real World Computing Partnership
{tosiyuki,tezuka,hori,s-sumi,h-harada,ishikawa}@rwcp.or.jp
†MRI Systems, Inc. ‡Los Alamos National Laboratory
ocarroll@rwcp.or.jp beckman@lanl.gov

Abstract. An MPI library, called MPICH-PM/CLUMP, has been implemented on a cluster of SMPs. MPICH-PM/CLUMP realizes zero copy message passing between nodes while using one copy message passing within a node to achieve high performance communication. To realize one copy message passing on an SMP, a kernel primitive has been designed which enables a process to read the data of another process. The get protocol using this primitive was added to MPICH. MPICH-PM/CLUMP has been run on an SMP cluster consisting of 64 Pentium II dual processors and Myrinet. It achieves 98 MByte/sec between nodes and 100 MBytes/sec within a node.

1 Introduction

As SMP machines have come into wide use, they are becoming less expensive. Notably, a dual Pentium II machine is now almost the same price as a single Pentium II machine plus the cost of the additional CPU and main memory. This low cost drives construction of cluster systems using dual Pentium II machines. Another advantage is that such a cluster requires less space than a single CPU-based cluster with the same number of CPUs.

In this paper a cluster system whose components are SMPs is called an SMP cluster and a single processor based cluster is called a UP (uniprocessor) cluster. On an SMP cluster, the following programming models are taken into account:

1. Mixture of multi-threaded and message passing models
   A single process runs on each SMP node. A multi-threaded programming model is employed within a process to utilize CPUs within a node, and message passing is used for communication between nodes.
2. Multi-threaded only model
   With the support of a distributed shared memory system on an SMP cluster, a multi-thread programming model is used.
3. Message passing only model
   Running a process on each processor of an SMP node, message passing is employed to communicate with all processes.

In this paper, we consider the third model because it has the advantages of portability and understandability for MPI[9] users. MPI software written for a

UP cluster runs on an SMP cluster without any modification. Programmers do not need to specially consider how to use the SMP node's multiple processors.

We have designed and implemented a high performance MPI library for SMP clusters, called MPICH-PM/CLUMP[1]. MPICH-PM/CLUMP features facilities for zero-copy message transfer between SMP nodes and one-copy message transfer within an SMP node. These facilities reduce message copy overhead, and make high performance possible.

In this paper, first, our cluster system software, called SCore, is briefly introduced in section 2. SCore 2.x, the previous SCore version, supports only UP clusters. In section 3, SCore 2.x is adapted to an SMP cluster, and then the basic performance is evaluated. The extended SCore is called SCore 3.x.

A naive MPI implementation on SMP uses the shared memory OS facility. However, this involves two message copies. A one-copy message transfer in an SMP node is designed in section 4 to better utilize the SMP memory bandwidth. Using this feature, an MPI library for SMP clusters called MPICH-PM/CLUMP is designed and evaluated.

Section 5 evaluates MPICH-PM/CLUMP using the NAS Parallel Benchmarks. Related work is described in section 6.

2 Background

2.1 Hardware Configuration

Table 1. Specification of LBP, an SMP Cluster

Processor	Intel Pentium II
Clock	333 MHz
Chipset	Intel 440LX
Memory/Node	512 MBytes
Number of Processors/Node	2
Number of Nodes	64
Network	Myrinet M2M-PCI32C, M2LM-SW16
Network Bandwidth	full-duplex 1.28+1.28 Gbits/sec/link

Table 1 shows the hardware specification of an SMP cluster named the ACL[2] "Little Blue Penguin" Cluster (LBP) at Los Alamos National Laboratory. It consists of 64 SMP nodes each having two Pentium II processors. The nodes are connected by a Myrinet network[4] which is a high performance network. LBP has eight 16-port switches. Eight ports of each switch are linked to nodes.

[1] MPICH using PM on a CLUster of SMPs
[2] Advanced Computing Laboratory

Table 2. Specification of RWC SMPC 0, another SMP Cluster

Processor	Intel Pentium Pro
Clock	200 MHz
Chipset	Intel 450GX
Memory/Node	256 MBytes
Number of Processors/Node	4
Number of Nodes	4
Network	Myrinet M2F-PCI32C, M2F-SW8
Network Bandwidth	full-duplex 1.28+1.28 Gbits/sec/link

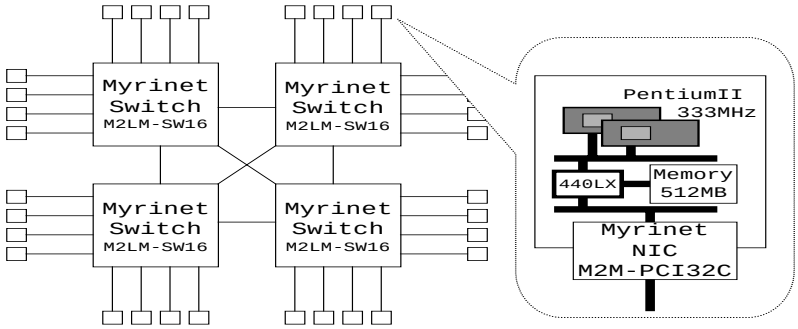

Fig. 1. Network Topology of LBP (32 Nodes)

Seven ports are used to interconnect the switches. The other port is for future expansion. Figure 1 illustrates the network topology of half (32 nodes) of LBP.

Table 2 shows the specification of another SMP cluster named RWC SMP Cluster 0 (SMPC). MPICH-PM/CLUMP supports both LBP and SMPC. As space is limited, only the results on LBP are shown in this paper.

2.2 SCore Cluster System Software

SCore is cluster system software package that realizes a multiple user environment on top of Unix kernels (including SunOS, NetBSD, and Linux). It was implemented by building a device driver and user level programs without any kernel modification. As shown in Figure 2, it consists of a communication driver and handler called PM, a global operating system called SCore, MPI implemented on top of SCore and PM called MPICH-PM, a multi-threaded template library called MTTL, and an extended C++ programming language called MPC++.

SCore assumes the SPMD execution model in which a parallel application is represented by a set of processes. All processes have the same code but may have

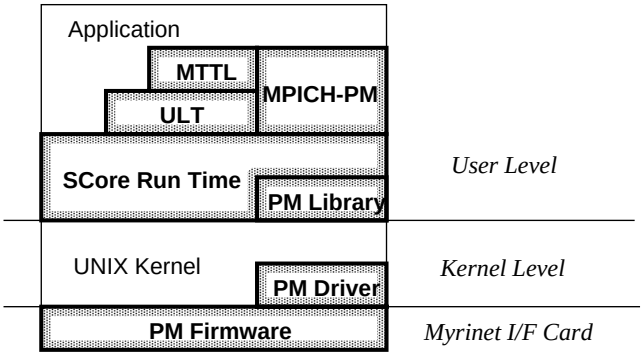

Fig. 2. Architecture of SCore Cluster System Software

different data sets. Each of these processes runs on its own node. We call the set of processes a parallel process, and call each process an element process. SCore has a gang scheduler which runs multiple parallel processes of the multi-user at the same time.

SCore Runtime System The SCore 2.x runtime system manages parallel processes on top of an unmodified Unix operating system kernel. The runtime system provides an interface for communication between element processes. It hides the behavior of the underlying operating system. Table 3 shows some of the primitives in the SCore 2.x runtime system. It also provides primitives for remote memory access. The SCore 2.x runtime system supports UP clusters running Sun OS 4.1.x, NetBSD and Linux operating systems.

Table 3. SCore runtime system primitives

Primitive	Feature
`_score_get_send_buf`	Get communication buffer for sending
`_score_send_message`	Send message written in the buffer
`_score_peek_network`	Test message reception
`_score_recv_message`	Receive message

PM To extract the performance data of Myrinet, a low level message facility called PM [11, 12, 6] has been implemented. It supports reliable asynchronous message passing on the Myrinet network. PM consists of three parts, code on the Myrinet card, a device driver and a user-level library. By accessing the Myrinet

interface directly from a user program, PM achieves low latency and high bandwidth message passing without the overhead of a system call or an interrupt. PM has the following additional features:

Remote Memory Write
Data is written directly to the receiver's memory by the remote memory write mechanism. Since this feature does not require buffering in main memory, so-called zero copy message transfer is realized. This is one of the reasons that PM achieves high bandwidth. [13, 14].

Multiple Communication Channels
PM provides a channel to support a *virtual network*. Each process of a parallel application uses the same channel number to communicate with another. Multiple processes or threads on a SMP node may exclusively use a channel number or may share a channel number.
The number of channels depends on the NIC's hardware resources. In the current implementation, four channels are supported.

Network Context Switch
Since channels are restricted resources, in order to support multiplexing a mechanism of loading and restoring the network context for each channel is provided. This feature is used by the SCore-D gang scheduler which realizes the multi-processes environment.

MPICH-PM MPICH-PM is an MPI library on top of PM, based on MPICH[8]. Using the PM remote memory write feature, MPICH-PM achieves high bandwidth. MPICH-PM has been developed for UP clusters[10]. MPICH-PM uses the *eager* and the *rendezvous* protocols internally, supported by the MPICH implementation.

The *eager* protocol pushes a message into the network as soon as an MPI send primitive is issued. When a message arrives at the receiver, MPICH stores the message into a temporary buffer if an MPI receive primitive has not been issued. When the receive primitive is issued, MPICH copies the message to the receiver's memory area.

The *rendezvous* protocol can realize zero-copy message transfer between nodes. In this protocol, a control message is sent to the receiver when an MPI send primitive is issued. After that, when an MPI receive primitive is issued on the receiver, the control message is accepted by the receive primitive, and then the receiver replies to the sender with a control message requesting transfer. The sender transfers the message using the remote memory write mechanism when it receives the control message. Since there is no copying of the message in main memory, message transfer using this protocol is called zero-copy message transfer.

In general, the *eager* protocol gives better performance when sending a short message while the *rendezvous* protocol is better when sending a long message. The boundary depends on the host machine. MPICH-PM has the ability to change between the two protocols by specifying the boundary at execution time.

3 SCore 3.x

3.1 Design and Implementation

Since the SCore 2.x runtime system assumes a UP cluster, the system assigns only one element process to each node. The new SCore 3.x system is designed to be capable of assigning multiple element processes to each node.

The SCore 3.x interface for message transfer between element processes within a node is the same as the one for message transfer between SMP nodes. An element process is able to communicate with other element processes transparently by specifying an element process number as the target.

The system recognizes whether the target element process is inside the node or not. It transfers the message using shared memory if the target is inside the node, and it transfers the message using PM if the target is outside.

Communication within an SMP node The runtime system allocates a shared memory across element processes in an SMP node. The shared memory is used as a communication buffer between the element processes. When the runtime system transfers a message, a sender process copies the message to the shared memory then a receiver copies the message from the shared memory. To realize mutual access to the buffer, the current implementation uses the i86's test-and-set instruction[3]. Note that it requires two processor-executed copies as a result.

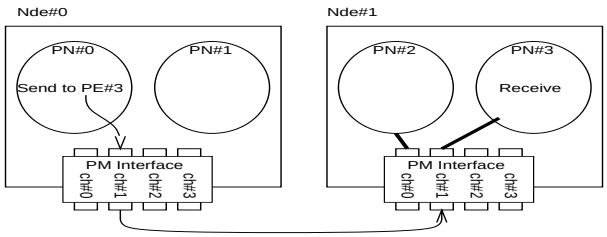

Fig. 3. An example of message transfer within an SMP node

Communication between SMP nodes There are two designs possible to realize communication between an element process and any element process in another SMP node using PM. One is that a single channel is shared by a parallel process. This design has two disadvantages: one is that all processes on a node

[3] Buffers for all the destinations should be implemented so that the runtime transfers messages without mutual access.

must mutually access the channel in both the send and the receive operations. Another is that the receiving process must dispatch an arrival message to the destination process. This is because all the senders send messages using the single channel, and thus, the receiver may receive a message sent to another process in the node.

Another design is that element processes on an SMP node have their own receiving channels. When sending a message to an element process on another node, a message is sent using a channel in which the destination process receives the message. This mechanism requires only shared access to a channel for sending messages. The SCore 3.x runtime is implemented based on this design.

Figure 3 illustrates an example of communication between an element process PN#0 on node 0 and an element process PN#3 on node 1. PN#3 is assigned channel 1. When PN#0 sends a message to PN#3, the SCore 3.x runtime on PN#0 uses channel 1 exclusively.

It is also noted that the implementation uses the i86's test-and-set instruction to realize mutual access to the channels.

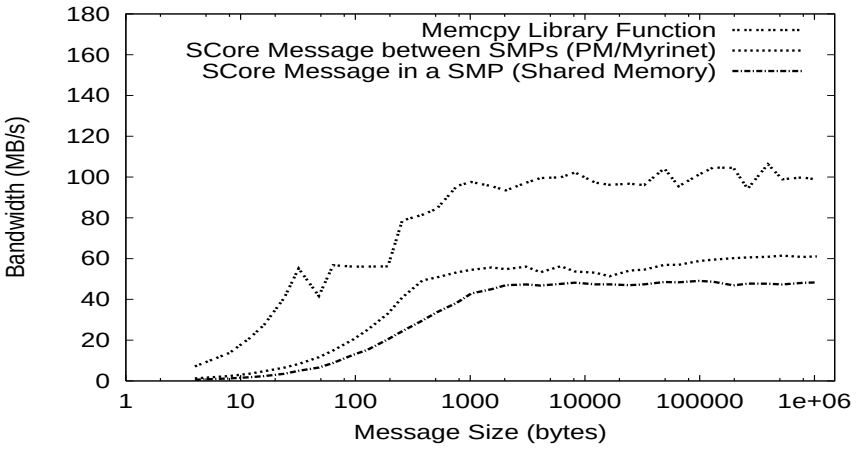

Fig. 4. Basic Performance of SCore 3.x runtime

3.2 Basic Communication Performance

Figure 4 shows the point-to-point bandwidth of the SCore 3.x runtime. For comparison we show bandwidth of the `memcpy` function provided by a Unix kernel in the graph. The bandwidth of communication in an SMP node is half of the bandwidth of memory copy. This is due to the two processor-executed copies.

4 MPICH-PM/CLUMP

4.1 Design and Implementation

Using SCore 3.x it is not difficult to port MPICH-PM to an SMP cluster. However, we could not achieve the same performance as the memory bandwidth using SCore 3.x when communicating within an SMP node because it involves two memory copies.

MPICH-PM realized zero-copy message transfer between nodes by using the PM remote memory write feature. Inside an SMP node, we could not remove both copies for message transfer, but we could design and implement it using only one copy. We extend MPICH-PM to run on an SMP cluster and support a one-copy message transfer within an SMP node. The extended MPICH-PM is called MPICH-PM/CLUMP.

The `_pmVmRead` function is introduced in section 4.1 in order to realize one-copy message transfer. The function copies data directly from the virtual memory of another process using a feature implemented inside the PM device driver without any modification of the Linux kernel. Section 4.3 shows the MPICH *get* protocol that implements one copy message transfer using the `_pmVmRead` function.

Direct Memory Copy between Processes First we show why two copies are required to transfer data between processes in an SMP node. To transfer data between processes, we usually use Unix SYS/V's shared memory or Mach's Memory Object. These require mapping user memory to shared memory. Processes are able to transfer data via the shared memory. We can implement MPI using such a mapping feature. The library has to map the memory used for messages to other processes before the communication so that the sender and the receiver share it. To transfer a message, the sender writes a message to the communication buffer then the receiver reads the message from the buffer. This is the reason why two copies are required using features provided by a Unix kernel.

Since the access cost of main memory is a significant in performance, it is important to reduce the number of copies. To realize one-copy message transfer within a node, we need a feature that makes it possible to copy messages directly from the sender to the receiver without involving the communication buffer. So we design and implement a kernel primitive to read the virtual memory of another process without mapping.

To implement the feature as a kernel primitive, there are two designs:

1. Permit server/client processes to exchange data. First a client negotiates access permission with a server using a well known port or known file name. Then the server permits the client to read or write memory. Only the client process allowed by the server is able to access memory with the access permission. It is dangerous to give write permission to the client, this is the responsibility of the designer of the server process.

2. Permit descendants of a process to exchange data. All Unix processes are ordered by a parent and child relationship. Use this order to inherit permission. Some type of framework is required to specify permission.

Since the reason why we need a memory access facility between processes is the realization of one copy message transfer, we implemented the following simple features based on the latter design.

- Permit descendants of a parent process to mutually read data. The exclusive `/dev/pmvm` device open by the parent process realizes this. The device implements the direct memory copy operation as an `ioctl` system call.
- The library function `_pmVmRead`, realized using the `ioctl` system call, copies data from the memory of the process specified by arguments to the memory of the current process directly. The following are the declarations of `_pmVmRead`:

```
int _pmVmRead(src_pid, src_vaddr, dst_vaddr, size)
    int src_pid;
    caddr_t src_vaddr, dst_vaddr;
    size_t size;
```

Implementation of the direct memory copy uses the following three features of the Linux kernel:

1. Looking up the page table for a specified process
2. Paging-in the physical page allocated by other processes
3. Reading the physical page allocated by other processes

In other words, the function can be implemented on all UNIX kernels that support the three features.

4.2 Basic Performance of Direct Memory Copy

Figure 5 shows the bandwidth of `_pmVmRead`. For comparison, we show the bandwidth of the memcpy function. The result of `_pmVmRead` shows that the performance is lower than memcpy when sending short messages while the performance is the same as memcpy when sending messages longer than 3 KBytes. The reason is the overhead of the `ioctl` system call. The `_pmVmRead` function calls the `ioctl` to kick the device driver. The call takes 5 usec every time. The overhead is caused by context switches between user level and kernel level.

4.3 The get protocol

We have explained the *eager* and the *rendezvous* protocols used in MPICH in section 2.2. MPICH also defines a *get* protocol. We implemented one-copy message transfer using the *get* protocol with the `_pmVmRead` function. Figure 6 illustrates one-copy message transfer.

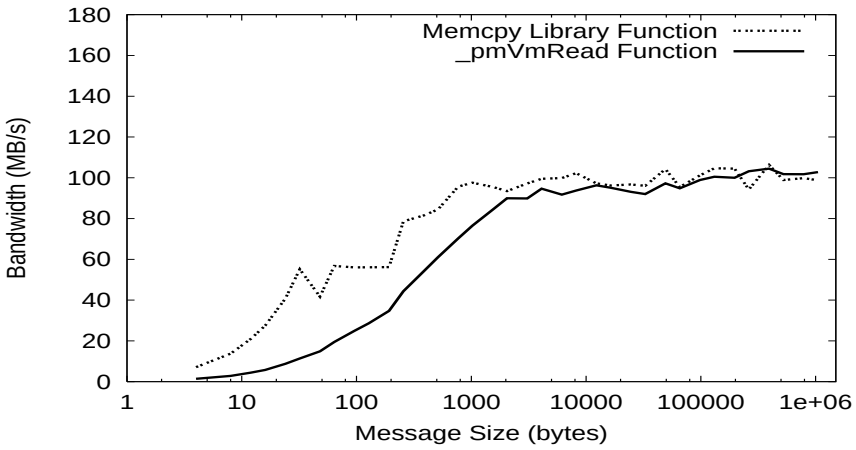

Fig. 5. Bandwidth of _pmVmRead

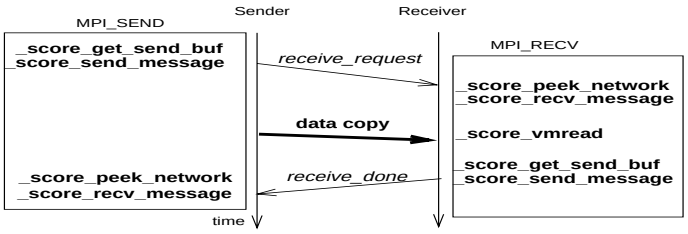

Fig. 6. The get protocol

1. When an MPI send primitive is invoked on the sender, the sender sends a receive_request control message to the receiver.
2. The sender waits for a receive_done control message from the receiver.
3. The receiver receives the receive_request control message and when the MPI receive primitive is invoked, it copies the message from the sender process using the `_pmVmRead` function.
4. After completing the copy, the receiver sends the receive_done control message to the sender.

5 Evaluation

5.1 Point-to-Point Performance

Table 4 and figure 7 show the point-to-point performance of MPICH-PM/CLUMP measured on LBP. The latency is half of the average round trip time of ping-pong

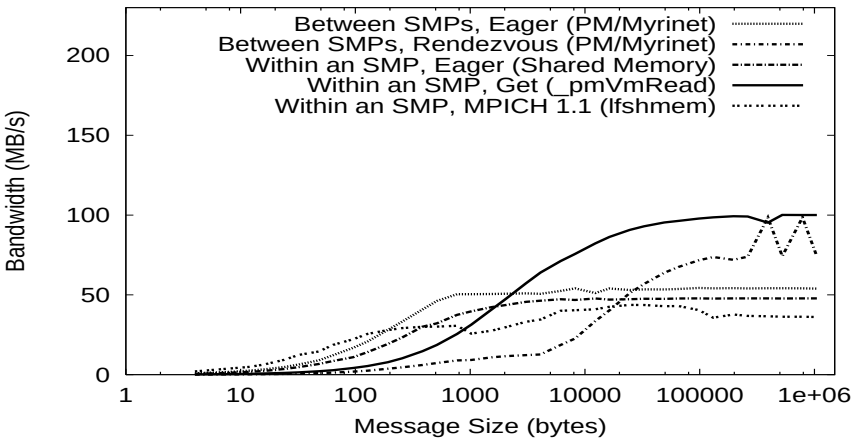

Fig. 7. Point-to-Point Bandwidth of MPICH-PM/CLUMP

Table 4. Point-to-Point Performance of MPICH-PM/CLUMP on the ACL LBP Cluster

	Within an SMP		Between SMPs	
	eager	*get*	*eager*	*rendezvous*
Minimum Latency (usec)	8.62	9.98	11.08	12.75
Maximum Bandwidth (MBytes/s)	48.03	100.02	54.92	97.73
Half Bandwidth Point (KBytes)	0.3	2.5	0.2	21

messages. The bandwidth is the amount of messages transfered. The half bandwidth point is the message length that achieves half of maximum bandwidth. To obtain both results, benchmark programs run for more than one second.

For comparison the graph shows the bandwidth of the `lfshmem` device that is included in MPICH 1.1 distribution. The `lfshmem` device has the best performance among the devices included in the distribution. It is noted that the device doesn't have the code to realize communication between SMP nodes.

The crossing point of the *rendezvous* protocol with the *eager* protocol depends on the configuration of the cluster system. On LBP the point is about 30 KBytes. It is natural to attempt to switch protocols depending on the size of messages. However the analysis is not easy, as we should consider the characteristics of each protocol. The memory access load of the *rendezvous* protocol is lighter than that of the *eager* protocol. On the other hand, the *rendezvous* protocol requires additional control messages to synchronize sender and receiver. It is noted that when sending the results of a calculation immediately, the cache system reduces the overhead of memory copies, which will improve the performance of the *eager*

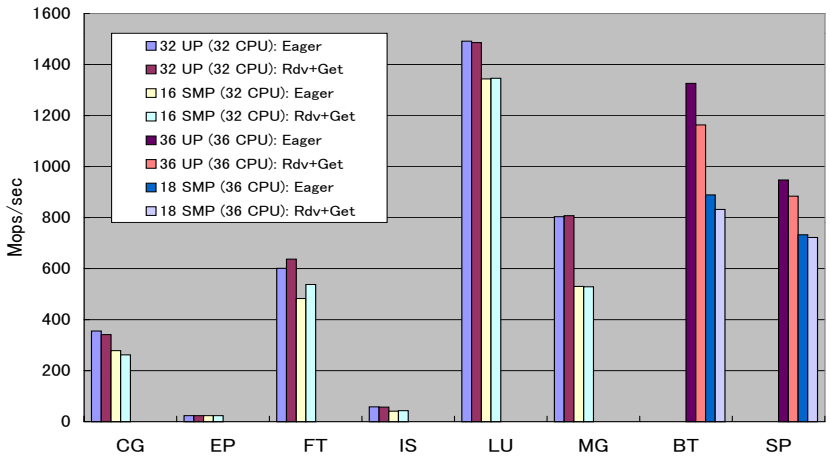

Fig. 8. NPB: Results on 32 CPU or 36 CPU out of LBP

protocol.

The *get* protocol of MPICH-PM/CLUMP is the best of the protocols within an SMP node when sending messages larger than 4 KBytes. The *get* protocol achieves 100 MBytes/s of bandwidth when sending 128 KBytes while the maximum bandwidth of the *eager* protocol within an SMP node is 48 MBytes/s. The performance is twice that of the `lfshmem` device. The *get* protocol realizes one-copy transfer of messages. The result shows the protocol achieves its designed performance in practice.

5.2 NAS Parallel Benchmarks

Figure 8 and Figure 9 show the results of NAS Parallel Benchmark 2.3 Class B[5] on LBP. The first graph shows the performance results printed by each NAS benchmark program. The combination of the *rendezvous* protocol and the *get* protocol is slower than the *eager* protocol alone except when running FT and IS. The following reasons are considered.

- Contention of messages reduces the performance of the *rendezvous* protocol drastically.
- When sending calculated results immediately, the cache system reduces the copy overhead of the *eager* protocol.

The second graph compares the results of an SMP cluster and a UP cluster with same number of CPUs. Each bar shows the ratio of the SMP cluster to the UP cluster. The SMP cluster achieves 70% to 100% of the performance of the UP cluster. The combination of the *rendezvous* protocol and the *get* protocol shows good performance compared to that of the *eager* protocol.

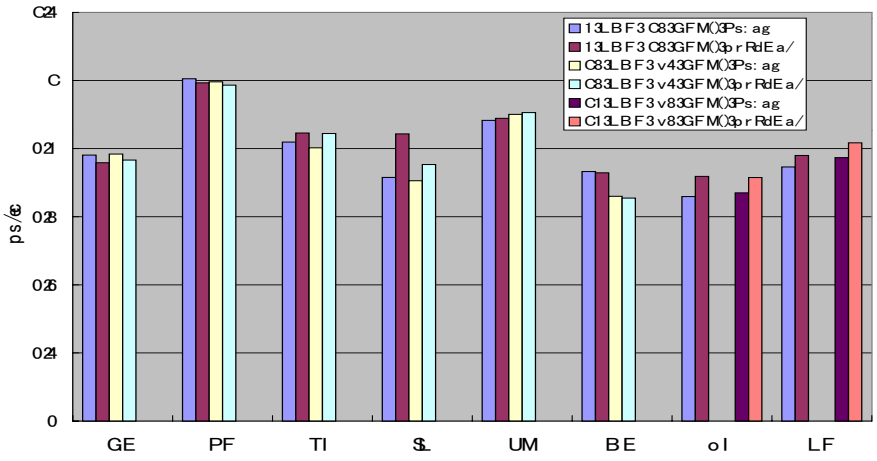

Fig. 9. NPB: Comparison with UP cluster

The following reasons are considered: When using the combination, the *rendezvous* protocol and *get* protocol each handle half of the messages on the SMP cluster. On the other hand, the *rendezvous* protocol handles all the messages on the UP cluster. The SMP and UP clusters each pass the same number of messages to the network. But the number of control messages for the *rendezvous* protocol is half that on UP cluster. Since it is considered that the latency of control messages is significant to the performance when the network is crowded, the number of control messages affects performance.

Intuitively, an SMP cluster is faster than a UP cluster if both clusters have the same number of CPUs because communication within an SMP node is faster than communication across nodes. Using the third programming model, however, some applications that frequently synchronize among processes do not run faster than a UP cluster with the same number of CPUs.

The following reasons are considered.

- Since the synchronization mechanism among processes is realized by message passing, the synchronization cost is dominated by the communication cost. Though communication is faster within a node, the synchronization cost is dominated by communication cost among nodes.
- Total amount of messages of the execution on an SMP cluster is equal to the one on a UP cluster.

Hence, the performance of such applications on the SMP cluster does not exceed the performance of the UP cluster. The NAS Parallel Benchmarks 2.3 are examples of such applications.

6 Related Work

The Active Messages group at the University of California, Berkeley[1], the Fast Messages group at the University of Illinois, Urabana-Champaign[3] and the BIP group at University Claude Bernard Lyon 1[2] are also developing MPI libraries over Myrinet. As far as we know, there is no report describing the results of NAS Parallel benchmarks using these MPI on recent SMP clusters.

The implementation of MPICH 1.1 uses shared memory for communication within an SMP node. It makes two copies to send/receive a message. These copies lower the bandwidth.

From the point of view of SMP implementation of MPI, TOMPI(Threads Only MPI)[7] realized an MPI implementation that copies a message only once when sending the message. TOMPI rewrites an MPI application using a source code translator to run it using multiple threads on an SMP node.

The Linux `proc` filesystem supports a feature that makes it possible to access the memory of other process. Basically the feature is the same as our kernel primitive `_pmVmRead`. However the current implementation of the `proc` filesystem limits the access area to pages resident in main memory. This is one reason why we did not choose it for the MPICH-PM/CLUMP implementation.

From the point of view of the design of a kernel primitive, the Linux `pread` primitive is able to read any region of a file. If a user process can access another process via a file, the primitive provides the same feature as `_pmVmRead`.

7 Conclusion

In this paper, we designed and implemented MPICH-PM/CLUMP, an MPI library for SMP clusters based on MPICH 1.0. MPICH-PM/CLUMP supports multiple processes on an SMP node. MPICH-PM/CLUMP realizes zero-copy transfer of messages between SMP nodes, and one-copy transfer of messages within an SMP node. In order to implement MPICH-PM/CLUMP, we first designed and implemented the SCore 3.x runtime. The runtime realizes process management on an SMP cluster. The runtime provides communication primitives to upper level libraries. The primitives realize transparent message transfer between the processes. Secondly, we designed a direct copy facility to realize one-copy message transfer on an SMP node. The facility is implemented inside the PM device driver. Then we used it to implement the *get* protocol in MPICH-PM.

We measured the performance of MPICH-PM/CLUMP using the ACL "Little Blue Penguin" Cluster at Los Alamos National Laboratory. The cluster consists of 64 Dual Pentium II 333MHz machines and a Myrinet network. MPICH-PM/CLUMP achieved 8.62 usec of latency and 100 MBytes/sec of bandwidth within an SMP node. The result is twice the bandwidth of the `lfshmem` device of the MPICH 1.1 implementation. MPICH-PM/CLUMP achieved 11.08 usec of latency and 98 MBytes/sec of bandwidth between SMP nodes. The resulting bandwidth achieved is close to the node's memory system limit.

We evaluated MPICH-PM/CLUMP using the NAS Parallel Benchmarks 2.3 class B. The ratio of the result on the SMP cluster to the result on a UP cluster that has same number of CPU's is 70% to 100%. As discussed in section 5.2, this is considered to be due to the fact that the network message-based synchronization cost on an SMP cluster is not much lower than that of an equivalent UP cluster, and total message traffic is equal to that of a UP cluster. We need further work to investigate and measure the causes of the disadvantage.

References

1. Message Passing Interface Implementation on Active Messages. http://now.CS.Berkeley.EDU/Fastcomm/MPI/.
2. MPI-BIP An implementation of MPI over Myrinet. http://lhpca.univ-lyon1.fr/mpibip.html.
3. MPI-FM: Message Passing Interface on Fast Messages. http://www-csag.cs.uiuc.edu/projects/comm/mpi-fm.html.
4. Myrinet. http://www.myri.com.
5. NAS Parallel Benchmarks.
6. PM: High-Performance Communication Library. http://www.rwcp.or.jp/lab/pdslab/pm/home.html.
7. Erik D. Demaine. A Threads-Only MPI Implementation for the Development of Parallel Programs. In *11th International Symposium on High Performance Computing Systems (HPCS'97)*, pp. 153–163, July 1997.
8. William Gropp and Ewing Lusk. MPICH Working Note: Creating a new MPICH device using the Channel interface. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.
9. Message-Passing Interface Forum. MPI: A message passing interface standard, version 1.1, June 1995.
10. Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. In *ICS'98*, July 1998.
11. Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. Design and Implementation of PM: a Communication Library for Workstation Cluster. In *JSPP'96 (in Japanese)*. IPSJ, June 1996. (in Japanese).
12. Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In *High-Performance Computing and Networking '97*, 1997.
13. Hiroshi Tezuka, Atsushi Hori, Francis O'Carroll, Hiroshi Harada, and Yutaka Ishikawa. A User level Zero-copy Communication using Pin-down cache. In *IPSJ SIG Notes*. IPSJ, August 1997. (in Japanese).
14. Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *IPPS'98*, April 1998.

This article was processed using the LaTeX macro package with LLNCS style