Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

Design and Implementation of Message Passing Interface for Clusters of Workstations on Local Area Networks

Yuan-Zi Chang Kuo-Shun Ding Jyh-Jong Tsay
Department of Computer Science and Information Engineering
National Chung Cheng University
MinHsiung, Chiayi 621, Taiwan, ROC

Abstract

Message passing is a paradigm widely used for writing parallel programs on clusters of workstations that have become an attractive and efficient alternative for parallel computing. Message Passing Interface (MPI) is a standard proposed for writing portable message-passing parallel programs. In this paper, we present a design and implementation of MPI that is optimized for LAN clusters of workstations. We develop a daemon-based implementation that consists of two main componenets: a MPI library that provides the functionality of MPI, and a multicast daemon that handles all message passing among MPI processes and provides multicast operations. We propose a user-level reliable multicast protocal, and implement it with sliding window techniques and hardware broadcast to improve performance. Performance measurement shows that our implementation outperforms two of the major implementations, MPICH and LAM, that are available from public domain. Furthermore, our implementation allows uncoordinated checkpointing that is simple and efficient.

1 Introduction

Clusters of workstations and personal computers running on Local Area Networks (LANs) have become an attractive and effective alternative for parallel computing as high performance and low cost workstations and networks are widely available. It uses softwares to integrate existing workstations and networks, and does not require purchase of expensive multiprocessor systems. Programmers can use the same editors, compilers, debuggers, and operating systems that are available on individual workstations. This substantially reduces development and debugging time.

Message passing is a widely used paradigm for writing parallel programs on parallel computers, especially scalable parallel computers with distributed memory such as clusters of workstations. A message-passing program consists of a set of autonomous processes running on each processor node. Processes communicate with each other by sending/receiving messages. Recently, a Message Passing Interface (MPI) [14] has been proposed for writing portable and efficient message-passing parallel programs. MPI supports process groups, and provides operations for point-to-point communications, collective communications, process topologies and profiling.

In this paper, we present a design and implementation of MPI that is optimized for LAN clusters of workstations and personal computers. One of our major effort is to optimize the performance of communication operations on LAN clusters of workstations. Notice that current LANs such as Ethernet and ATM provide efficient broadcast, but do not guarantee reliable message delivery. We propose and implement a user-level reliable multicast protocal that can be implemented on top of unreliable networks, and makes use of the broadcast capability of current LANs. Our implementation is daemon-based, and consists of two major componenets: a MPI library, and a multicast daomen. The multicast daemon provides reliable muulticast among application processes. Performance mesaurement shows that our implementation outperforms two of the major implementations, MPICH [1] and LAM [3], available from public domain. Furthermore, our implementation allows uncoordinated checkpointing that is simple and efficient.

The following summarizes the main features of Our implementation.

- We design and implement a daemon-based system that consists of two major components: the MPI library that provides the functionality of MPI, and a multicast daemon that provides efficient and reliable multicast operations among all MPI application processes. Notice that it is possible to use our multicast daemon as a basic component to implement other message passing system such as PVM (Parallel Virtual Machine).

- We propose a reliable multicast protocol that can be implemented on top of unreliable communication networks. We use the protocol to implement our multicast daemon on Ethernet LAN. The protocal is based on selective repeat protocol[7, 18]. It is observed that most of packet loss on LANs is due to buffer overflow. We modify sliding window protocol to control packet flow to reduce the possibility of packet loss. To further imporve the performance, we use hardware broadcast to implement some of collective communications on Ethernet.

- We use shared memory for communication between MPI library and multicast daemon, and develop simple techniques to synchronize their interactions. The shared memory reduces the communication overhead, and makes our MPI library easily ported to shared memory multiprocessors.

- In the MPI library, we optimize our implementation for point-to-point comunications as well as for collective communications. For point-to-point communication, we use packetization to overlap protocal processing and network transmission within the sending node, as well as message processing at sending and receiving nodes. For collective communication, we use hardware broadcast and nonblocking receive to improve the performance.

- Based on our system architecture, we have propose an approach for ckeckpointing our MPI programs [13] in an uncoordinated fashion. Since the multicast daemon handles all message passing and gurarntees reliable transmission, checkpointing on each node can be taken without any internode coordination.

Notice that there are several implementations of MPI on LAN clusters of workstations. Local Area Multicomputer(LAM) [3] is implemented by Ohio Supercomputer Center. It is a daemon-based implementation. Each processor node runs a daemon to synchronize all messages. It adopts Stop-and-Wait to provide reliability on top of unreliable UDP (User Datagram Protocol). MPICH [1] is implemented by Argonne National Laboratory. MPICH does not run a daemon on each processor node. All MPI communication functions are implemented on top of reliable TCP (Transmission Control Protocol). Both LAM and MPICH does not use any techniques to control message loss, and does not use hardware broadcast to improve the performance of collective communication.

Recently, J. Bruck *et.al.* [2] at IBM Almaden implement the communication subset of MPI on LAN clusters of wortkstations. Their implementation is the one closest to our implementation in the sense that both implementations make use of hardware broadcast for collective communication, and sliding window techniques to control message flow. Nonetheless, there are several differences. Their implementation modifies the kernel to improve the performance, and works only for programs that requires at most k packet buffers for message passing. Our implementation is daemon-based, is fully user-level, supports general MPI programs, and allows simple uncoordinated checkpointing.

2 System Architecture

In this section, we describe our implementation of MPI on cluster of workstations running on LANs. As in Figure 1, each workstation must run a multicast daemon proccess, and can run more than one MPI application proccesses. Application processes are connected to the multicast daemon, and communicate via multicast daemons. The multicast daemon handles all the message passing among application processes, and provides reliable multicast operations. The MPI library is linked as a part of the application process.

MPI application processes communicate with the daemon process via shared memory. Each application process has a send queue and a receive queue in the shared memory. To send a packet, an application process writes the packet to its send queue, and triggers the daemon to send out that packet. The trigger message is sent via a FIFO channel. The daemon always listens for messages from communication channels. When a trigger message arrives, the daemon will examine each send queue, and send out all the packets ready in the send queue. When a packet arrives, the daemon will receive the packet, and dispatches the packet to the receive queues of its destination processes.

3 Reliable Multicast Protocol

We propose a user-level multicast protocal to implement the multicast daemon. The protocol guarantees reliable delivery, and FIFO order [12] between

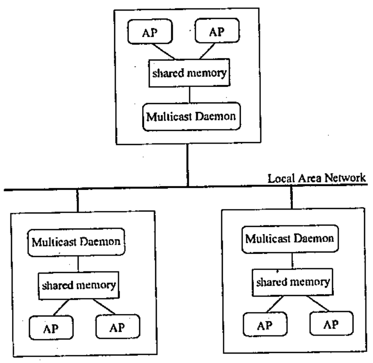

Figure 1: System Architecture

every source and destination pair. We assume that packet loss is possible; however, the content of a received packet is not corrupted beyond the tolerance of standard error correction. The protocol is based on selective repeat protocol in which a lost packet is resent until it is received. A packet is assumed to be lost when its acknowledgement is not returned after a certain period of time or when packets are not received in their FIFO order. Each packet consists of the following:

- a packet type that is used to distinguish normal packets from control packets such as acknowledgements and resend requests,
- the source of the packet that specifies the node ID of the multicast daemon sending the packet,
- the number of destinations that specifies the number of nodes to receive the packet,
- the content packet, and
- the target list that is an array of destination node IDs and their corresponding sequence numbers.

We maintain a sequence number between every source and destination pair, which guarantees FIFO delivery. Each sending packet is associated with an *ack_outstanding list*[7] that keeps the list of receivers whose acknowledgements are not yet received. The daemon transmits packets continuously and constantly listens for acknowledgements of previous transmissions. Immediately upon transmission of a packet, the daemon sets a time-out counter and initializes the ack_outstanding list which is simply the list of all destination daemon. The ack_outstanding lists of previously transmitted packets are constantly updated as new error free acknowledgements are received. If the ack_outstanding list of a particular packet becomes empty, the packet is deleted. Otherwise, after some fixed interval, the counter expires and the daemon retransmits it. The ack_outstanding list is implemented as a bit vector. In order to recude the acknowledgement overhead, acknowledgement is not sent for every packet. It is sent after every 25 packets in the current implementation.

Packet loss is detected by a receiver when packets are not received in FIFO order. As soon as packet loss is detected, the receiver sends a request to the sender to resend the lost packet.

The maximun packet size is about 1.3K bytes. Packetization is handled by application processes. Messages of size larger than 1.3K bytes are partitioned into packets that are written into consective entries in the send queue. It is observed that packet loss is the main source of unreliable delivery on LAN. Furthermore, most of the packet loss is due to system buffer overflow. We use sliding window techniques to control the flow and reduce the possibility of buffer overflow. From experiment, it is observed that when the system buffer is 32768 bytes, and the packet size is 1024 bytes, the best performance is achieved when the window size is 30. We choose 25 as window size in our implementation.

4 Sending/Receiving Packets

Each multicast daemon has a send buffer and a receive buffer that are shared by all application processes. The send buffer and receive buffer are divided into hundreds of entries. Each entry is a packet buffer that can hold a packet of size upto about 1.3K bytes. Each application process has a send queue and a receive queue. Each entry of the send/receive queue stores an index of a packet buffer in the shared send/receive buffer. When a packet is written to an entry of a send queue, the daemon will first replace that entry with a free packet buffer, send out the packet, and then move the packet to the *transmitting list* that keeps all the packets under transmitting. When the ack_outstanding list of a sending packet becomes empty, i.e. the acknowledgement from every receiving process has been received, the daemon will then delete that packet from the transmitting list. Notice that packets from the same send queue will be transmitted in their FIFO order; however, packets from different send queues can be mixed arbitrarily.

When a packet arrives, the daemon finds a free packet buffer from the receive buffer, reads the packet into the packet buffer, and then dispatches the packet to its receiving processes by writing the index of the packet buffer to thir receive queues. Each received packet is associated with a *read_outstanding list.* After the read_outstanding list becomes empty, i.e. all receiving processes have read this packet, the daemon free the packet buffer and returns it to the free list of receive buffer.

An acknowledgement contains the source node ID and the sequence numbers of lost packets. There are several situations for the daemon to send an acknowledgement. When packets from the same source node are not received in contiguous order, the daemon assumes that packets not received in order are lost, and send an acknowledgement to request the sender to resend the lost packets. When the daemon receives a packet that is previously received or a packet with sequence number not within the receiving window range, it implies that it is highly possible that previous acknowledgemt is lost. The daemon sends an acknowledgement that records the current receiving status. When everything is normal, the daemon sends an acknowledgement after receving every N packets. In current implementation, N is 25.

5 Point-to-Point Communicaiton

The basic communication mechanism of MPI is point-to-point communication that transmits messages between a pair of process. Send and receive are the basic functions of point-to-point communication. MPI provides blocking and nonblocking send and receive functions. The blocking send call blocks until the send buffer can be reclaimed. Similarly, a blocking receive call blocks until the receive buffer actually contains the contents of the message.

In our implementation, a MPI message consists of the following information:

- the number of destinations that specifies the number of processes to receive the message,
- the destination list that specifies the ranks of destination processes in the MPI_COMM_WORLD communicator,
- the rank of the source process in the MPI_COMM_WORLD communicator,
- the message tag that allows selectivity of messages at the receiving end,
- the context that specifies the world of the communication,
- the mode of communication, and
- the content of the message.

A send function appends the message header, partitions the message into packets, writes the packets into consecutive entries of the send queue, and triggers the daemon to transmit the packets. The first packet is marked as MPI_HEAD, and the last packet is marked as MPI_TAIL. Message header is stored in the first packet only. Packets from the same message will be transmitted continuously by the daemon. The receive function extracts packets from receive buffer, and assemble packets into messages. We next explain our implementation of blocking send and receive.

5.1 Blocking Communication

MPI provides 4 modes of send functions: *standard, buffered, synchronous* and *ready.* In our implementation, the standard send, MPI_Send blocks until the entire message is copied to the send buffer. The buffered send, MPI_Bsend, is the same as MPI_Send except that it returns an error if the send buffer is out of space. The synchronous send, MPI_Ssend, is the same as standard send except that MPI_Ssend waits an MPI acknowledgement from the receiving process. The ready send is implemented as the standard send.

MPI library maintains a *local buffer* that keeps all the packets read from the receive queue but not yet matched by any receive operation. A receive operation, MPI_Recv, will searches over all arrived messages. If a matched message is found, it returns that message; otherwise, it blocks until a matched message arrives. Arrived messages can be in either local buffer or receive queue. A receive will search the locall buffer first, and then the receive queue. We assume packets are extracted from the receive queue in their FIFO order. Arrived messages are copied to local buffer when the following situations occur. First, when packets are not received by receive operations in their FIFO order, the out-of-order packets are moved to local buffer. Secondly, when receive queue is full, the daemon will ask MPI library to move packets from receive queue to local buffer so that it can proceed to receive incoming packets. Notice that when all the packets are received in FIFO order, and the receive queue does not overflow, no packet will be copied to local buffer. In such situtaion, our implementation achieves the best performance.

5.2 Nonblocking Communication

Similar to blocking send, nonblocking send also has 4 modes: *standard, buffered, synchronous* and *ready.* In our implementation, nonblocking sends are implemented in a fashion similar to blocking sends except that in the synchronous mode, instead of blocking until acknowledgement returns, it generates a MPI request for the acknowledgement.

Nonblocking receive operation, MPI_Irecv, does not read any message unless the message has already arrived. When it can not find a mactched message, it generates MPI request, and inserts the request to a request list. When an arriving message mathces the request, it will be copied to the local memory specified by the request.

A MPI_Wait can be called to wait until the corresponding request is satisfied. A MPI_Test can be called to test if a request has been satisfied.

Notice that nonblocking receive is very useful when a process needs to receive messages from multiple source nodes, and the order of message arrival is nondeterministic. Such situation occurs in some collective communication such as MPI_Gather, MPI_Reduce, MPI_Alltoall, and all-node gather and reduce.

5.3 Performance of Point-to-Point Communication

The experiment is run on a cluster of Sun workstations connected by a 10M Ethernet and running SunOS 4.1.3.

Three simple programs have been used to evaluate the basic performance of point-to-point communication. Each program involves two processes each running on one workstation. The first one is called Ping in which one process keep sending messages to the other process. The second one is called PingPong in which a message bounces back and forth between the two processes. The time measured in PingPong approximates the latency of a message passing. The third one is called Exchange in which two processes exchange data.

For Ping, experiment shows that our system is always better than MPICH. LAM is slightly better than our system when message size is between 6000 bytes and 8000 bytes. This is because the message of size less than 8192 bytes is sent in only one message by LAM. However, LAM is much worse than our system for messages of other sizes.

For PingPong and Exchange, our system is always better than LAM and MPICH. Figure 2 and figure 3 shows the performance of PingPong on 2 nodes. Note that the time measured in PingPong approximates the latency of a message passing. Figure 4 gives the bandwidth estimated from the running times of PingPong. It shows that our system achieves better bandwidth for long messages. This is because the longe messages allow more overlap of protocol processing and network transmition, as well as concurrent processing at sending and receiving ends. Our system achieves a bandwidth of about 6.4 Mbit/sec when the message size is about 8,000 bytes, and a bandwidth of about 7Mbit/sec when the message size is about 16,000 bytes.

6 Collective Communication

Our implementation of collective communications makes use of hardware broadcast and nonblocking receive to improve the performance. Nonblocking receive improves the performance when a process needs to receive messages from muitlple processes, and the order of message arrival can be nondeterministic. Due to length limitation, we give performance data with brief description for MPI_Bcast, MPI_Gather, MPI_Scatter, and MPI_Reduce. For other operations, we give performance data without description. Further details can be found in [4]. All the performance data are measured on 8 workstations connected in the same segment of Ethernet.

6.1 MPI_Bcast

One MPI process is run on each of the 8 workstations. All processes first run a barrier to synchronize. Process with rank 0 then broadcasts a message to all other processes by MPI_Bcast. Above processes is repeated 50 times. The maximum time taken over all the non-root processes gives an estimate of the maximum time taken by any process participating in the broadcast. In Ethernet, broadcast can be performed by either hardware broadcast or point-to-point communication. Figure 5 shows that our implementation outperforms LAM and MPICH with either hardware broadcast or point-to-point communication. When all the workstations are connected in the same segment, hardware broadcast achieves better performance. In next following subsections, we assume MPI_Bcast is implemented with hardware broadcast.

6.2 MPI_Gather

Each process sends the message to the root process. The root process uses nonblocking receive, MPI_Irecv, in rank order. It then uses uses MPI_Waitall to complete all receive operations. Figure 6 shows the results. The byte count refers to the size of the data sent by each process to the root.

6.3 MPI_Scatter

We call MPI_Bcast to broadcast the whole message to each process. Each process receives the message and extracts the part it needes. Using broadcast reduces the sending time in the sending process, especially for short messages that are smaller than a packet. For long messages that are larger than a packet, we simply use unicast to send each part of the message to each process. Figure 7 shows the results. The maximum time taken over all the non-root processes gives an estimation of the maximum time taken by any process participating in the broadcast. This maximum time is what is shown in the graph plotted against the byte count which refers to the size of data sent by the root to each process.

6.4 MPI_Reduce

We first call MPI_Gather to gather the array to the root process. The root process then performs the reduce operation. Figure 8 shows the results. The time reported is the running time of the process with rank 0, and the integer count refers to the size of the data send by each process to root process.

7 Bitonic Sort and Matrix Multiplication

To measure overall performance of our system, we also run two programs *Bitonic Merge Sort* and *Cannon's Matrix Multiplication* on 8 and 9, respectively, workstations. Bitonic Merge Sort [19] runs in phases. In each phase, processes exchange data to form larger sorted list. We take the longest time of the 8 processes as the running time. Figure 9 shows that our system achieves better performance for various data size. In addition, our system has the smallest variance of the running times of the 8 processes.

In Cannon's algorithm [19] for matrix multiplication, two $n \times n$ square matrices A and B is partitioned into p square submatrices. We label the processors from $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$, and initially assign $A_{i,j}$ and $B_{i,j}$ to $P_{i,j}$.

The first communication step of the algorithm aligns the blocks of A and B in such a way that each processor multiplies its local submatrices. This alignment is achieved for matrix A by shifting all submatrics $A_{i,j}$ to the left by i steps and all submatrices $B_{i,j}$ are shifted up. After a submatrix multiplication step, each block of A move one step left and each block of B moves one step up. A sequence of $\sqrt{p}$ such submatrix multiplications and single-step shifts pairs up each $A_{i,k}$ and $B_{k,j}$ for $k(0 \leq k \leq \sqrt{p})$ at $P_{i,j}$. This completes the matrix multiplication of matrices A and B. The communication activity in each submatrix multiplication is two message passing. We run Cannon's algorithm on 9 workstations. Figure 10 gives the maximum time of 9 processes to multiply matrices of various size. It shows that our system achieves the best performance.

8 Conclusion and Future Work

In this paper, we have presented a daemon-based implementation of MPI that achieves better performance than MPICH and LAM on clusters of workstations connected by Ethernet LANs. We believe our performance improvement is a combined effect of the following:

- the reduction of packet loss that is a combined results of the daemon-based approach and the use of silding window techniques,
- the overlap of protocal processing and message transmitting, and the concurrent execution of sending and receiving processes, which are due to packetization, and
- the use of hardware broadcast and nonblocking receive in collective communications.

In the future, we will proceed to measure the performance of our implementation on other LANs such as FDDI and ATM.

References

[1] P. Bridges, A. Bruce, J. Mills, and A. Simith. User's Guide of MPICH, A Portable Implementation of MPI, 1994.

[2] J. Bruck, D. Dolev, C.-T. Ho, M.-C. Rosu, and R. Strong. Efficient Message Passing Interface(MPI) for Parallel Computing on Clusters of Workstations, *ACM 1995 Annual Symposium on Parallel Algorithms and Architectures*.

[3] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI, Ohio Supercomputer Center, 1994.

[4] Y.-Z. Chang, K.-S. Ding, and J.-J. Tsay. Design and Implementation of Message Passing Interface for Clusters of Workstations on Local Area Networks, Technical Report, Department of Computer Science and Information Engineering, National Chung Cheng University, 1996.

[5] K. Ravindran and S. Samdarshi. A Flexible Casual Broadcast Communication Interface for Dirtributed Applications. *Journal OF Parallel AND Distributed Computing*,16,134-157,1992.

[6] R. Subramonian and N. Venkatasubramanyan. Optimal broadcast in a distributed memory model of parallel computation.

[7] I. S. Gopal and J. M. Jaffe. Point-to-Multipoint Communication Over Broadcast Links. *IEEE Trans. on Communications*, Vol COM-32, No.9, Sept. 1984.

[8] R.H. Dong, X.Y. Zhang, and Y.K. Tham. New point-to-multipoint communication protocols. *IEEE Proceedings I*, Vol:136, pages 312-316, Aug 1989.

[9] P.M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, Vol:1, pages 17-25.

[10] Y.-I. Chang and M.-H. Hwang. A Counter-Based Reliable Broadcast Protocol. *EUROMICRO 94. System Architecture and Integration, pages 396-403.*

[11] I. Gopal,and R. Rom. Multicasting to Multiple Groups Over Broadcast Channels. *IEEE Trans. on Communications*, Vol. 42, No. 7 , July 1994."

[12] C. Kim,and J. Y. Lee. Message Ordering in Multicast Communication. 1993 Global Data Networking, pages 186-190.

[13] W.-J. Li and J.-J. Tsay. Checkpointing Message Passing Interface Programs. Technical Report, Department of Computer Science and Information Engineering, National Chung Cheng University, 1996.

[14] Message Passing Interface Forum: MPI: A message-passing interface standard, version 1.0, May 1994.

[15] G.A. Geist and V.S. Sunderam. Network-based concurrent computing on the PVM system. *In Concurrency: Practice and Experience*, pages 293-311, June 1992."

[16] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrence:Practice and Experience*, 2(4):315-339, Dec. 1990.

[17] A. Geist, A. Beguilin, J.J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. PVM3 users's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National laboratory, May 1993.

[18] A. S. TanenBaum. Computer Networks, pages 223-239.

[19] V. Kumar, A. Grama, A. Gupta, G. Karypis. Introduction to Parallel Computing, Benjamin/Cummings, 1994.

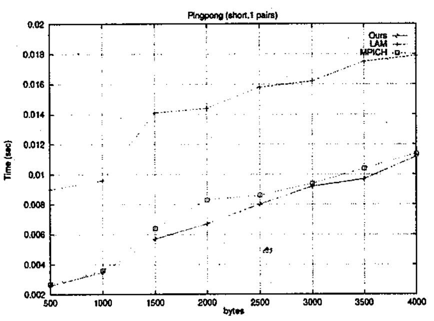

Figure 2: PingPong with short messages

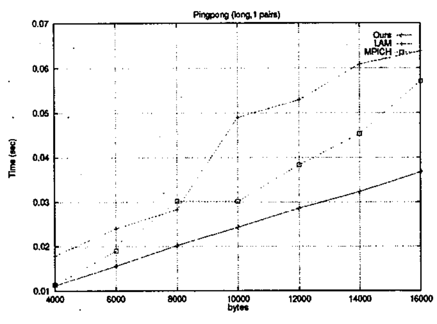

Figure 3: PingPong with long messages

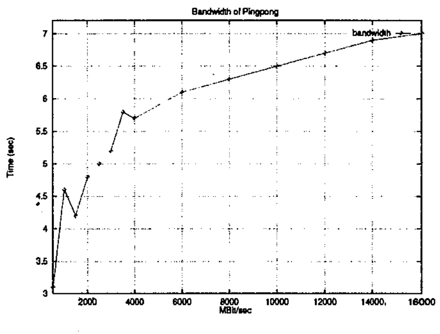

Figure 4: Bandwidth of PingPong

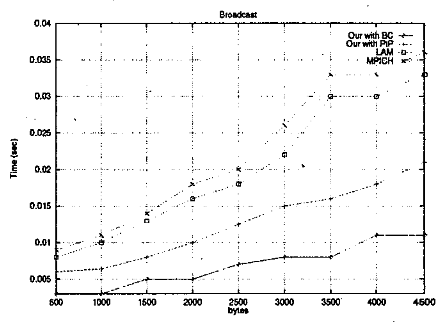

Figure 5: MPI_Bcast

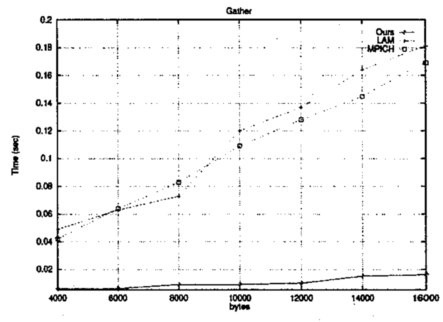

Figure 6: MPI_Gather

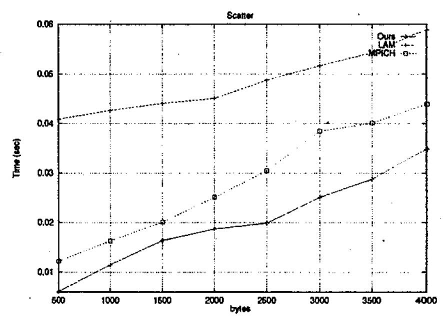

Figure 7: MPI_Scatter

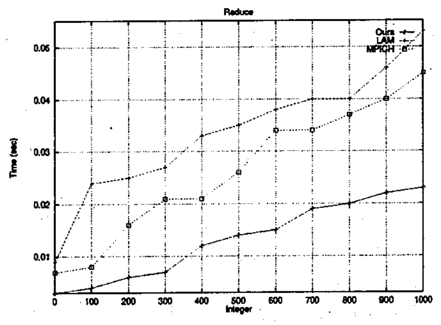

Figure 8: MPI_Reduce

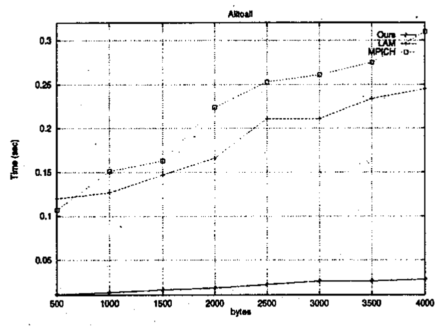

Figure 12: MPI_Alltoall

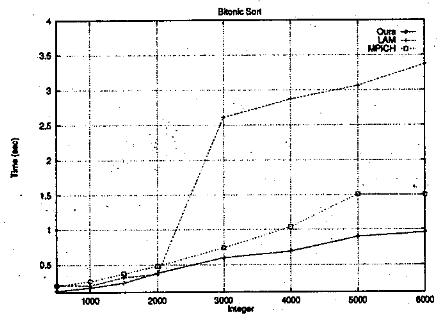

Figure 9: Bitonic Sort

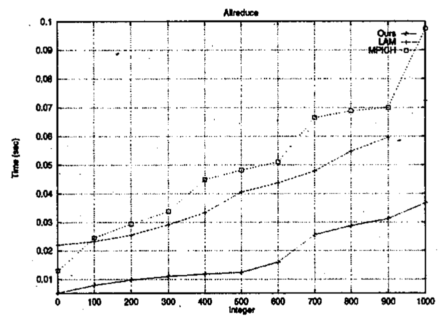

Figure 13: MPI_Allreduce

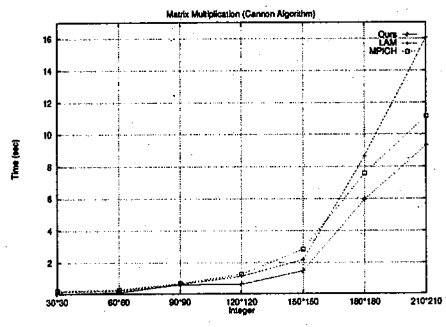

Figure 10: Matrix Multiplication

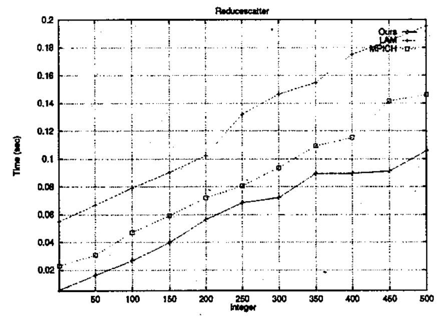

Figure 14: MPI_Reduce_Scatter

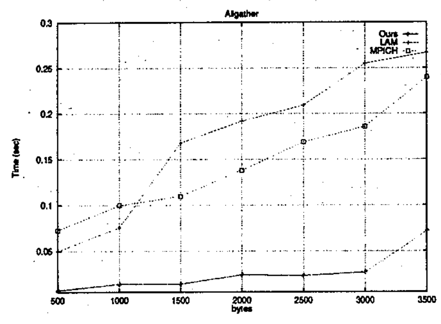

Figure 11: MPI_Allgather

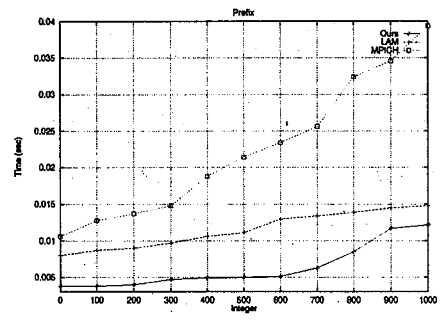

Figure 15: MPI_Scan