MPF: A Portable Message Passing Facility for Shared Memory Multiprocessors

Allen D. Malony[†]

Center for Supercomputing Research and Development
University of Illinois
Urbana, Illinois 61801

Daniel A. Reed[‡]
Patrick J. McGuire[††]

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

ABSTRACT

This paper presents the design, implementation and performance evaluation of a message passing facility (MPF) for shared memory multiprocessors. MPF is based on a message passing model conceptually similar to *conversations*. Participants (parallel processes) can enter or leave a conversation at any time. The message passing primitives for this model are implemented as a portable library of C function calls. MPF is currently operational on a Sequent Balance 21000, and several parallel applications have been developed and tested. We present several simple benchmark programs to establish interprocess communication performance for common patterns of interprocess communication. Finally, we present performance figures for two parallel applications, linear systems solution and iterative solution of partial differential equations.

[†] Supported in part by NSF Grant Nos. NSF DCR 84-10110 and NSF DCR 84-06916, DOE Grant No. DOE DE-FG02-85ER25001, and a donation from IBM.

[‡] Supported in part by NSF Grant No. NSF DCR 84-17948 and NASA Contract No. NAG-1-613.

[††] Present address: Hewlett Packard Laboratories, Palo Alto, CA.

1. Introduction

Historically, programming models for parallel processing have largely been architecture dependent. The absence of shared memory, for example, typically promotes message passing as a computing paradigm. In contrast, software for shared memory multiprocessors encourages the use of shared variables for inter-process communication and synchronization. Although the reflection of the underlying architecture in the programming support software encourages efficient use of the hardware, it constrains the programmer to a single world view. Because certain algorithms are naturally expressed in either shared memory or message passing forms, the programmer is forced to adapt the algorithm formulation to the available programming model. Unfortunately, this adaptation may incur a substantial performance penalty.

Snyder [Snyd86] has argued eloquently that we must develop a suitable set of *type architectures* that elide unnecessary architectural details while retaining those necessary to reflect the performance constraints imposed by the hardware. These type architecture abstractions would, for example, permit an algorithm designer to accurately estimate the performance penalties when moving from one type architecture to another. Unfortunately, no such abstractions and performance models yet exist.

To investigate the performance tradeoffs between the shared memory and message passing paradigms, we have used the existing primitives on a shared memory machine to develop a message passing facility. With the versatility of shared memory machines, the message passing primitives are easily implemented. In contrast, realizing the functionality of shared memory on message passing architectures is considerably more

difficult.[1]

This paper presents the design, implementation and performance of a general message passing facility (MPF) for shared memory multiprocessors. As stated earlier, the motivation for this work is not merely to produce a message passing implementation, but also to explore the problems and performance penalties of cross–architecture algorithm ports. esh 1 "The MPF Message Passing Model"

To assess the advantages, disadvantages, and performance penalties of message passing on a shared memory architecture, it is crucial that the implementation be based on a fully general message passing model. Hence, the MPF message passing model is generic, independent of that supported by any vendors of message passing machines (e.g., Intel [Ratt85]). We develop the notion of *logical, named virtual circuits* as a basis for the MPF message passing semantics.

A virtual circuit is, according to standard definition, a *logical* connection between two communicating entities; the physical realization is unspecified. In the MPF model, the communicating entities are *sets* of processes whose membership can change during the lifetime of a virtual circuit. Because processes can join or leave virtual circuits, messages are directed to a virtual circuit, not individual participants. By defining names for virtual circuits, participants can join or leave the associated *conversations* [CoPe86]; clearly, these mutually selected names must be unique. The resulting abstraction is a logical, named virtual circuit (LNVC).

It is important to understand that LNVC's provide a fully general communication paradigm. Conversations, by analogy with everyday life, include dialogue, group

[1] This asymmetry is the crux of the type architecture notion.

discussions, and lectures. Equivalently, LNVC's support both bi–directional and unidirectional communication with two or more participants. To permit specification of a particular conversation type, we must define communications protocols for LNVC's.

Each process that is a member of an LNVC conversation is either a message sender or receiver, or both;[2] see Figure 1. Message receivers identify themselves as FCFS (first–come, first–serve) or BROADCAST when they join the conversation. If there is a single message receiver in a conversation, FCFS and BROADCAST are equivalent. However, when multiple message receivers exist, only one FCFS receiver will receive each message. In contrast, all BROADCAST receivers receive all messages. Both FCFS and BROADCAST receivers can exist simultaneously during a conversation.[3] In this case, however, a message will be sent to all BROADCAST receiving processes and to only one of the FCFS processes.

Justification for this LNVC model comes from two sources: conversation–based electronic mail [CoPe86] and distributed variables [Debe86]. Like LNVC's, conversation–based mail permits participants to enter or leave the discussion at their discretion.

In contrast, distributed variables arose as a programming paradigm for message passing systems. Intuitively, a distributed variable exists in a *name space* that is global to the processes but accessible only by a message passing protocol with associated read and write operations. In addition, a set of process interaction rules describes the semantically valid operations for each type of distributed variable. Like LNVC's, a distributed variable permits multiple readers and writers.

---

[2] An LNVC exists only if the set of senders or receivers is not null.

[3] The only restriction is that a receiving process of an LNVC cannot use both FCFS and BROADCAST protocols.

2. MPF Programming Environment

The user interface to the MPF message passing environment is a library of high-level interface routines for LNVC management, protocol establishment and message transfer. Because our current implementation is based on the C programming language, the MPF programming primitives are defined below as C function calls.

init (max_LNVC's, max_processes)

open_send (process_id, lnvc_name)

open_receive (process_id, lnvc_name, protocol)

close_send (process_id, lnvc_id)

close_receive (process_id, lnvc_id)

message_send (process_id, lnvc_id, send_buffer, buffer_length)

message_receive (process_id, lnvc_id, receive_buffer, buffer_length)

check_receive (process_id, lnvc_id)

Init() is the initialization routine for MPF. Most architecture specific initialization and allocation are done here. In particular, shared memory is allocated for LNVC's and synchronization variables are initialized for exclusive access to internal data structures. The parameters *max_LNVC's* and *max_processes*, the maximum number of LNVC's and processes, respectively, are used to estimate the amount of shared memory necessary.

Open_send() establishes a send connection for the process *process_id* on the LNVC *lnvc_name*. If *lnvc_name* did not previously exist, it is created. The integer returned by **open_send()** is MPF's internal LNVC identifier for *lnvc_name*. This identifier must be used in the **message_send()** and **close_send()** routines.

Open_receive() establishes a receive connection for the process *process_id* on the LNVC *lnvc_name* with the communication protocol *protocol* (FCFS or BROADCAST). Again, if *lnvc_name* did not previously exist, it is created. An integer return value specifies MPF's internal LNVC identifier for *lnvc_name* for use in **close_receive()**, **message_receive()** and **check_receive()**.

Close_send() and **close_receive()** remove send and receive connections, respectively, for process *process_id* on LNVC *lnvc_id*. If this is the last process connected to *lnvc_id*, the LNVC is deleted and all unread messages are discarded.

Message_send() transfers a message from process *process_id* to the LNVC *lnvc_id*. The contents are taken from *(char *) send_buffer* and the message length is *buffer_length* bytes. Message sending is asynchronous, allowing a process to proceed before the message reaches its destination(s). **Message_receive()** transfers a message from LNVC *lnvc_id* to the process *process_id* into the receive buffer starting at *(char *) receive_buffer*. *Buffer_length* is set to the number of bytes transferred. **Message_receive()** is blocking; it returns only after a message has been received.

Check_receive() allows process *process_id* to check for the existence of any messages in LNVC *lnvc_id*. A non-zero return value indicates the existence of a message. If the receive connection is BROADCAST, the message is guaranteed to be present when a **message_receive()** is executed. However, a process with a FCFS receive connection must use **check_receive()** with caution. Although **check_receive()** may indicate that a message is present, another process with a FCFS receive connection for *lnvc_id* may acquire the message before the checking process can receive the message.

These MPF programming primitives provide the user a high-level interface to the MPF message passing model. The following section describes the implementation of the

MPF programming environment for a shared memory multiprocessor system.

3. MPF Implementation

Intuitively, one would expect a significant performance and programming overhead to realize LNVC conversations. Our implementation experience on a Sequent Balance 21000 suggests that this is not the case. The MPF run-time support is only a few hundred lines of C code, and the only system dependent code involves shared memory allocation and synchronization. MPF could be easily ported to any system providing these facilities. The remainder of this section describes the underlying MPF implementation and the motivation for the choice of MPF primitives.

3.1. Data Structures

Dynamically linked data structures are used extensively in the MPF implementation for programming flexibility. The fundamental data structure is the MPF *message*. During MPF initialization, a free list of linked message blocks is created in shared memory[4]. Space allocated from this free list is used for messages during program execution. Messages are composed of linked message blocks together with a header for saving pertinent message information (e.g., message length, a pointer to the tail, and a pointer to the next message in a list of messages for an LNVC). During execution of a **message_send()**, the sending buffer is copied into the message block data fields. The message is then copied into the receiving buffer as part of the **message_receive()** operation.

[4] In all of our experiments, 16 byte message blocks were used.

The key MPF design problem was identifying an effective data structure for an LNVC. Virtual circuits provide time–ordered message delivery [Tane81]. LNVC's behave similarly. Because shared memory multiprocessors provide a global clock and access to message buffers is serialized by synchronization primitives, most complications arising in a distributed memory context are avoided. A time–ordered message stream will be seen by all BROADCAST receiving processes. In contrast, a FCFS receiving process will see only a part of the message stream. However, the sequence preserving LNVC forces a time–ordering of this sub–stream as well.

Clearly, a FIFO queue suffices to maintain sequentiality of messages between sending and receiving processes. However, each BROADCAST receive process must have its own head pointer, and the FCFS receive processes must share a head pointer. Figure 2 illustrates one possible state of an LNVC with FCFS and BROADCAST receive processes. Hence, an *LNVC descriptor* contains the LNVC name, its internal identifier, the number of queued messages, a FIFO queue implemented as a linked list of messages, a FIFO tail pointer for sending processes, a FIFO head pointer for FCFS receiving processes, a description of all connections to the LNVC, and a synchronization lock for mutual exclusive access to the LNVC descriptor. The LNVC connections are represented by *send descriptors* and *receive descriptors*, which contain the process identifier of the connected process. BROADCAST receive processes have an additional descriptor field used for individual FIFO head pointers. Like message blocks, LNVC, send, and receive descriptors are linked into free lists when not in use.

3.2. Programming Primitives

The constraints necessary to insure consistent and efficient access to the MPF data structures by concurrently executing processes, dictate the implementation of the MPF

programming primitives. Rather than describing the details of data structure manipulation and process mutual exclusion, we focus on two of the interesting design issues that arose during the implementation.

Implementing the LNVC close operations raises the fundamental question of LNVC lifetime. We generally regard an LNVC as existing only when there is a connected sending or receiving process; the current implementation is based on this principle. The semantics of the close operations state that the entire LNVC FIFO structure is discarded, including messages, if the closed sender or receiver process is the last one connected to the LNVC. However, this implementation decision has ramifications on process interaction. Some care must be taken to ensure that messages will not be lost due to unconnected processes. For instance, a sending process might want to open a send connection on an LNVC, send some messages, and then close the connection. However, if none of the processes intending to receive these messages have established a receiver connection before the closing of the sender connection, the messages could be lost when the LNVC is removed.

The **close_receive()** operation poses a particularly vexing problem. If the FIFO head pointer for a receiving process is pointing to the head of the FIFO message list and the receiver connection is closed, all messages unread by the receiver but read by all other connected receiver processes must be deleted. Unfortunately, it is difficult to determine which messages should be deleted without comparing the FIFO head pointers of all receiving processes with the starting address of each message considered.

4. MPF Experiments

To investigate the ease of use and the performance of MPF, we developed several test programs for the Sequent Balance 21000 [Sequ86]. All experiments were conducted on a machine containing 20 processors and 16 Mbytes of memory. Each Balance 21000 processor is a 10 MHz National Semiconductor NS32032 microprocessor, and all processors are connected to shared memory by a shared bus with a 80 Mbyte/s (maximum) transfer rate. Each processor has a 8K byte, write-through cache and an 8K byte local memory; the latter contains a copy of selected read-only operating system data structures and code.

With MPF on the Balance 21000, parallel programs consist of a group of Unix processes that interact using LNVC's. The shared memory used by MPF is implemented by mapping a region of physical memory into the virtual address space of each process.

Our initial experiments concentrated on verifying the LNVC implementation and establishing performance benchmarks for simple message transfer configurations. To test MPF's performance in a parallel program, we developed a message based version of the Gauss–Jordan algorithm (with partial pivoting) for solving linear systems. As an additional test, we implemented a successive over-relaxation (SOR) algorithm for solving Poisson's equation, an elliptic partial differential equation. Each of these tests is discussed in detail below.

Perhaps the simplest performance test is the throughput of an LNVC consisting of one sender and one receiver. To determine the LNVC throughput, measured in bytes per second, we designed a simple program, *base*, that establishes a loop-back connection through an LNVC for a single process, and then alternates between sending and receiving fixed-length messages. Figure 3 shows the performance as a function of message length.

Although throughput increases with increasing message length, it approaches an asymptote. Detailed measurements show that, for large messages, LNVC updates are of negligible cost. Instead, message copying costs dominate; memory bandwidth is the performance limiting factor.

Although the *base* benchmark defines the byte transfer rate limit between a sender and one receiver of an LNVC, typical applications involve many processes. The message transfer rate for parallel programs depends on the relative amounts of FCFS and BROADCAST communication. However, it is possible to compare the performance of a set of parallel processes that use FCFS LNVC's to a similar set using BROADCAST LNVC's.[5] Hence, we developed two synthetic parallel programs, *fcfs* and *broadcast*. The program *fcfs* uses one process to send messages of length K to an LNVC with N FCFS receiving processes. The program *broadcast* is similar except the receiving processes are of type BROADCAST. Figures 4 and 5 show the message transfer rates for *fcfs* and *broadcast*, respectively.

The benefit of larger messages is evident in both Figures 4 and 5. However, the throughput for *fcfs* is very different from *broadcast*. With the *fcfs* benchmark, only one FCFS process can receive each message. Hence, the total message throughput is limited by the message transmission rate. The decreasing throughputs for 16–byte and 128–byte messages are caused by increased LNVC contention with additional receiver processes. For larger messages, this contention is masked by message copying costs.

The throughputs for the *broadcast* benchmark illustrate the MPF support for concurrent **message_receive()** operations by BROADCAST receivers. Although the

[5] In a FCFS LNVC, all receiving processes are FCFS, and there are no BROADCAST receive connection. A BROADCAST LNVC is the converse.

actual message transmission rate is unchanged from the *fcfs* benchmark, all message receivers obtain a copy of each message. Thus, by allowing the receiver processes to copy messages concurrently, higher throughputs can be achieved. The maximum attainable *broadcast* throughput is limited by the concurrent efficiency of MPF, as well as the memory bandwidth. MPF achieved an effective throughput of 687,245 bytes per second for 1024–byte messages and 16 receiving processes. As with the *fcfs* benchmark, message throughput is sub–linear with the number of processes when the message length is small; contention is again the reason.

As a final throughput benchmark, we constructed a synthetic program whose processes can each send to and receive from all other processes. The communications pattern is fully–connected with a FCFS LNVC defined for each destination process. In this benchmark, each process sends a specified number of fixed–length messages; destinations are selected randomly. Each time a process executes a **message_send()**, it then receives all messages that are queued in its LNVC.

Figure 6 shows the results obtained with this benchmark program. As expected, throughput increases as message length increases. More importantly, message throughput increases as additional processes are added to the benchmark. This implies that MPF can support concurrent operation on multiple LNVC's. We expect increasing overhead with more processes, however, and this is evident in the decreasing slope of the throughput curves.

When a large number of processes are transmitting large messages, MPF must allocate a large amount of memory for message buffers. The larger the memory requirements for message transfer, the more susceptible MPF performance is to virtual memory overheads. For 1024–byte messages, paging overhead increases rapidly for more

than 10 processes; this is the reason for the decrease in observed throughput. Paging overheads are also significant for 256-byte messages but do not occur until their are 20 active processes. Similar behavior would likely occur for smaller message sizes if the Balance 21000 had additional processors.

As an application test program, the Gauss-Jordan algorithm (with partial pivoting) for solving linear systems is ideal; it contains both one-to-one and broadcast communications. The Gauss-Jordan algorithm converts the linear system $Ax = b$, where A is non-singular, to the equivalent linear system $A'x = b'$ where A' is diagonal. The parallel implementation of this algorithm partitions the matrix A into equal sized groups of contiguous rows; each partition is assigned to a process. Each process searches for the maximum element in the current column, and sends this value to an arbiter process. The arbiter process identifies the maximum of the maxima, and advises the process holding this value. The identified process broadcasts the selected pivot row to all other processes. The processes then sweep the rows of their partition using this pivot row and begin a new iteration.

Figure 7 shows the speedup for the Gauss-Jordan algorithm as a function of matrix size and the number of processors. Speedup is greater with larger matrices; this is the classic computation versus communication balance faced by message-passing systems. As noted above, there are two types of communication: selecting a pivot row and broadcasting that pivot row. Increased parallelism increases the number of FCFS messages sent to the arbiter process during pivot selection. Similarly, increased parallelism means additional processes must capture the pivot row as it is broadcast. Conversely, increased parallelism decreases the number of matrix rows assigned to each task. Hence, the computation per process decreases while the communication cost

increases. In the extreme, excessive parallelization yields insufficient computation per iteration, and speedup declines. Larger matrices permit effective use of more processors. The most important conclusion to be drawn from Figure 7 is that real speedups *can* be obtained in the MPF environment.

As a final test of the flexibility of the MPF programming environment, we adapted a parallel, elliptic partial differential equations solver, written for a hypercube [Ratt85]. The solver iterates over a grid of points, using successive over–relaxation (SOR), until the grid converges to a solution of the partial differential equation. If the grid of points contains $P \times P$ points, it is partitioned into $N \times N$ subgrids of size $\frac{P}{N} \times \frac{P}{N}$. Each subgrid is assigned to a processor, and each processor iterates over its subgrid. On each iteration, the boundaries of each sub–grid must be exchanged with the four neighboring processors. In addition, the processors determine if the local sub–grid has converged and send this status information to a monitoring process. Because the computation cost for an iteration is proportional to the area of the sub–grids, and the communication cost is proportional to their perimeter, the computation/communication ratio can be adjusted by varying the number of processors.

Porting the hypercube program to MPF was very simple. The interprocess communication among neighbors corresponds naturally to FCFS LNVC's. Similarly, BROADCAST LNVC's were used to broadcast convergence information from the monitoring process. Figure 8 shows speedup as a function of grid size and number of processes.[6]

---

[6] Because no equivalent, sequential solver was available, all speedups are shown relative to the smallest parallel solver: 4 processes.

Certainly, more experimentation is necessary to explore the usefulness of MPF for real message passing applications development. However, the experimental programs and results presented above are encouraging as an indication of the programming flexibility of the MPF environment and MPF's ability to support concurrent operation.

5. Conclusion

A message passing environment for shared memory multiprocessors is interesting for several reasons. As a parallel programming paradigm conceptually different from the shared memory approach, message passing offers the user a different programming alternative. A particularly interesting benefit of a message passing facility for shared memory machines is the ability to develop a program using a hybrid parallel programming paradigm.

MPF supports the paradigm with a general message passing model and an implementation that hides the details of the underlying message communications. Programs destined for message passing systems can be easily prototyped in the MPF environment. The versatility of a shared memory machine provides a flexible implementation base for a message passing facility. The Sequent Balance 21000 MPF implementation takes only 800 lines of heavy-commented C code and adds 7000 bytes to a user's program. Furthermore, the implementation is completely portable between shared memory multiprocessors that provide locking and memory sharing between concurrently executing processes.

The MPF implementation discussed in this paper has clear inefficiencies resulting from the full support of the general message passing model. One method to improve the performance of the MPF system is to restrict the generality of message communication

and process interaction. Clearly, simpler and more efficient implementations can result. For instance, to support synchronous message passing, copying of data from a sending buffer to a linked message buffer and then to the receiving buffer is unnecessary; direct data transfer is possible. Furthermore, if only one–to–one communication is implemented, all locking associated with message handling is removed. Studies of simplified message passing systems for shared memory multiprocessors are currently underway. One important research issue with these systems is the effect of the parallel programming paradigm (message passing or shared memory) on application performance.

6. Acknowledgments

Jack Dongarra and the Advanced Computing Research Facility of Argonne National Laboratory graciously provided both advice and access to the Sequent Balance 21000.

References

[CoPe86] D. E. Comer and L. L. Peterson, "Conversation-Based Mail," *ACM Transactions on Computer Systems,* Vol. 4, No, 4, pp. 299–319, November 1986.

[Debe85] E. P. Debenedictis, "Multiprocessor Programing with Distributed Variables," *Proceedings of the First Conference on Hypercube Multiprocessors,*" 1986, SIAM Press, pp. 70–86.

[Duni86] T. H. Dunigan, "A Message-Passing Multiprocessor Simulator," Oak Ridge National Laboratory, Technical Report No. ORNL/TM-9966, May 1986.

[Fuji83] R. M. Fujimoto, "*SIMON: A Simulator for Multicomputer Networks,*" University of California at Berkeley, Computer Science Division, Report No. UCB/CSD83/140, September 1983.

[Hoar78] C. A. R Hoare, "Communicating Sequential Processes," *Communications of the ACM,* August 1978, pp. 666–667.

[LeBl86] T. J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," *Proceedings of the International Conference on Parallel Processing,* August 1986, pp. 463–466.

[PuRG86] J. Purtilo, D. A. Reed, D. C. Grunwald, "Environments for Prototyping Parallel Algorithms," University of Illinois, Department of Computer Science, Technical Report No. UIUCDCS-R-86-1249, February 1986.

[Ratt85] J. Rattner, "Concurrent Processing: A New Direction in Scientific Computing, *Conference Proceedings of the 1985 National Computer Conference,* AFIPS Press, Vol. 54, pp. 157–166, 1985.

[Snyd86] L. Snyder, "Type Architectures, Shared Memory and the Corollary of Modest Potential," University of Washington, Department of Computer Science, Technical Report No. TR 86-03-04, 1986.

[Sequ86] Sequent Computer Systems, *Guide to Parallel Programming on Sequent Computer Systems,* 1986.

[Tane81] A. S. Tanebaum, *Computer Networks,* Prentice Hall, Englewood Cliffs, New Jersey, 1981.

Figure 1
MPF Message Passing Model

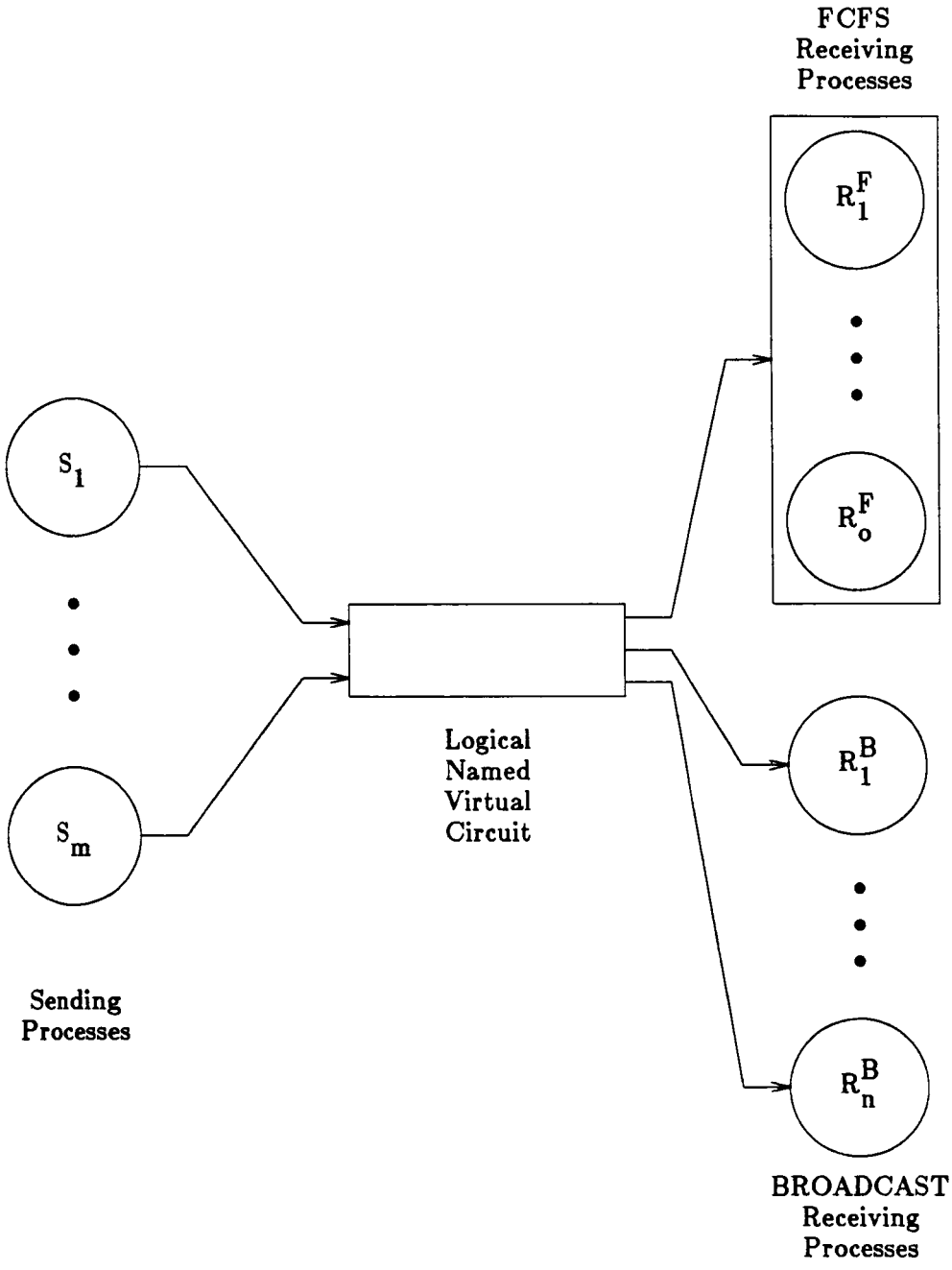

Figure 2
Possible State of an LNVC

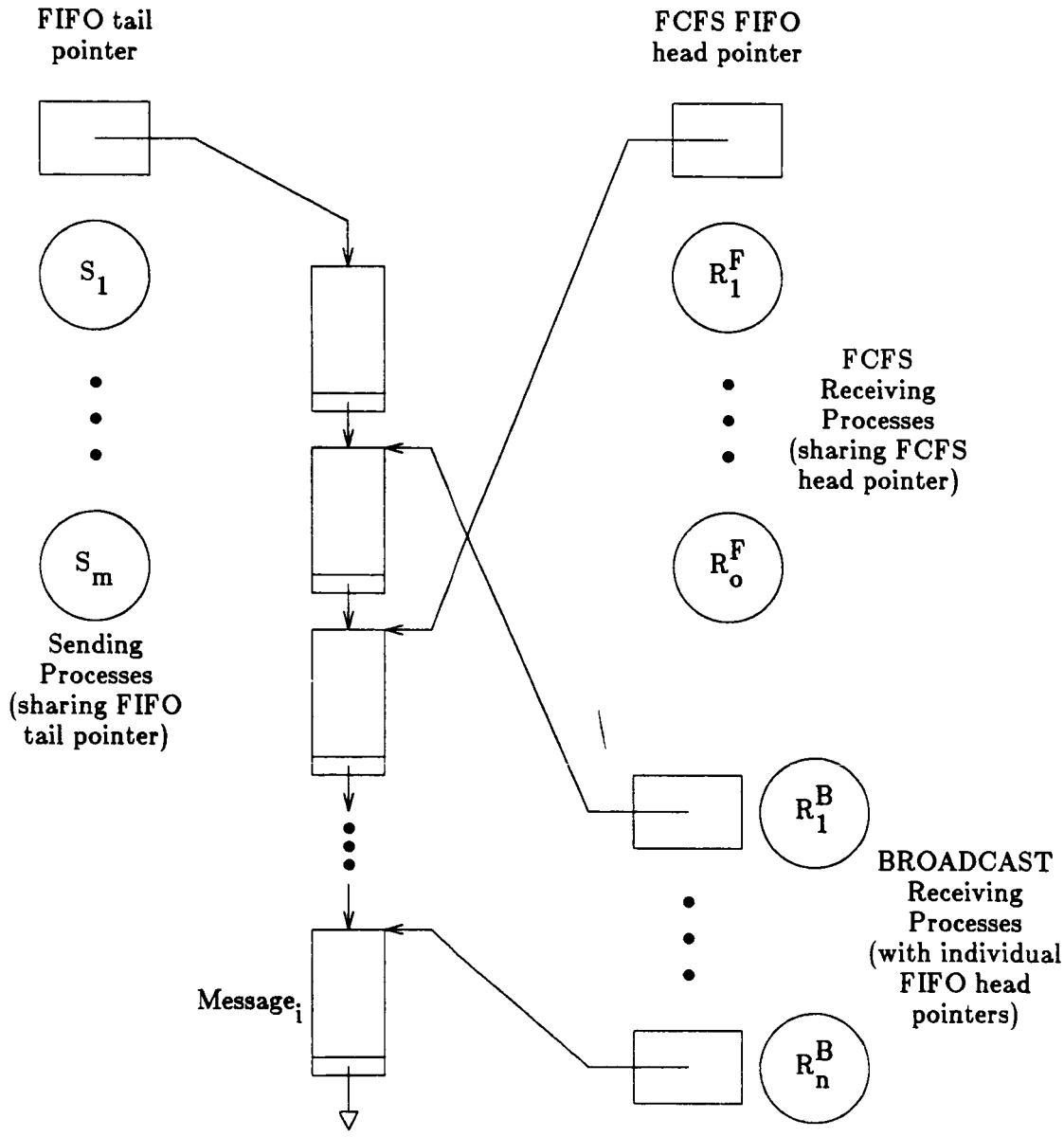

Figure 3
Base Benchmark
Throughput vs. Message Length

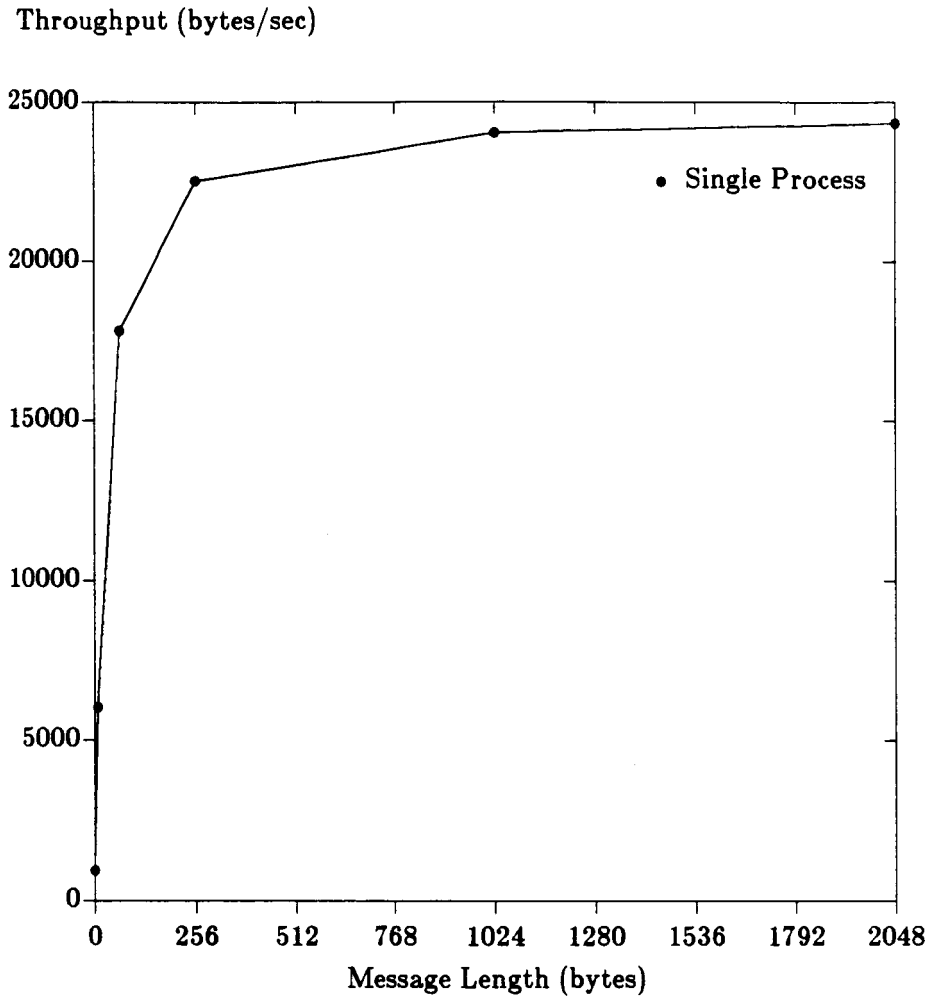

Figure 4
Fcfs Benchmark
Throughput vs Receiving Processes

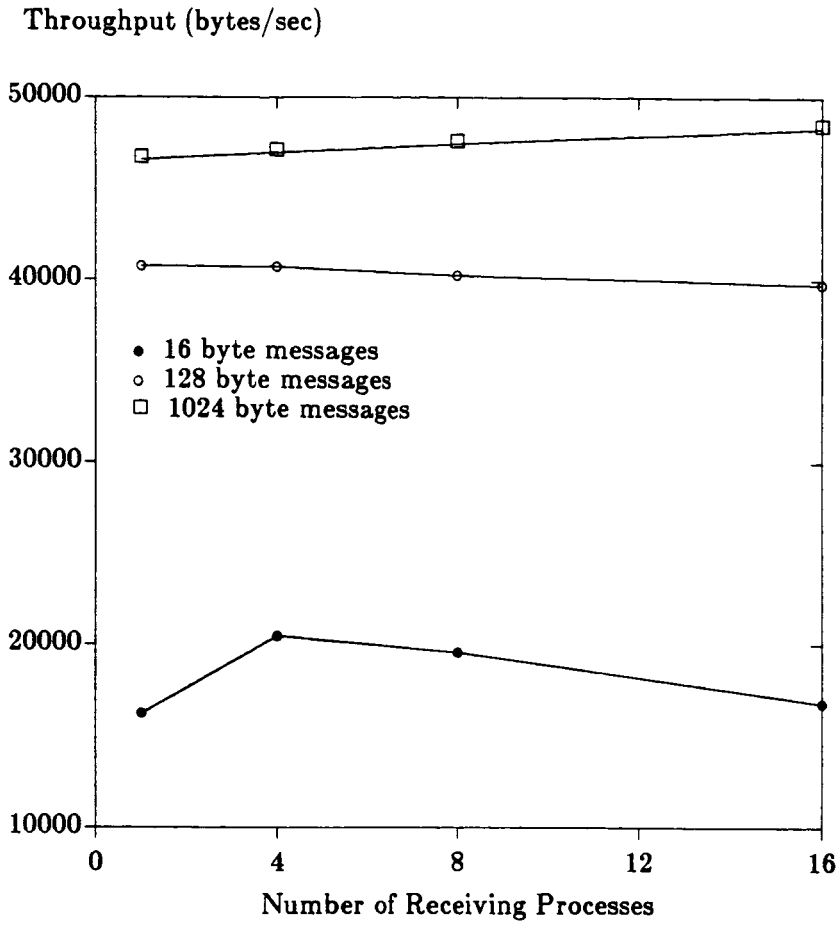

Figure 5
Broadcast Benchmark
Throughput vs Receiving Processes

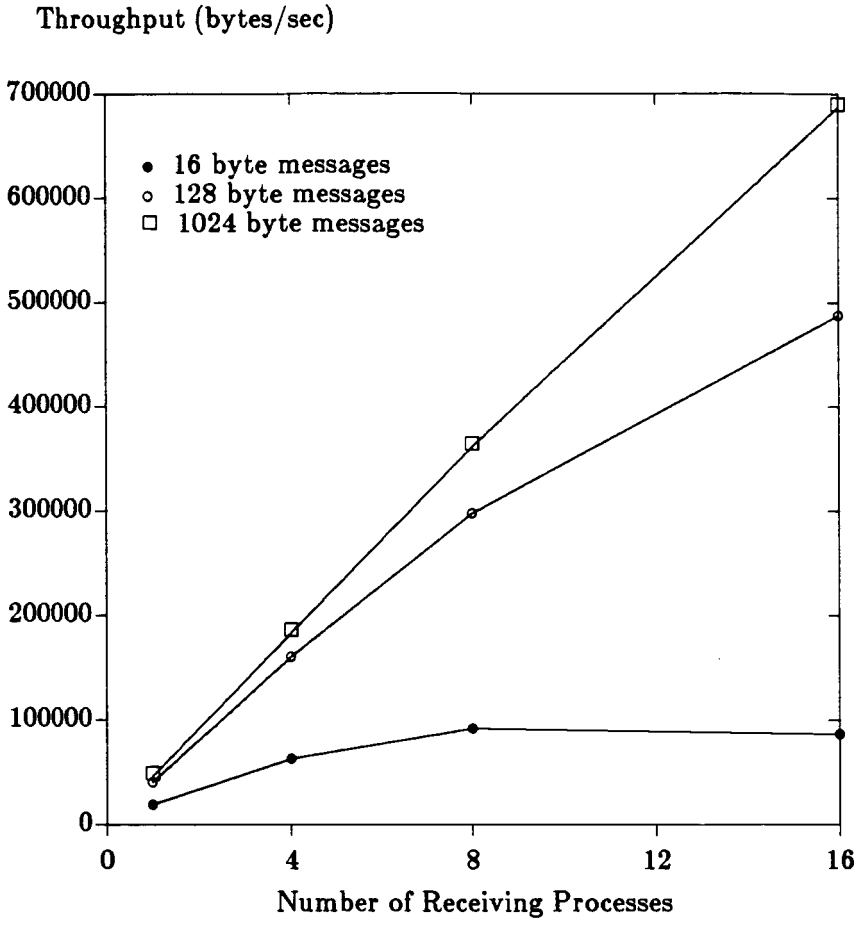

Figure 6
Random Benchmark
Throughput vs Processes

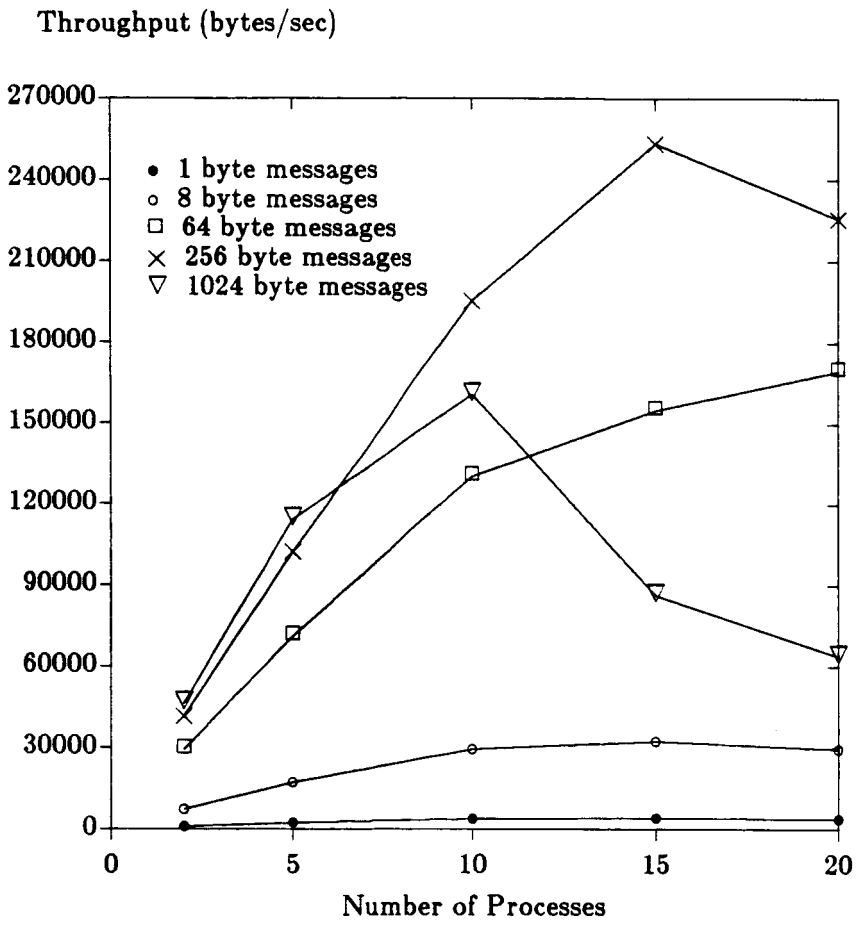

Figure 7
Gauss Jordan
Speedup vs. Processes

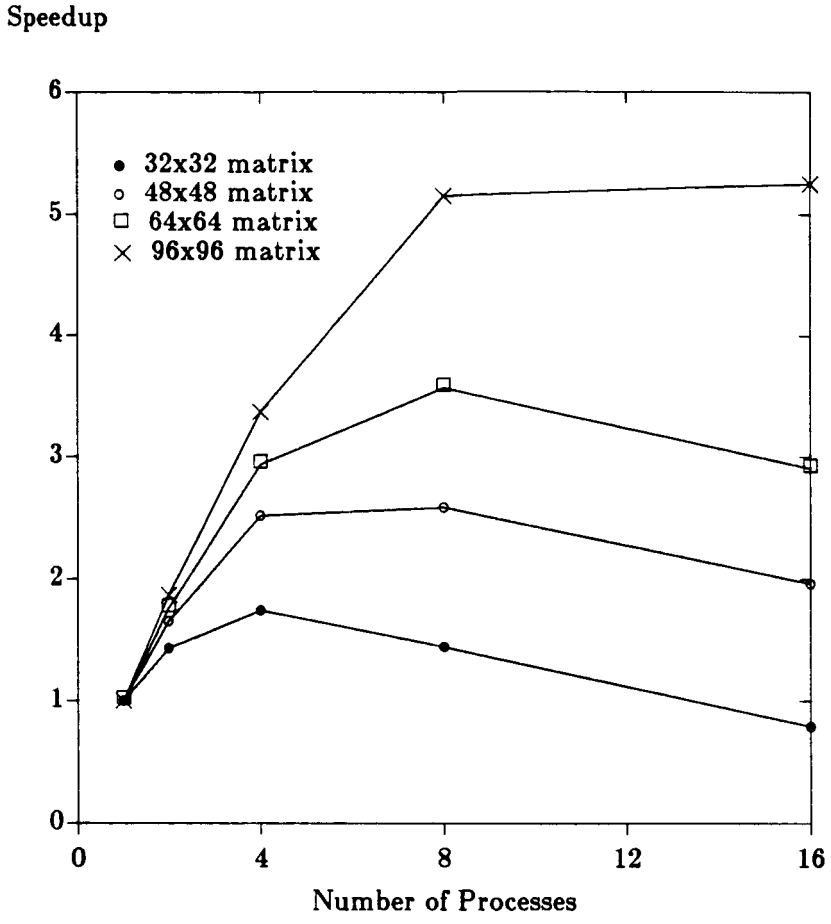

Figure 8
Poisson Elliptic PDE Solver with SOR Iterations
Per Iteration Speedup vs. Dimension (N)