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 — A message passing facility (MPF) for shared memory multiprocessors is presented. MPF is based on a message passing model conceptually similar to *conversations*. The message passing primitives for this model are implemented as a portable library of C function calls. The performance of interprocess communication benchmark programs and two parallel applications are given.

1. Introduction

Historically, programming models for parallel processing have largely been architecture dependent. The underlying architecture reflected in the programming support software encourages efficient use of the hardware, but constrains the programmer to a single world view. Although some algorithms are naturally expressed in a certain type of programming paradigm, the algorithm formulation must be adapted to the available programming model. Unfortunately, this adaptation may incur a substantial performance penalty.

Snyder [4] has argued for a 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 permit an algorithm designer to accurately estimate the performance penalties when moving from one type architecture to another. To investigate the performance tradeoffs between the shared memory and message passing paradigms, we developed a message passing facility (MPF) using the existing primitives of a shared memory machine.

2. The MPF Message Passing Model

To assess the advantages and disadvantages of message passing on a shared memory architecture, it is crucial that the implementation be based on a fully general message passing model. We develop the notion of *logical, named virtual circuits* as a basis for the MPF message passing semantics.

A virtual circuit is a *logical* connection between two communicating entities. In the MPF model, the communicating entities are *sets* of processes whose membership can change during the lifetime of a virtual circuit. For this reason, messages are directed to a virtual circuit, not individual participants. By defining names for virtual circuits, participants can join or leave the associated *conversations* [1]. The resulting abstraction is a logical, named virtual circuit (LNVC).

LNVC's provide a fully general communication paradigm. Each process that is a member of an LNVC conversation is either a message sender or receiver, or both; see Figure 1. Message receivers identify themselves as FCFS (first–come, first–serve) or BROADCAST when they join the conversation. Only one FCFS receiver will receive each message, whereas, all messages are received by each BROADCAST receiver. Both FCFS and BROADCAST receivers can exist simultaneously during a conversation.

Justification for this LNVC model comes from two sources: conversation–based electronic mail [1] and distributed variables [2]. 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. A distributed variable supports multiple readers and writers, as well as, general process interaction semantics.

3. MPF Programming Environment

The MPF user interface is a library of C interface routines for LNVC management, protocol establishment and message transfer. The programming primitives are:

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_buff, length)
message_receive (process_id, lnvc_id, receive_buff, length)
check_receive (process_id, lnvc_id)

Init() is the initialization routine for MPF. 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 that must be allocated.

Open_send() establishes a send connection for the process *process_id* on the LNVC *lnvc_name*. **Open_receive()** establishes a receive connection for the process *process_id* on the LNVC *lnvc_name* with the communication protocol *protocol* (FCFS or BROADCAST). **Close_send()** and **close_receive()** remove send and receive connections, respectively.

Message_send() transfers a message from process *process_id* to the LNVC *lnvc_id*. Message sending is asynchronous; a process can proceed before the message reaches its destination(s). **Message_receive()** transfers a message from LNVC *lnvc_id* to the process *process_id*. **Message_receive()** is blocking; it returns only after a message has been received. **Check_receive()** checks for the existence of any messages in LNVC *lnvc_id*.

4. 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 only system dependent code involves shared memory allocation and synchronization allowing MPF to be easily ported to any system providing these facilities.

4.1. Data Structures

The fundamental data structure is the MPF *message*. During MPF initialization, a free list of linked message blocks is created in shared memory. Messages are composed of linked message blocks together with a header for saving pertinent message information. 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.

The key MPF design problem was identifying an effective data structure for an LNVC. Time–ordered message delivery must be supported, and BROADCAST and FCFS receiving processes must be effectively managed. A FIFO queue suffices to maintain sequentiality of messages between sending and receiving processes. Each BROADCAST receive process will have its own queue head pointer, and the FCFS receive processes share a head pointer.

[†] 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.

4.2. 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 comment on one interesting design issue 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.

5. MPF Experiments

To investigate the ease of use and the performance of MPF, we developed several test programs for the Sequent Balance 21000 [5]. The 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.

To determine the throughput characteristics of a single LNVC, we designed a simple program that establishes a loop-back connection through an LNVC for a single process, and alternates between sending and receiving fixed-length messages. Throughput increases as message length increases because the fixed-time LNVC updates decrease in relative cost. However, copying overhead limits throughput for large messages. A throughput of 25,000 bytes per second was achieved with 2048-byte messages.

In general, the message transfer rate for parallel programs depends on the relative amounts of FCFS and BROADCAST communication. However, it is possible to compare the throughput performance of a set of parallel processes that use FCFS LNVC's to a similar set using BROADCAST LNVC's; Figures 2 and 3 show the results. Because only only one FCFS process can receive each message, the FCFS benchmark 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 BROADCAST throughputs illustrate MPF's support for concurrent message_receive() operations by BROADCAST receivers. Although the 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.

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.

Figure 4 shows the results obtained with this benchmark program. Although message throughput increases as additional processes are added to the benchmark, overhead also increases resulting 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.

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 parallel implementation of this algorithm partitions a 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 it to an arbiter process. The arbiter process identifies the global maximum, 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 4 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. 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 5 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 [3]. 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 $(P/N) \times (P/N)$. 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 simple. The interprocess communication among neighbors corresponds naturally to FCFS LNVC's. Similarly, BROADCAST LNVC's were used to broadcast convergence information. Figure 6 shows speedup as a function of grid size and number of processes; all speedups are relative to the smallest parallel solver: 4 processes.

6. 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. Furthermore, the MPF implementation is completely portable between shared memory multiprocessors that provide locking and memory sharing between concurrently executing processes.

7. 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

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

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

[3] 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.

[4] 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.

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

Figure 1
MPF Message Passing Model

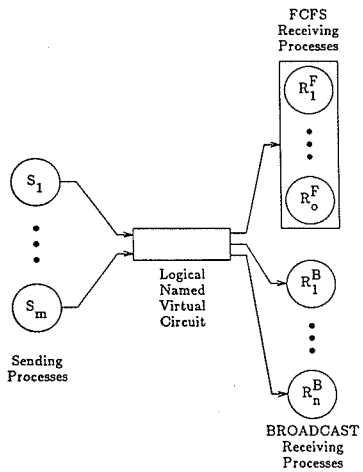

Figure 2
Fcfs Benchmark
Throughput vs Receiving Processes

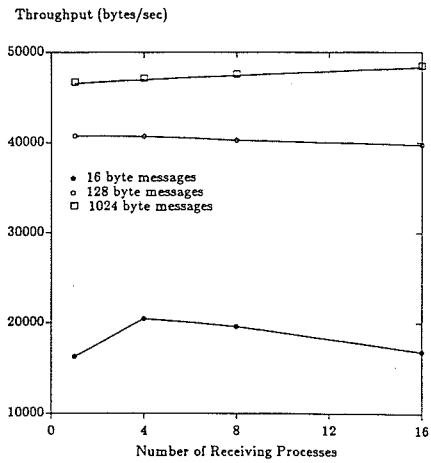

Figure 3
Broadcast Benchmark
Throughput vs Receiving Processes

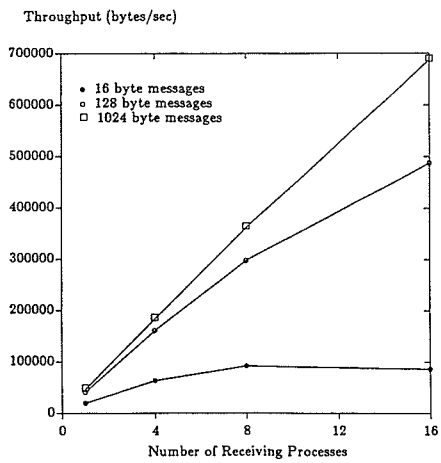

Figure 4
Random Benchmark
Throughput vs Processes

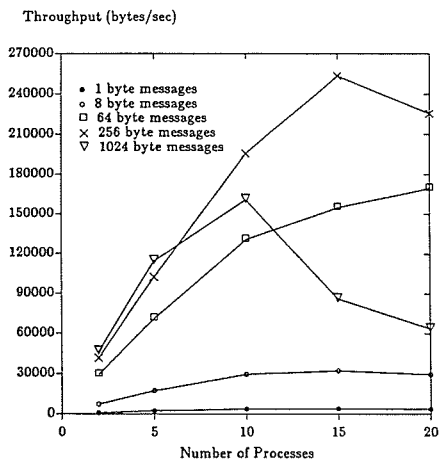

Figure 5
Gauss Jordan
Speedup vs. Processes

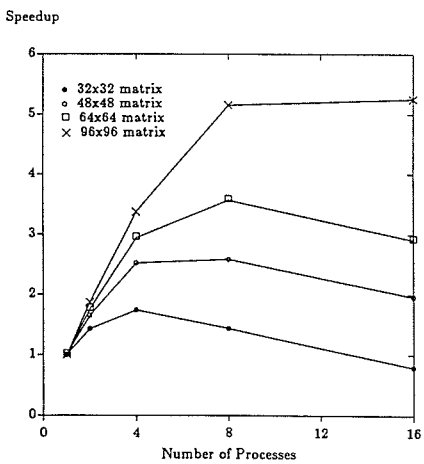

Figure 6
Poisson Elliptic PDE Solver with SOR Iterations
Per Iteration Speedup vs. Dimension (N)